

Bash în profunzime

De la linia de comandă la automatizare și internals

Această carte nu a fost scrisă pentru a memora răspunsuri. A fost scrisă pentru a înțelege mecanisme. Un examen poate dura câteva ore. Înțelegerea durează o viață.

Cui se adresează această carte?

Această carte nu este destinată să înlocuiască un curs introductiv de Linux.

Ea este destinată cititorului care știe deja să folosească sistemul și dorește să înțeleagă mecanismele din spatele comenzilor pe care le utilizează zilnic.

Dacă ai folosit Linux luni sau ani de zile și ai simțit vreodată că execuți comenzile corect fără să știi exact de ce funcționează astfel, această carte este pentru tine.

Scopul ei nu este să te învețe ce să tastezi.

Scopul ei este să îți explice ce se întâmplă după ce ai apăsat Enter.

Scopul acestui manual nu este să te învețe ce să tastezi.

Scopul lui este să te învețe să poți explica de ce ceea ce tastezi produce rezultatul pe care îl produce.

Capitolul 1

De ce există shell-urile?

În fiecare calculator există două lumi complet diferite.

Prima este lumea hardware-ului: procesorul, memoria, discurile, plăcile de rețea și toate celelalte componente fizice. Acestea nu înțeleg cuvinte precum „copiază”, „creează un director” sau „șterge un fișier”. Ele execută doar instrucțiuni în limbaj mașină și comunică prin registre, întreruperi și apeluri de sistem.

A doua este lumea utilizatorului. Oamenii gândesc în termeni de fișiere, directoare, documente și aplicații. Ei au nevoie de un mod simplu de a exprima intenții, fără să cunoască detaliile interne ale sistemului de operare.

Între aceste două lumi există un intermediar.

Acesta este **shell-ul**.

Shell-ul nu este sistemul de operare și nici kernelul. El este un program care interpretează comenzile utilizatorului și le transformă într-o serie de acțiuni pe care kernelul le poate executa. Când scrii:

```
ls -l /etc
```

procesorul nu înțelege această comandă. Nici kernelul nu primește textul exact așa cum l-ai tastat. Bash citește linia de comandă, o analizează, separă argumentele, efectuează expansiunile necesare, caută executabilul `ls`, pregătește mediul de execuție și, în final, solicită kernelului să pornească programul prin apeluri de sistem precum `fork()` și `execve()`.

Astfel, shell-ul devine traducătorul dintre limbajul uman și mecanismele interne ale sistemului de operare.

Această idee aparent simplă stă la baza întregului ecosistem Unix și Linux. Filosofia Unix spune că programele trebuie să facă un singur lucru și să-l facă bine. Shell-ul este instrumentul care le conectează între ele, permițând construirea unor soluții complexe din componente mici și reutilizabile.

Mai mult decât o interfață, shell-ul este un limbaj de programare. Orice comandă pe care o tastezi poate deveni parte a unui script, iar orice script poate automatiza sarcini repetitive, administra sute de servere sau controla infrastructuri întregi. Din acest motiv, Bash nu este doar un interpretor de comenzi, ci unul dintre cele mai importante instrumente din lumea Linux.

BASH ÎN PROFUNZIME

De la linia de comandă la automatizare și internals

Capitolul 1

De ce există shell-ul?

„The shell is both a command language and a programming language.”
— Documentația GNU Bash

1.1 Omul și calculatorul vorbesc limbi diferite

Dacă ai privi pentru prima dată un calculator fără monitor, tastatură sau mouse, ai vedea doar o colecție de componente electronice: un procesor, module de memorie, dispozitive de stocare și circuite care transportă impulsuri electrice. Nimic din această imagine nu sugerează existența fișierelor, directoarelor sau comenzilor pe care le folosim zilnic.

Procesorul nu știe ce este un document. Nu știe ce înseamnă un director și nici nu poate interpreta cuvinte precum **copiază**, **șterge** sau **instalează**. El execută doar instrucțiuni elementare din limbajul mașină, reprezentate prin șiruri de biți.

La polul opus se află omul.

Oamenii gândesc în concepte. Ei vor să spună:

„Arată-mi conținutul acestui director.”

sau

„Copiază acest fișier.”

sau

„Pornește serverul web.”

Niciuna dintre aceste propoziții nu are sens pentru procesor.

Apare astfel o problemă fundamentală: **cum comunică două lumi care vorbesc limbi complet diferite?**

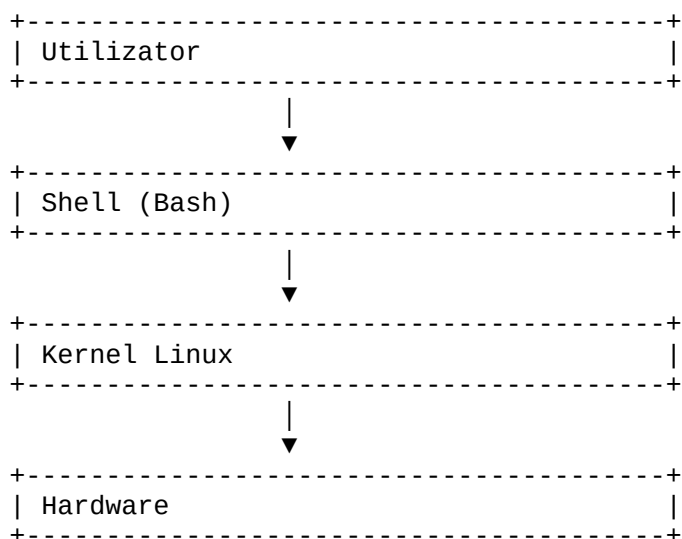
Răspunsul este unul dintre cele mai elegante concepte din istoria sistemelor de operare: **shell-ul**.

1.2 Ce este, de fapt, un shell?

Cuvântul *shell* înseamnă, în limba engleză, **coajă** sau **înveliș**.

Denumirea nu este întâmplătoare.

Imaginează-ți sistemul de operare ca pe un nucleu aflat în centrul mai multor straturi concentrice.



Kernelul este partea centrală a sistemului de operare. El gestionează procesorul, memoria, dispozitivele de stocare, plăcile de rețea și toate celelalte resurse hardware.

Dar kernelul nu este conceput pentru a comunica direct cu utilizatorii.

Rolul acesta îi revine shell-ului.

Shell-ul reprezintă stratul prin care utilizatorul transmite intențiile sale către sistemul de operare. El citește comenzile introduse, le interpretează, pregătește mediul de execuție și solicită kernelului să execute operațiile necesare.

Din acest motiv spunem că shell-ul este **interfața dintre om și sistemul de operare**.

Totuși, această definiție este incompletă.

Un shell modern, precum Bash, este în același timp:

- un interpretor de comenzi;
- un limbaj de scripting;
- un mediu de automatizare;
- un orchestrator al proceselor;
- un instrument pentru administrarea sistemelor Unix și Linux.

Cu alte cuvinte, Bash nu este doar un program care execută comenzi, ci un mediu complet de lucru.

1.3 O analogie

Imaginează-ți un director de orchestră.

El nu produce sunet. Nu cântă la vioară, nu suflă în trompetă și nu bate toba.

Totuși, fără el, zeci de muzicieni ar începe să cânte fiecare în ritmul propriu.

Directorul interpretează partitura și coordonează întreaga orchestră.

Într-un mod asemănător, Bash nu calculează rezultatul comenzii `ls`, nu citește conținutul unui fișier și nu trimite pachete prin rețea. Aceste activități sunt realizate de alte programe sau de kernel.

Bash coordonează execuția lor.

El decide:

- ce program trebuie pornit;
- cu ce argumente;
- în ce director;
- cu ce variabile de mediu;
- unde sunt trimise intrările și ieșirile;
- dacă rezultatul unei comenzi trebuie transmis alteia printr-un *pipe*;
- dacă execuția trebuie să aibă loc în prim-plan sau în fundal.

În acest sens, Bash seamănă mai degrabă cu un dirijor decât cu un instrumentist.

1.4 De ce nu comunică utilizatorul direct cu kernelul?

Întrebarea pare firească.

Dacă kernelul este cel care controlează întregul sistem, de ce nu îi trimitem comenzile direct?

Răspunsul este simplu: **kernelul nu este proiectat să interpreteze limbajul uman.**

Kernelul oferă un set de servicii accesibile prin **apeluri de sistem** (*system calls*). Aceste apeluri reprezintă o interfață de programare destinată aplicațiilor, nu utilizatorilor.

Atunci când tastezi:

```
ls -l /etc
```

kernelul nu primește șirul de caractere `ls -l /etc`.

Mai întâi, Bash:

1. citește linia introdusă;
2. identifică numele comenzii și argumentele;
3. verifică dacă este un alias, o funcție, un built-in sau un program extern;
4. aplică expansiunile necesare;
5. configurează eventualele redirecționări și pipe-uri;
6. creează un nou proces, dacă este cazul;
7. solicită kernelului încărcarea executabilului.

Abia în acel moment kernelul preia controlul și execută programul cerut.

Această separare a responsabilităților este unul dintre motivele pentru care sistemele Unix sunt atât de flexibile: kernelul rămâne concentrat pe administrarea resurselor, iar shell-ul se ocupă de dialogul cu utilizatorul.

Caseta „În spatele cortinei”

Ce înseamnă, de fapt, „interpretor de comenzi”?

Termenul *interpretor* poate crea confuzie.

Bash nu traduce comenzile în limbaj mașină, așa cum un compilator traduce un program C.

El citește textul introdus, îi analizează structura, identifică elementele semnificative și decide ce acțiuni trebuie întreprinse. Apoi folosește serviciile oferite de kernel pentru a executa aceste acțiuni.

Din acest motiv spunem că Bash **interpretează** comenzile.

Administrator Notes

Mulți administratori cu experiență folosesc Bash zilnic de ani de zile fără să se întrebe de ce unele comenzi, precum `cd` sau `export`, nu există ca executabile în `/usr/bin`.

Răspunsul este că aceste comenzi trebuie să modifice starea procesului Bash însuși. Dacă ar fi programe externe, modificările s-ar produce într-un proces separat și ar dispărea imediat după terminarea acestuia.

Vom demonstra acest lucru practic într-un capitol dedicat built-in-urilor.

Ce trebuie să reții

- Hardware-ul, kernelul și utilizatorul aparțin unor niveluri diferite ale sistemului.
 - Shell-ul este interfața dintre utilizator și kernel.
 - Bash este atât un interpretor de comenzi, cât și un limbaj de scripting.
 - Kernelul nu execută direct comenzile text introduse de utilizator.
 - Bash analizează, pregătește și coordonează execuția comenzilor înainte ca acestea să ajungă la kernel.
 - Filosofia Unix separă clar responsabilitățile: shell-ul comunică cu utilizatorul, iar kernelul administrează resursele sistemului.
-

Notă de autor

Acesta este primul capitol și aș vrea să stabilim stilul cărții chiar de acum. Prefer un ton universitar, dar accesibil, în care fiecare afirmație este construită logic peste cea anterioară. Nu vom presupune că cititorul știe deja conceptele fundamentale, însă nici nu vom simplifica excesiv. Obiectivul este ca un începător să poată înțelege, iar un administrator experimentat să găsească explicații care îi clarifică mecanisme pe care până acum doar le folosea.

Capitolul 1

1.5 De la comutatoare la shell-uri – Cum au ajuns oamenii să vorbească cu calculatoarele

„Pentru a înțelege prezentul, trebuie să înțelegem drumul care a dus până aici.”

Înainte de shell exista... aproape nimic

Astăzi deschidem un terminal, tastăm o comandă și, într-o fracțiune de secundă, calculatorul răspunde.

Pare atât de firesc încât rareori ne întrebăm: **a fost mereu așa?**

Răspunsul este **nu**.

Primele calculatoare nu aveau ecran, tastatură sau mouse. Nu exista un terminal și nici ideea de a introduce o comandă și de a primi imediat un răspuns.

Interacțiunea cu un calculator era un proces lent, costisitor și, uneori, frustrant.

Era panourilor cu comutatoare

Imaginează-ți că ești în anul 1955.

În fața ta se află un calculator care ocupă o cameră întreagă.

Nu vezi pictograme și nici un prompt care să te invite să scrii o comandă.

În schimb, vezi sute de întrerupătoare și becuri.

Fiecare comutator reprezintă un bit.

Pentru a introduce un număr în memorie, operatorul trebuia să poziționeze manual fiecare întrerupător în starea 0 sau 1.

După aceea apăsă un buton pentru a încărca valoarea în memoria calculatorului.

Apoi repeta procedura pentru următoarea instrucțiune.

Astăzi pare de neconceput.

La acea vreme era normal.

Cartelele perforate

Pe măsură ce programele au devenit mai complexe, metoda cu întrerupătoarele nu mai era practică.

A apărut o idee revoluționară.

Programul era scris pe hârtie specială, sub forma unor găuri.

Aceste cartele perforate (*punched cards*) conțineau atât instrucțiunile, cât și datele.

Operatorul introducea un teanc de cartele într-un cititor, iar calculatorul le procesa una câte una.

Dacă exista o singură greșeală pe una dintre ele, întregul program trebuia corectat și rulat din nou.

Uneori, programatorii așteptau ore sau chiar o zi întreagă doar pentru a afla că lipsea un singur caracter.

Acest mod de lucru ne amintește cât de prețios este feedbackul instantaneu pe care îl avem astăzi în terminal.

Apar terminalele

În anii '60 și începutul anilor '70, calculatoarele au început să fie conectate la terminale.

Un terminal era, în esență, un dispozitiv alcătuit dintr-o tastatură și un ecran (sau chiar o imprimantă, în primele variante). El nu executa programe și nu avea putere de calcul proprie. Rolul lui era să transmită caracterele tastate către calculator și să afișeze răspunsurile primite.

Pentru prima dată, utilizatorul putea avea o conversație directă cu sistemul.

Acesta a fost momentul în care a devenit necesar un program care să citească textul introdus și să decidă ce trebuie făcut mai departe.

Așa s-a născut ideea de **shell**.

De ce se numește „shell”?

Dacă privești structura unui sistem Unix, observi mai multe straturi.

În centru se află kernelul.

În jurul lui există un strat care îl protejează și îl face accesibil utilizatorului.

Acest strat este shell-ul.

Numele nu este doar o metaforă frumoasă.

El descrie foarte bine poziția acestui program în arhitectura sistemului.

Utilizatorul nu comunică direct cu nucleul sistemului de operare, ci cu „învelișul” care îl înconjoară.

De la un program simplu la un limbaj complet

Primele shell-uri făceau foarte puține lucruri.

Primeau o comandă.

Porneau un program.

Așteptau terminarea lui.

Apoi afișau din nou promptul.

Dar utilizatorii și-au dorit mai mult.

Au început să apară întrebări precum:

- Putem executa mai multe comenzi una după alta?
- Putem salva comenzile într-un fișier?
- Putem lua decizii în funcție de un rezultat?
- Putem repeta automat anumite operații?

Răspunsul la toate aceste întrebări a fost afirmativ.

Astfel, shell-ul a evoluat dintr-un simplu interpretor de comenzi într-un limbaj de programare.

Acesta este unul dintre motivele pentru care Bash este atât de puternic: aceeași sintaxă pe care o folosești pentru o comandă interactivă este baza unui script care poate administra sute de servere.

Caseta „În spatele cortinei”

De ce contează această istorie?

La prima vedere, poate părea doar o incursiune în trecut.

În realitate, multe dintre particularitățile Bash și ale sistemelor Unix sunt consecința directă a acestor etape istorice.

De exemplu:

- promptul există pentru că utilizatorul trebuia să știe când sistemul este pregătit să primească o nouă comandă;
- redirectionările și pipe-urile au apărut din dorința de a conecta programe simple între ele;
- scripturile au apărut pentru a elimina repetarea acelorași secvențe de comenzi.

Cu alte cuvinte, istoria explică de ce Bash arată și funcționează așa cum îl cunoaștem astăzi.

Filosofia Unix

În această carte vom reveni des la un principiu simplu, dar profund:

Un sistem bun nu devine puternic printr-un singur program uriaș, ci prin multe programe mici care colaborează eficient.

Această idee este prezentă peste tot în Unix și Linux.

Comanda `grep` caută.

`sort` sortează.

`uniq` elimină duplicatele.

`wc` numără.

Fiecare program face un singur lucru, dar îl face bine. Shell-ul este cel care le permite să colaboreze, iar din această colaborare apar soluții surprinzător de puternice.

Administrator Notes

Un administrator experimentat nu începe prin a scrie un script de sute de linii. Începe prin a rezolva problema folosind comenzi mici, pe care le testează separat. Abia după ce fiecare piesă funcționează corect le leagă într-un script.

Aceasta este una dintre cele mai importante lecții moștenite din filosofia Unix și unul dintre motivele pentru care scripturile bine scrise sunt mai ușor de înțeles, testat și întreținut.

Ce trebuie să reții

- Primele calculatoare nu aveau interfețe interactive.
 - Shell-ul a apărut pentru a permite utilizatorilor să comunice cu sistemul prin comenzi text.
 - Numele „shell” reflectă poziția sa între utilizator și kernel.
 - Bash este rezultatul unei evoluții de zeci de ani, de la un interpretor simplu la un limbaj complet de automatizare.
 - Multe caracteristici ale Bash își au originea în probleme practice pe care inginerii au încercat să le rezolve de-a lungul timpului.
 - Filosofia Unix favorizează programe mici, specializate, care colaborează prin intermediul shell-ului.
-

Notă pentru ediția finală

Aș vrea ca această secțiune să fie însoțită de o ilustrație pe două pagini, cu o cronologie vizuală:

- **1940–1950** – panouri cu comutatoare;
- **1950–1960** – cartele perforate;
- **1960–1970** – terminale text;
- **1971** – apariția primului shell Unix;
- **1977** – Bourne Shell;
- **1989** – Bash;
- **astăzi** – Bash în cloud, containere, DevOps și automatizare.

Cred că un astfel de început îi va ajuta pe cititori să înțeleagă că Bash nu este un instrument apărut întâmplător, ci rezultatul unei evoluții tehnologice și al unei filosofii de proiectare care au influențat întreaga lume Unix. În capitolele următoare vom începe să coborâm din perspectiva istorică spre mecanismele concrete ale sistemului și ale shell-ului.

Capitolul 1

1.6 Ce se întâmplă când apeși Enter?

„O comandă de câteva caractere declanșează o succesiune impresionantă de operații în sistemul de operare.”

O comandă aparent banală

Deschide un terminal și tastează:

```
ls -l /etc
```

Apoi apasă **Enter**.

Pentru tine, utilizatorul, totul pare simplu.

După o fracțiune de secundă apare lista fișierelor din directorul /etc.

Dar între momentul în care ai apăsât tasta **Enter** și momentul în care ai văzut rezultatul, sistemul de operare a executat zeci de operații.

Majoritatea utilizatorilor nu le văd niciodată.

Administratorii experimentați le intuiesc.

Inginerii de sisteme le înțeleg.

În această secțiune vom ridica cortina și vom urmări întregul proces.

Pasul 1 – Terminalul citește tastatura

Primul lucru important este să înțelegem că **Bash nu citește direct tastatura**.

Această afirmație îi surprinde pe mulți.

De fapt, tastatura este gestionată de sistemul de operare, iar aplicația pe care o vezi pe ecran este **emulatorul de terminal**.

Poate fi:

- GNOME Terminal;
- Konsole;
- xterm;
- Kitty;
- Alacritty;

- Windows Terminal;
- sau orice alt emulator compatibil.

Acesta primește fiecare caracter tastat și îl afișează pe ecran.

Când apeși **Enter**, terminalul trimite întreaga linie către Bash.

Observă diferența.

Nu trimite fiecare caracter separat pentru a fi executat.

Trimite **linia completă**.

Imaginează-ți un dialog

Poți privi terminalul ca pe un secretar.

Tu îi dictezi:

„ls -l /etc”

Secretarul nu execută nimic.

El doar notează mesajul.

Abia când îi spui:

„Trimite.”

(adică apeși **Enter**)

mesajul pleacă spre Bash.

Această analogie explică foarte bine de ce poți edita comanda înainte de execuție, folosind tastele săgeți, Backspace sau combinații precum **Ctr l+A** și **Ctr l+E**.

Toate aceste modificări au loc **înainte** ca Bash să primească linia de comandă.

Pasul 2 – Bash citește linia

Acum Bash primește textul:

```
ls -l /etc
```

Pentru Bash, aceasta este doar o succesiune de caractere.

Încă nu știe:

- unde începe comanda;
- unde se termină argumentele;

- dacă există variabile;
- dacă există ghilimele;
- dacă trebuie făcută o substituție de comandă;
- dacă există un pipe;
- dacă există redirectionări.

Cu alte cuvinte, înainte de a executa ceva, Bash trebuie să **înțeleagă ce a primit**.

Acest proces se numește **analiză sintactică** (*parsing*).

O comparație cu limbajul uman

Imaginează-ți propoziția:

„Ion vede fata cu binoclul.”

Cine are binoclul?

Ion?

Sau fata?

Fără analiză gramaticală nu putem răspunde.

Exact aceeași problemă apare și în Bash.

De exemplu:

```
echo "$HOME"
```

și

```
echo '$HOME'
```

arată aproape identic.

Dar produc rezultate complet diferite.

Diferența nu este dată de cuvântul `echo`.

Diferența este dată de modul în care Bash interpretează ghilimelele.

Vom dedica un capitol întreg acestui subiect, deoarece regulile de quoting reprezintă una dintre cele mai importante surse de erori în scripturile Bash.

Pasul 3 – Bash împarte linia în token-uri

După ce citește linia, Bash începe să o fragmenteze în elemente logice.

Aceste elemente poartă numele de **token-uri**.

Pentru comanda:

```
ls -l /etc
```

rezultatul este:

Token	Semnificație

ls	numele comenzii

-l	opțiune

/etc	argument

Acesta pare un pas banal.

Nu este.

Fără această etapă Bash nu ar putea înțelege structura comenzii.

Pasul 4 – Încep transformările

Acum începe partea fascinantă.

Linia pe care ai scris-o nu este neapărat linia care va fi executată.

Bash începe să facă transformări.

De exemplu:

```
echo $HOME
```

devine

```
echo /home/daniel
```

sau

```
echo *.txt
```

devine

```
echo raport.txt note.txt curs.txt
```

Acest proces poartă numele de **expansiune** (*expansion*).

Este una dintre cele mai puternice caracteristici ale Bash și unul dintre motivele pentru care comenzile pot părea „inteligente”.

În realitate, programele precum `echo`, `ls` sau `cp` nu știu nimic despre variabile sau caractere wildcard.

Ele primesc deja argumentele expandate de Bash.

Acest detaliu este esențial și îl vom demonstra practic în capitolele dedicate expansiunilor.

Caseta „În spatele cortinei”

Cine interpretează caracterul *?

Mulți utilizatori cred că:

```
ls *.txt
```

înseamnă că programul `ls` caută toate fișierele `.txt`.

Nu este adevărat.

În cele mai multe cazuri, `ls` nu vede niciodată caracterul `*`.

Bash îl înlocuiește înainte de lansarea programului.

Dacă în director există fișierele:

```
manual.txt  
raport.txt  
note.txt
```

programul `ls` primește efectiv:

```
ls manual.txt raport.txt note.txt
```

Acesta este unul dintre cele mai elegante exemple care arată cât de mult lucrează Bash înainte ca un program să înceapă efectiv execuția.

Administrator Notes

În depanarea scripturilor, una dintre cele mai utile întrebări este:

„Ce comandă ajunge, de fapt, să fie executată?”

Foarte multe probleme nu sunt cauzate de programul lansat, ci de faptul că Bash a modificat argumentele prin expansiuni, substituții sau împărțirea în cuvinte (*word splitting*).

De aceea, un administrator experimentat verifică mai întâi ce a produs Bash și abia apoi analizează comportamentul programului.

Ce trebuie să reții

- Emulatorul de terminal și Bash sunt două programe diferite, cu roluri diferite.
- Bash primește linia completă după apăsarea tastei **Enter**.
- Înainte de execuție, Bash analizează sintaxa și împarte linia în token-uri.
- Linia introdusă de utilizator poate fi modificată prin expansiuni înainte ca un program să fie lansat.
- Programele externe primesc, de regulă, rezultatul expansiunilor, nu textul tastat inițial.

Notă de autor

Aici aş face o pauză intenţionată. În secţiunea următoare vom continua cu paşii rămaşi: rezolvarea aliasurilor şi funcţiilor, identificarea built-in-urilor, căutarea executabilelor în PATH, crearea proceselor cu `fork()`, încărcarea programului prin `execve()`, rolul kernelului şi întoarcerea codului de ieşire către Bash.

Am ales să nu comprim totul într-o singură secţiune. Experienţa mea este că atunci când un concept complex este împărţit în etape naturale, cititorul îl înţelege şi îl reţine mult mai bine. Acest ritm ne va permite să construim, pas cu pas, o înţelegere solidă a mecanismelor interne ale Bash şi ale sistemului Linux.

Capitolul 1

1.7 Cum decide Bash ce trebuie să execute?

„Nu orice cuvânt tastat într-un terminal este un program. Prima sarcină a lui Bash este să descopere ce reprezintă acel cuvânt.”

O întrebare simplă

Să presupunem că scrii:

`ls`

Cum ştie Bash că trebuie să execute programul `ls`?

Şi mai ales...

De unde ştie unde se află?

Este în `/usr/bin`?

În `/bin`?

În `/usr/local/bin`?

Sau poate nu este deloc un program?

Aceste întrebări deschid unul dintre cele mai interesante mecanisme din Bash.

Nu tot ce tastezi este un executabil

Când introduci primul cuvânt al unei comenzi, Bash nu presupune automat că este un fișier executabil.

Înainte de a lansa un program, el verifică mai multe posibilități, într-o ordine bine stabilită.

Această ordine este foarte importantă, deoarece poate schimba complet rezultatul unei comenzi.

Pasul 1 – Este un alias?

Primul lucru pe care îl verifică Bash este existența unui **alias**.

De exemplu:

```
alias ll='ls -aLF'
```

După această definiție:

```
ll
```

nu mai este interpretat ca un nume de program.

Înainte de orice altă verificare, Bash îl înlocuiește cu:

```
ls -aLF
```

Acest proces se numește **expansiunea aliasurilor**.

Observă un detaliu important: aliasul este înlocuit **înainte** ca Bash să continue analiza comenzii.

De aceea, aliasurile sunt foarte utile pentru scurtături, dar au și limite. Ele funcționează doar în anumite contexte și nu pot înlocui flexibilitatea funcțiilor Bash.

Pasul 2 – Este o funcție?

Dacă nu există un alias, Bash verifică dacă numele respectiv aparține unei funcții definite în shell.

Exemplu:

```
backup() {  
    echo "Pornesc backup-ul..."  
}
```

Acum:

```
backup
```

nu caută niciun executabil pe disc.

Bash execută direct funcția aflată în memoria procesului său.

Acesta este unul dintre motivele pentru care funcțiile sunt foarte rapide: nu implică încărcarea unui program extern.

Pasul 3 – Este un built-in?

Dacă nu există nici alias, nici funcție, Bash verifică lista comenzilor interne (*built-ins*).

Aici găsim comenzi precum:

- `cd`
- `pwd`
- `export`
- `readonly`
- `alias`
- `history`
- `jobs`
- `wait`
- `type`
- `set`
- `shopt`

Aceste comenzi nu există ca fișiere executabile pe disc. Ele fac parte din codul sursă al lui Bash și sunt încărcate în memorie odată cu shell-ul.

Acesta este motivul pentru care nu vei găsi, de exemplu, un fișier executabil pentru `cd` în `/usr/bin`.

De ce `cd` trebuie să fie un built-in?

Aceasta este una dintre cele mai frumoase întrebări pe care le putem pune.

Imaginează-ți că `cd` ar fi un program extern.

Bash ar crea un proces nou pentru acel program.

Programul ar schimba directorul curent al propriului proces.

Apoi programul s-ar închide.

Ce s-ar întâmpla cu Bash?

Absolut nimic.

El ar rămâne în același director în care se afla înainte.

Motivul este simplu: fiecare proces are propriul său director de lucru curent. Un proces nu poate schimba directorul de lucru al altui proces.

De aceea, `cd` trebuie executată în procesul Bash însuși. Numai astfel modificarea persistă după terminarea comenzii.

Această explicație este valabilă și pentru alte built-in-uri care modifică starea shell-ului, cum ar fi `export`, `unset`, `readonly` sau `umask`.

Pasul 4 – Este un program extern?

Abia acum Bash începe să caute un executabil pe disc.

Dar unde?

Răspunsul este: în directoarele enumerate în variabila de mediu `PATH`.

De exemplu:

```
echo "$PATH"
```

ar putea afișa:

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

Fiecare director este verificat, de la stânga la dreapta.

Dacă primul director nu conține executabilul căutat, Bash trece la următorul și continuă până când găsește o potrivire sau epuizează lista.

Ordinea directoarelor este esențială. Dacă două directoare conțin un executabil cu același nume, va fi lansat primul găsit în `PATH`.

Cum poți vedea ce execută Bash?

Comanda:

```
type ls
```

este una dintre cele mai utile pentru un administrator.

Ea poate răspunde:

```
ls is /usr/bin/ls
```

sau:

```
cd is a shell builtin
```

sau:

```
ll is aliased to 'ls -aF'
```

sau chiar:

backup is a function

Observă cât de multe informații oferă o singură comandă. `type` este un instrument excelent pentru a înțelege deciziile luate de Bash.

Caseta „În spatele cortinei”

Cum evită Bash să caute mereu în PATH?

Dacă la fiecare comandă Bash ar verifica toate directoarele din PATH, ar pierde timp inutil.

Pentru a accelera execuția, Bash folosește un **tabel de hash** (*hash table*). Prima dată când găsește un executabil, reține calea completă în memorie.

Astfel, la următoarea execuție a aceleiași comenzi, nu mai este nevoie să parcurgă din nou toate directoarele.

Poți vedea conținutul acestui tabel cu:

```
hash
```

iar dacă ai mutat sau înlocuit un executabil și vrei ca Bash să uite căile memorate, poți folosi:

```
hash -r
```

Acest detaliu explică de ce uneori, după instalarea sau mutarea unui program, Bash pare să „țină minte” vechea locație.

Administrator Notes

În depanare, una dintre cele mai frecvente greșeli este presupunerea că se execută programul „corect”.

Înainte de a investiga un comportament ciudat, verifică întotdeauna:

```
type nume_comandă  
which nume_comandă  
command -V nume_comandă
```

Dacă rezultatele diferă de ceea ce te aștepți, este foarte posibil ca problema să nu fie programul, ci faptul că Bash execută un alias, o funcție sau un alt executabil aflat mai devreme în PATH.

Ce trebuie să reții

- Primul cuvânt al unei comenzi nu este întotdeauna un executabil.

- Bash verifică, într-o ordine bine definită: aliasuri, funcții, built-in-uri și abia apoi programe externe.
 - Built-in-urile există pentru că anumite operații trebuie să modifice starea shell-ului curent.
 - Variabila PATH stabilește unde caută Bash programele externe.
 - Bash utilizează un tabel de hash pentru a accelera localizarea executabilelor.
 - Comanda `type` este unul dintre cele mai utile instrumente pentru a înțelege ce execută Bash cu adevărat.
-

Notă de autor

Acesta este unul dintre capitolele pe care aș vrea să le țină minte orice cititor. De multe ori, probleme aparent misterioase – un alias care schimbă comportamentul unei comenzi, un executabil lansat din alt director sau o funcție care maschează un program extern – se explică imediat dacă înțelegi ordinea în care Bash rezolvă numele comenzilor.

În secțiunea următoare vom coborî încă un nivel: vom vedea cum Bash creează procese, ce înseamnă `fork()` și `execve()`, de ce fiecare comandă rulează într-un proces separat și de ce această arhitectură este una dintre marile idei care au făcut Unix atât de robust și de flexibil. Acela va fi, probabil, primul contact al cititorului cu mecanismele interne ale sistemului de operare și va pune bazele înțelegerii întregului ecosistem Linux.

Capitolul 1

1.8 De la comandă la proces – Cum prinde viață un program

„Un program aflat pe disc este doar o colecție de biți. Devine viu abia atunci când sistemul de operare îi oferă un proces.”

Ce înseamnă, de fapt, să rulezi un program?

Spunem foarte des:

„Am rulat `ls`.”

„Am pornit Apache.”

„Am executat un script Bash.”

Aceste expresii sunt corecte în limbajul de zi cu zi, dar ascund o realitate tehnică mult mai interesantă.

Pe disc, un program este doar un fișier.

Dacă deschizi executabilul `/usr/bin/ls` cu un editor hexazecimal, vei vedea doar o succesiune de octeți. Procesorul nu îl execută direct de pe disc. El trebuie mai întâi încărcat în memorie, iar sistemul de operare trebuie să creeze un context de execuție.

Acest context se numește **proces**.

Fără proces, niciun program nu poate rula.

Program sau proces?

Acești doi termeni sunt adesea folosiți ca sinonime, dar în informatică înseamnă lucruri diferite.

Un **program** este un fișier. El conține instrucțiuni și date stocate pe un dispozitiv de stocare.

Un **proces** este o instanță aflată în execuție a acelui program.

Imaginează-ți o rețetă de prăjitură.

Rețeta tipărită într-o carte este programul.

Prăjitura pe care o pregătești în bucătărie este procesul.

Poți folosi aceeași rețetă de mai multe ori și poți avea mai multe prăjituri în același timp. La fel, același executabil poate avea mai multe procese active simultan.

Acesta este motivul pentru care poți deschide mai multe ferestre de terminal sau mai multe instanțe ale aceluiași editor de text. Fiecare este un proces separat, chiar dacă toate folosesc același program de pe disc.

Cine creează procesul?

Mulți presupun că procesorul sau kernelul „pornește programul”.

Realitatea este puțin diferită.

Inițiativa aparține lui Bash.

După ce a analizat comanda și a găsit executabilul, Bash cere kernelului să creeze un proces nou.

În Linux, această cerere se face prin apeluri de sistem. Din perspectiva utilizatorului, totul pare instantaneu. Din perspectiva sistemului, însă, începe o secvență bine definită de pași.

Primul pas: `fork()`

În lumea Unix există o idee elegantă:

un proces nou este creat pornind de la unul existent.

Bash nu construiește un proces de la zero.

El spune kernelului, în esență:

„Fă o copie a mea.”

Această operație este realizată prin apelul de sistem `fork()`.

Rezultatul este existența a două procese aproape identice:

- procesul părinte (Bash);
- procesul copil.

Ambele continuă execuția din același punct al programului, dar au identificatori de proces (PID-uri) diferiți și spații de memorie separate.

Este una dintre cele mai elegante idei introduse de Unix și reprezintă fundamentul modelului de execuție folosit și astăzi în Linux.

Dar de ce să copiezi Bash?

Întrebarea este firească.

Dacă scopul este să rulezi `ls`, de ce mai creezi o copie a lui Bash?

Pentru că Bash trebuie să rămână disponibil după terminarea comenzii.

Dacă Bash s-ar transforma direct în `ls`, atunci, după ce `ls` s-ar închide, terminalul nu ar mai avea cine să afișeze promptul și să primească următoarea comandă.

Prin urmare, Bash își creează un copil. Copilul va executa comanda, iar părintele va rămâne în așteptare.

Această separare între părinte și copil este unul dintre motivele pentru care shell-ul poate lansa succesiv mii de comenzi fără să-și piardă propria stare.

Al doilea pas: `execve()`

După `fork()`, procesul copil este încă o copie a lui Bash.

El nu a devenit încă `ls`.

Urmează apelul de sistem `execve()`.

Acesta înlocuiește imaginea procesului copil din memorie cu imaginea programului care trebuie executat.

Imaginează-ți că actorul de pe scenă își schimbă complet costumul, decorul și rolul, dar rămâne aceeași persoană care ocupă scena.

La fel, procesul își păstrează identitatea de bază (PID-ul), dar codul și datele sale sunt înlocuite cu cele ale noului program.

Din acel moment, copilul nu mai execută cod Bash.

Execută codul lui `ls`.

O analogie din viața de zi cu zi

Imaginează-ți un atelier de tâmplărie.

Meșterul principal (Bash) primește o comandă:

„Construiește un scaun.”

El nu părăsește atelierul.

În schimb, desemnează un ucenic (procesul copil), îi oferă planurile și materialele (programul și argumentele), apoi îl lasă să lucreze.

În timp ce ucenicul construiește scaunul, meșterul rămâne disponibil pentru următoarea comandă.

Când ucenicul termină, se întoarce și raportează rezultatul.

Acesta este modelul părinte–copil folosit de Unix de peste cinci decenii.

Caseta „În spatele cortinei”

Copiază Linux chiar toată memoria la `fork()`?

Dacă ar face acest lucru, fiecare comandă ar consuma foarte mult timp și foarte multă memorie.

Linux folosește o tehnică numită **Copy-on-Write (COW)**.

La început, procesul părinte și copil împart aceleași pagini de memorie. Acestea sunt marcate ca fiind doar pentru citire.

Doar în momentul în care unul dintre procese încearcă să modifice o pagină, kernelul creează o copie separată pentru acel proces.

Acest mecanism face ca `fork()` să fie surprinzător de eficient și explică de ce modelul părinte–copil poate fi folosit intensiv fără penalizări majore de performanță.

Acest subiect va reveni atunci când vom discuta despre memorie virtuală și managementul proceselor.

Administrator Notes

Dacă ai rulat vreodată:

```
ps -ef
```

și ai observat relații între procese, acum începi să înțelegi de ce există.

Fiecare proces are un părinte. Această relație formează un arbore de procese care pornește, în cele din urmă, de la procesul cu PID 1 (în sistemele moderne, de regulă `systemd`).

În capitolele dedicate administrării proceselor vom explora acest arbore și vom vedea cum ne ajută să înțelegem comportamentul sistemului și să diagnosticăm probleme.

Ce trebuie să reții

- Un program și un proces nu sunt același lucru.
 - Programul este un fișier; procesul este o instanță aflată în execuție.
 - Bash cere kernelului să creeze un proces copil prin `fork()`.
 - Procesul copil este transformat în programul dorit prin `execve()`.
 - Bash rămâne activ pentru a putea primi și executa următoarele comenzi.
 - Modelul părinte–copil este unul dintre conceptele fundamentale ale Unix și Linux.
 - Tehnica **Copy-on-Write** face ca `fork()` să fie mult mai eficient decât pare la prima vedere.
-

Notă de autor

Aici aş încheia cu o idee care, după părerea mea, merită să rămână în mintea cititorului:

Bash nu execută programe. El creează condițiile pentru ca acestea să poată fi executate.

Este o diferență subtilă, dar profundă. Când cineva înțelege această idee, începe să privească Linux ca pe un sistem de procese care colaborează, nu ca pe o simplă colecție de comenzi.

Capitolul 1

1.9 Cum știe kernelul ce program să încarce?

„Între comanda tastată și programul executat există o etapă invizibilă: încărcarea executabilului în memorie. Aici începe cu adevărat colaborarea dintre Bash și kernel.”

Dintr-un fișier pe disc într-un program aflat în execuție

În secțiunea anterioară am văzut că Bash creează un proces copil prin `fork()`, iar apoi apelează `execve()` pentru a transforma acel proces în programul dorit.

Dar apare o întrebare firească.

Cum reușește kernelul să transforme un simplu fișier de pe disc într-un program care rulează în memorie?

La prima vedere pare magie.

În realitate, este un proces riguros, construit în mai multe etape.

Un executabil este doar un fișier

Privește fișierul:

```
/usr/bin/ls
```

Pentru noi este „programul `ls`”.

Pentru sistemul de fișiere este doar un fișier oarecare.

Poți să-i vezi dimensiunea:

```
ls -lh /usr/bin/ls
```

Poți să-i schimbi proprietarul.

Poți să-i modifici permisiunile.

Poți chiar să-l copiezi.

Nimic din toate acestea nu îl face să ruleze.

Un fișier devine program doar atunci când kernelul decide să îl încarce în memorie.

Ce vede kernelul?

Când Bash apelează `execve()`, nu îi spune kernelului:

„Rulează `ls`.”

Îi spune ceva mult mai precis.

În esență:

„Deschide acest fișier, verifică dacă este executabil și pregătește-l pentru execuție.”

Din acel moment, Bash nu mai controlează procesul de încărcare.

Kernelul preia responsabilitatea.

Prima verificare: există fișierul?

Kernelul începe cu cea mai simplă întrebare.

Există fișierul?

Dacă nu există, `execve()` eșuează, iar Bash afișează un mesaj de forma:

```
bash: program: command not found
```

Este important să observăm că acest mesaj este afișat de Bash, nu de kernel.

Kernelul doar întoarce un cod de eroare.

Bash îl transformă într-un mesaj pe care utilizatorul îl poate înțelege.

A doua verificare: ai voie să îl execuți?

Existența fișierului nu este suficientă.

Kernelul verifică și permisiunile.

Are utilizatorul drept de execuție?

Există bitul x?

Dacă nu, rezultatul va fi ceva asemănător cu:

```
Permission denied
```

Din nou, kernelul întoarce o eroare, iar Bash o prezintă într-o formă prietenoasă.

A treia verificare: este un executabil valid?

Acesta este unul dintre cele mai interesante momente.

Kernelul nu execută orice fișier.

El trebuie mai întâi să înțeleagă **ce tip de fișier este**.

Poate fi:

- un executabil ELF;
- un script care începe cu `#!`;
- un binar pentru altă arhitectură;
- sau un fișier care nu poate fi executat.

Această verificare este făcută automat.

ELF – Limba executabilelor Linux

Majoritatea programelor compilate pentru Linux folosesc formatul **ELF** (*Executable and Linkable Format*).

Acesta descrie:

- unde se află codul programului;
- unde se află datele;
- ce biblioteci trebuie încărcate;
- unde începe execuția;
- informații despre depanare;
- informații despre legare.

Cu alte cuvinte, ELF este harta după care kernelul construiește procesul.

Poți verifica tipul unui executabil cu:

```
file /usr/bin/ls
```

Rezultatul va fi asemănător cu:

```
ELF 64-bit LSB executable...
```

Această comandă este utilă și pentru identificarea scripturilor, bibliotecilor și altor tipuri de fișiere.

Dar scripturile Bash?

Dacă rulezi:

```
./backup.sh
```

kernelul observă că fișierul nu este un executabil ELF.

În schimb, citește primele două caractere:

```
#!
```

Aceasta este celebra linie **shebang**.

De exemplu:

```
#!/bin/bash
```

sau:

```
#!/usr/bin/env python3
```

Kernelul înțelege că nu trebuie să execute direct fișierul, ci să pornească mai întâi interpretul indicat după `#!` și să îi ofere scriptul ca argument.

Acesta este motivul pentru care un script Bash nu este executat direct de kernel, ci prin intermediul interpretului Bash.

O analogie

Imaginează-ți că primești un plic.

Înainte să citești conținutul, verifici adresa și destinatarul.

Dacă este adresat altcuiva, nu îl deschizi.

Dacă este în limba pe care o cunoști, îl citești.

Dacă este într-o limbă necunoscută, cauți un traducător.

Kernelul procedează asemănător.

ELF înseamnă:

„Știu cum să citesc acest fișier.”

Shebang înseamnă:

„Am nevoie de un interpret.”

Iar un fișier obișnuit fără niciuna dintre aceste caracteristici nu poate fi executat.

Caseta „În spatele cortinei”

Cine execută, de fapt, scriptul?

Mulți spun:

„Kernelul execută scriptul Bash.”

În realitate, lucrurile sunt mai subtile.

Kernelul execută **programul Bash**.

Acesta este cel care citește, interpretează și execută fiecare linie din script.

Această diferență explică de ce același script poate fi rulat și cu:

```
bash script.sh
```

sau chiar:

```
sh script.sh
```

în funcție de interpretul folosit.

Scriptul nu este cod mașină. Este text care trebuie interpretat.

Administrator Notes

Când un script refuză să ruleze, nu presupune imediat că problema este în cod.

Verifică, în această ordine:

1. Există fișierul?
2. Are permisiune de execuție?
3. Linia `#!` indică un interpret valid?
4. Fișierul are finaluri de linie specifice Windows (CRLF) în loc de LF?
5. Interpretul există în locația specificată?

Foarte multe probleme aparent complicate se rezolvă în primele două minute dacă urmezi această ordine.

Ce trebuie să reții

- Un executabil este doar un fișier până când kernelul îl încarcă în memorie.
- `execve()` transferă responsabilitatea către kernel.
- Kernelul verifică existența fișierului, permisiunile și formatul executabilului.
- ELF este formatul standard al executabilelor Linux.
- Scripturile sunt executate prin intermediul unui interpret indicat de linia `#!`.
- Kernelul execută interpretul; interpretul execută scriptul.

Capitolul 1

1.10 Viața unui proces

„Orice proces are o poveste: se naște, trăiește, comunică, creează alți copii și, într-un final, moare. Linux nu vede programe. Linux vede procese.”

Tot ceea ce rulează este un proces

Privește în jurul tău.

Editorul de text în care lucrezi.

Browserul.

Serverul web.

Playerul audio.

Terminalul.

Bash.

Toate au un lucru în comun.

Pentru Linux, toate sunt procese.

Kernelul nu știe că unul este un browser și altul un editor.

El știe doar că există procese care concurează pentru aceleași resurse:

- timpul procesorului;
- memoria RAM;
- accesul la disc;
- accesul la rețea;
- dispozitivele hardware.

Din punctul de vedere al kernelului, toate sunt egale.

Aceasta este una dintre cele mai frumoase idei din Unix.

Nașterea

Un proces apare atunci când alt proces execută `fork()`.

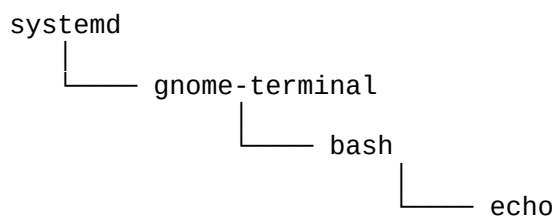
Asta înseamnă că, în Linux, aproape orice proces are un părinte.

Acest lucru nu este întâmplător.

Dacă deschizi un terminal și rulezi:

```
echo "Salut"
```

sucesiunea este aproximativ aceasta:



Observă ceva interesant.

`echo` nu apare din senin.

Fiecare proces are un istoric.

Există întotdeauna cineva care l-a creat.

Acest lanț de relații este ceea ce numim **arborele de procese** (*process tree*).

Arborele vieții

În biologie există arbori genealogici.

În Linux există arbori de procese.

Rulează:

`pstree`

Prima dată când cineva vede rezultatul pare haotic.

În realitate, privești genealogia întregului sistem.

Vei vedea:

```
systemd
├── NetworkManager
├── sshd
├── cron
├── nginx
├── ...
└── gnome-terminal
    ├── bash
    │   └── vim
```

Acesta este unul dintre cele mai utile instrumente pentru depanare.

Vom reveni asupra lui de multe ori.

Identitatea

La naștere, fiecare proces primește ceva unic.

Un **PID**.

(Process Identifier)

Niciun alt proces activ nu poate avea același PID.

Este echivalentul unui CNP pentru procese.

Dacă un proces moare, PID-ul său poate fi reutilizat ulterior.

Dar niciodată simultan.

Procesul își construiește propria lume

Acesta este un aspect extraordinar de important.

Fiecare proces are propria sa lume.

Are:

- propriile variabile;
- propriul director curent;
- propria memorie;
- proprii descriptori de fișiere;
- propriile permisiuni;
- propriile semnale;
- propriul cod de execuție.

Acesta este motivul pentru care două ferestre Bash pot avea directoare diferite.

Una poate fi în:

/etc

iar cealaltă în:

/home/daniel

Fiecare proces își păstrează propria stare.

De ce nu poate un proces să modifice alt proces?

Aceasta este una dintre cele mai inteligente decizii din Unix.

Imaginează-ți haosul dacă orice aplicație ar putea modifica memoria altui program.

Un browser ar putea rescrie memoria serverului SSH.

Un editor ar putea modifica memoria bazei de date.

Sistemul ar deveni inutilizabil.

Kernelul izolează procesele.

Fiecare trăiește în propriul spațiu de adrese.

Singurul care poate vedea imaginea completă este kernelul.

Comunicarea

Dar atunci apare o întrebare.

Dacă procesele sunt izolate...

cum comunică între ele?

Răspunsul este fascinant.

Kernelul oferă mecanisme speciale.

Printre ele:

- pipe-uri;
- socket-uri;
- memorie partajată;
- semnale;
- mesaje IPC.

Vom dedica aproape un întreg volum acestor mecanisme.

Pentru moment este suficient să înțelegem că procesele nu își scriu direct unul în memoria celuilalt.

Kernelul este intermediarul.

Caseta „În spatele cortinei”

De ce Bash nu poate schimba variabilele shell-ului părinte?

Acum putem explica o întrebare care apare frecvent.

De ce:

```
./script.sh
```

nu modifică variabilele shell-ului după ce se termină?

Pentru că scriptul rulează într-un alt proces.

La terminarea procesului, întreaga sa memorie dispare.

Variabilele dispar împreună cu el.

În schimb:

```
source script.sh
```

execută scriptul în procesul Bash curent.

Nu există proces copil.

De aceea modificările rămân active.

Aceasta este una dintre cele mai importante diferențe din Bash și una dintre cele mai frecvente întrebări la LPIC.

Administrator Notes

Când un administrator spune:

„Problema este procesul.”

de cele mai multe ori nu se referă la program.

Se referă la starea acelui proces:

- consumă prea multă memorie;
- este blocat;
- a devenit zombie;
- nu răspunde;
- a pierdut conexiunea;
- a rămas fără descriptori de fișiere.

Aceasta este o diferență subtilă, dar esențială.

Ce trebuie să reții

- Linux administrează procese, nu „aplicații”.
- Aproape fiecare proces are un proces părinte.
- Procesele formează un arbore.
- Fiecare proces are propriul PID și propria stare.
- Procesele sunt izolate între ele.
- Comunicarea dintre procese este realizată prin mecanisme oferite de kernel.
- `source` rulează în procesul curent; `./script.sh` creează un proces separat.

Capitolul 1

1.11 Cine conduce calculatorul?

„Mulți utilizatori cred că Bash conduce sistemul de operare. În realitate, Bash este doar unul dintre mulții participanți la o conversație mult mai complexă.”

O întrebare aparent simplă

Imaginează-ți următoarea situație.

Tastezi:

```
rm raport.txt
```

Fișierul dispare.

Întrebarea este:

Cine l-a șters?

Majoritatea oamenilor răspund imediat:

Bash.

Pare logic.

Ai tastat comanda în Bash.

Dar răspunsul este...

nu.

Bash nu șterge niciun fișier

Acesta este unul dintre cele mai importante lucruri pe care le vom înțelege în această carte.

Bash aproape că nu face niciodată operația pe care crezi că o face.

El nu șterge fișiere.

Nu copiază fișiere.

Nu sortează.

Nu comprimă arhive.

Nu trimite pachete în rețea.

Nu listează directoare.

Atunci...

ce face?

Bash este un coordonator

Imaginează-ți un șantier.

Există:

- arhitectul;
- dirigintele de șantier;
- zidarii;
- electricienii;
- instalatorii;
- șoferii de utilaje.

Dirigintele nu pune cărămizi.

Dar fără el șantierul ar deveni haotic.

Exact acesta este rolul lui Bash.

El coordonează.

Cine execută, atunci, rm?

Programul:

`/usr/bin/rm`

Acesta este executabilul.

El este cel care citește argumentele.

El verifică opțiunile.

El cere kernelului să elimine intrările din sistemul de fișiere.

Bash doar îl pornește.

Această idee pare simplă, dar schimbă complet perspectiva asupra Linux.

Atunci cine șterge efectiv fișierul?

Nici `rm`.

Da...

ai citit bine.

Programul `rm` nu poate șterge nimic de unul singur.

El nu are acces direct la disc.

Nu poate modifica sistemul de fișiere.

Nu poate muta blocuri pe SSD.

Tot ce poate face este să spună kernelului:

„Te rog să ștergi această intrare din sistemul de fișiere.”

Kernelul verifică:

- permisiunile;
- proprietarul;
- sistemul de fișiere;
- dacă fișierul este deschis;
- dacă partiția este montată doar pentru citire;

- dacă există alte restricții.

Abia apoi decide dacă operația este permisă.

Observă schimbarea de perspectivă.

Nu aplicațiile controlează calculatorul.

Kernelul controlează calculatorul.

Aplicațiile doar solicită servicii.

Linux este un stat

Imaginează-ți un stat modern.

Cetățenii nu construiesc singuri drumurile.

Nu emit pașapoarte.

Nu înregistrează proprietăți.

Ei depun cereri.

Instituțiile statului verifică și aprobă sau resping solicitările.

Kernelul joacă un rol asemănător.

Programele sunt cetățenii.

Ele nu fac operații direct asupra hardware-ului.

Ele solicită kernelului să le facă.

Kernelul decide.

De ce există această separare?

Imaginează-ți ce s-ar întâmpla dacă orice program ar putea scrie direct pe SSD.

Sau în memoria altui proces.

Sau pe placa de rețea.

Sistemul ar deveni imposibil de controlat.

Orice eroare ar putea compromite întregul calculator.

Unix a ales o arhitectură elegantă.

Există o singură autoritate care controlează resursele.

Aceasta este kernelul.

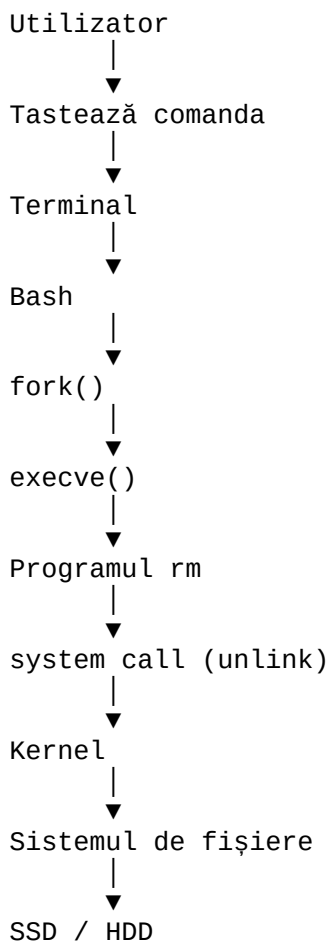
Toți ceilalți trebuie să îi ceară permisiunea.

O comandă simplă, o conversație complexă

Să urmărim din nou:

```
rm raport.txt
```

În spate se întâmplă aproximativ următoarele:



Această diagramă este extraordinar de importantă.

Ea arată că între utilizator și hardware există mai multe niveluri.

Fiecare are propriul rol.

Un lucru pe care mulți nu îl observă

Dacă rulezi:

```
cp mare.iso backup.iso
```

și apoi:

```
mv backup.iso arhiva.iso
```

cele două comenzi par asemănătoare.

În realitate sunt operații complet diferite.

Prima presupune copierea datelor.

A doua, în multe cazuri, înseamnă doar modificarea unei intrări din sistemul de fișiere.

Vom descoperi mai târziu că uneori mv nu mută aproape nimic.

Schimbă doar metadatele.

Acesta este unul dintre motivele pentru care mutarea unui fișier mare pe aceeași partiție este aproape instantanee.

Caseta „În spatele cortinei”

Ce este un apel de sistem?

De mai multe ori am folosit expresia „programul cere kernelului”.

Dar cum îi cere?

Prin **apeluri de sistem** (*system calls*).

Acestea reprezintă punctele oficiale prin care un program poate solicita servicii kernelului.

Printre cele mai cunoscute se numără:

- `open()`
- `read()`
- `write()`
- `close()`
- `fork()`
- `execve()`
- `unlink()`
- `stat()`
- `mkdir()`

Le vom întâlni constant pe parcursul cărții.

Nu trebuie memorate acum.

Este suficient să înțelegem că ele reprezintă limbajul prin care aplicațiile comunică cu kernelul.

Administrator Notes

Una dintre cele mai mari greșeli pe care le fac începătorii este să creadă că o comandă „face” ceva.

În realitate, aproape toate comenzile doar orchestrează apeluri către kernel.

Această perspectivă schimbă complet modul în care depanezi un sistem.

Nu mai întrebi:

De ce nu merge `cp`?

Întrebi:

Ce solicitare către kernel a eșuat și de ce?

Este o întrebare mult mai aproape de cauza reală.

Ce trebuie să reții

- Bash nu execută operațiile asupra hardware-ului.
- Programele externe nu controlează direct resursele sistemului.
- Kernelul este singura componentă care poate administra hardware-ul și resursele critice.
- Aplicațiile comunică cu kernelul prin apeluri de sistem.
- Fiecare comandă executată într-un terminal reprezintă, de fapt, o conversație între mai multe componente ale sistemului.

Capitolul 1

1.12 Marea conversație

„Un calculator nu execută comenzi. Un calculator poartă conversații.”

Privește din nou o comandă banală

`ls -l /etc`

Acum câteva ore părea doar o comandă.

Acum știm că nu este.

În spatele ei există o întreagă conversație.

Hai să o ascultăm.

Utilizatorul

Vreau să văd conținutul directorului /etc.

Terminalul

Am înregistrat ce ai tastat.

O trimit către Bash.

Bash

Am primit textul.

Îl analizez.

Îl împart în token-uri.

Verific aliasurile.

Verific funcțiile.

Verific built-in-urile.

Caut executabilul.

L-am găsit.

Kernel, am nevoie de un proces nou.

Kernelul

Creez procesul.

Copiez contextul shell-ului.

Încarc executabilul ls.

Procesul este pregătit.

Programul ls

Mulțumesc.

Acum am nevoie să citesc directorul /etc.

Kernelul

Ai dreptul?

Programul ls

Da.

Kernelul

Atunci îți citesc conținutul.

Sistemul de fișiere

Acestea sunt fișierele.

Programul ls

Le formatez frumos.

Terminalul

Le afișez utilizatorului.

Bash

Comanda s-a terminat.

Codul de ieșire este 0.

Pot primi următoarea comandă.

Ai observat ceva?

Bash nu a citit directorul.

ls nu a citit direct discul.

Terminalul nu a executat nimic.

Kernelul nu a afișat nimic.

Fiecare componentă și-a făcut doar partea ei.

Exact aceasta este frumusețea Unix.

Filosofia responsabilităților mici

Există o regulă nescrisă care apare peste tot în Unix.

Fiecare componentă trebuie să aibă o responsabilitate clară și limitată.

Terminalul afișează.

Bash interpretează.

Kernelul administrează.

ls listează.

sort sortează.

grep caută.

wc numără.

Această separare face sistemul ușor de înțeles și de extins.

Dacă fiecare program ar încerca să facă totul, Linux ar deveni imposibil de întreținut.

De ce este atât de important acest model?

Pentru că îl vei întâlni peste tot.

Când vei învăța Docker.

Când vei învăța Kubernetes.

Când vei învăța Git.

Când vei învăța systemd.

Când vei învăța baze de date.

Toate aceste sisteme sunt construite pe aceeași idee:

componente mici, responsabilități clare și colaborare prin interfețe bine definite.

Linux nu este doar un sistem de operare.

Este o lecție de arhitectură software.

Caseta „În spatele cortinei”

De ce Linux rezistă de peste 30 de ani?

Tehnologia se schimbă.

Procesoarele se schimbă.

SSD-urile au înlocuit hard diskurile.

Containerele au apărut.

Cloud-ul a schimbat infrastructura.

Dar filosofia Unix a rămas.

Nu pentru că este veche.

Ci pentru că este solidă.

O idee bună îmbătrânește greu.

Administrator Notes

În producție, rareori rezolvi o problemă memorând o comandă.

O rezolvi punând întrebările potrivite:

- Cine trebuia să facă această operație?
- Cine nu și-a îndeplinit rolul?
- În ce etapă s-a întrerupt conversația?
- Cine a returnat eroarea?
- Ce componentă trebuie investigată?

Acest mod de gândire reduce enorm timpul de diagnosticare.

Oprește-te un minut...

Aș vrea să faci un exercițiu mental.

Data viitoare când tastezi o comandă, nu te gândești la rezultat.

Gândește-te la conversația care începe în acel moment.

Imaginează-ți fiecare participant.

Terminalul.

Bash.

Kernelul.

Programul.

Sistemul de fișiere.

Din acel moment, Linux nu va mai părea o cutie neagră.

Va deveni un sistem logic.

Ce trebuie să reții

- O comandă este începutul unei colaborări între mai multe componente ale sistemului.
- Fiecare componentă are o responsabilitate bine definită.
- Bash coordonează, kernelul administrează, programele rezolvă sarcini specifice.
- Înțelegerea responsabilităților este cheia depanării și a administrării eficiente.
- Filosofia Unix se bazează pe componente mici care cooperează prin interfețe clare.

Capitolul 1

1.13 Cum gândește un inginer de sisteme?

„Diferența dintre un utilizator și un inginer nu este cât de multe comenzi știe, ci întrebările pe care și le pune.”

Aceeași problemă, două moduri de a gândi

Imaginează-ți că cineva spune:

„Nu merge ping.”

Această propoziție este foarte des întâlnită.

Dar ea nu descrie problema.

Describe doar simptomul.

Un utilizator obișnuit începe imediat să încerce soluții:

- repornește calculatorul;

- schimbă cablul;
- dezactivează firewall-ul;
- reinstalează pachetul;
- caută pe internet aceeași eroare.

Uneori funcționează.

Alteori nu.

Cum gândește un inginer?

Un inginer nu începe cu soluțiile.

Începe cu întrebările.

Prima întrebare nu este:

Cum repar?

Prima întrebare este:

Ce se întâmplă, de fapt?

Pare o diferență mică.

În realitate este enormă.

Modelul mental

De fiecare dată când apare o problemă, un inginer încearcă să construiască în mintea lui un model al sistemului.

Nu caută imediat răspunsul.

Încearcă să înțeleagă traseul.

De exemplu:

ping openai.com

Nu înseamnă doar:

trimit un pachet.

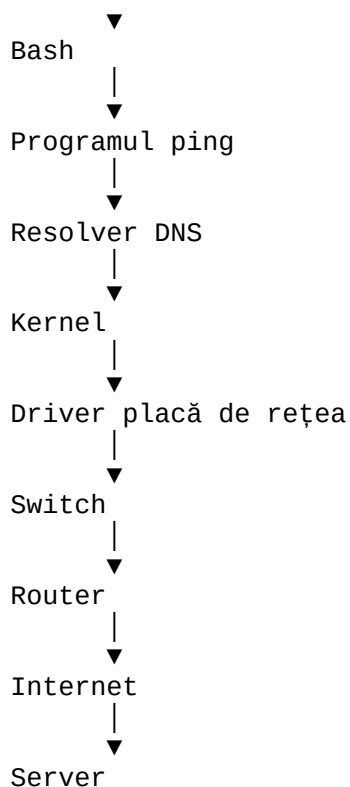
Înseamnă:

Utilizator



Terminal





Acum apare o întrebare.

Unde s-a oprit conversația?

Linux este o succesiune de responsabilități

Acesta este unul dintre cele mai importante lucruri pe care vrem să le transmitem în această carte.

Foarte puține probleme sunt "magice".

Cele mai multe sunt doar o responsabilitate care nu și-a îndeplinit rolul.

De exemplu:

Nu merge browserul.

Întrebarea nu este:

De ce nu merge browserul?

Ci:

- Browserul nu pornește?
- Pornește, dar nu rezolvă DNS?
- DNS funcționează, dar nu există rută?
- Există rută, dar firewall-ul blochează?
- Firewall-ul permite, dar serverul nu răspunde?

Observi diferența?

Problema începe să se descompună.

Arta descompunerii

Una dintre cele mai valoroase abilități ale unui administrator Linux este capacitatea de a împărți o problemă mare în probleme mici.

Aceasta este o idee care apare peste tot în inginerie.

Nu repari sistemul.

Repari o componentă.

Nu înțelegi Linux dintr-o dată.

Înțelegi procesele.

Apoi memoria.

Apoi sistemul de fișiere.

Apoi rețeaua.

Apoi modul în care colaborează.

La final apare imaginea de ansamblu.

O întrebare pe care mi-aș dori să și-o pună fiecare cursant

În loc de:

Ce comandă rezolvă asta?

Aș vrea să întreb:

Cine este responsabil pentru această operație?

Este o întrebare extraordinară.

Dacă înveți să o pui, ai făcut un pas important spre gândirea inginerească.

Filosofia Unix revine

Observă că această idee este aceeași pe care am întâlnit-o deja.

Responsabilități mici.

Componente independente.

Interfețe clare.

Linux este construit astfel.

Și cred că și mintea unui administrator ar trebui să funcționeze la fel.

Caseta „În spatele cortinei”

De ce administratorii foarte buni par atât de rapizi?

Mulți oameni cred că administratorii experimentați memorează mii de comenzi.

Nu.

Ei memorează modele.

Când apare o problemă nouă, ei nu pornesc de la zero.

O compară cu modele pe care le-au întâlnit înainte.

Este același motiv pentru care un medic cu experiență recunoaște rapid anumite simptome. Nu pentru că știe pe de rost toate bolile, ci pentru că a construit în timp modele mentale care îi permit să lege rapid indiciile între ele.

Acesta este și motivul pentru care, pe parcursul cărții, vom reveni constant la aceleași concepte fundamentale. Nu pentru a le repeta, ci pentru a le consolida și a le transforma în instrumente de gândire.

Administrator Notes

Începătorii caută comenzi.

Administratorii caută cauze.

Arhitecții caută modele.

Aceasta este evoluția firească în lumea tehnică.

Scopul nostru este ca această carte să îl ajute pe cititor să parcurgă acest drum.

Ce trebuie să reții

- O problemă nu este același lucru cu simptomul ei.
- Înainte de soluții, încearcă să înțelegi mecanismul.
- Problemele complexe se rezolvă prin descompunere în componente mai mici.
- Întrebarea „Cine este responsabil pentru această operație?” este una dintre cele mai utile în administrarea Linux.

- Modelele mentale sunt mai valoroase decât memorarea comenzilor.

Capitolul 1

1.14 Iluzia simultaneității

„Calculatorul pare că face zeci de lucruri în același timp. În realitate, el este un maestru al alternanței rapide.”

Uită-te la desktopul tău

În acest moment, foarte probabil rulează simultan:

- un browser;
- unul sau mai multe terminale;
- editorul în care scrii;
- managerul de fișiere;
- serviciile de rețea;
- serverul de sunet;
- managerul de autentificare;
- poate o mașină virtuală sau un container.

Și totuși, dacă ai avea un singur nucleu de procesor, acesta poate executa **o singură instrucțiune** la un moment dat.

Cum este posibil ca toate aceste aplicații să pară active simultan?

Răspunsul este unul dintre cele mai elegante mecanisme din sistemele de operare moderne.

Procesorul este un bucătar

Imaginează-ți un bucătar într-o bucătărie.

Are de pregătit:

- o supă;
- o friptură;
- un desert;
- o salată.

Dacă ar termina complet fiecare fel înainte să înceapă următorul, clienții ar aștepta foarte mult.

În schimb, el alternează:

- pune supa la fiert;
- taie legumele pentru salată;
- întoarce friptura;
- pregătește desertul;
- verifică din nou supa.

Privit din exterior, pare că gătește toate preparatele simultan.

În realitate, atenția lui este mutată foarte repede de la o activitate la alta.

Exact asta face și procesorul.

O felie de timp

Kernelul împarte timpul procesorului în intervale foarte scurte, numite **cuante de timp** (*time slices* sau *time quanta*).

Fiecare proces primește pentru scurt timp controlul procesorului.

Când timpul alocat expiră, kernelul întrerupe execuția celui proces și oferă procesorul altuia.

Acest schimb este atât de rapid încât, pentru utilizator, toate programele par că rulează simultan.

Nu este magie.

Este planificare.

Cine decide?

Nu Bash.

Nu aplicațiile.

Kernelul.

Mai precis, o componentă numită **scheduler**.

Schedulerul decide:

- ce proces rulează acum;
- cât timp primește;
- cine urmează;
- ce proces trebuie întrerupt;
- ce proces trebuie reluat.

Este, dacă vrei, dirijorul întregului sistem.

Dar dacă un proces spune...

„Mai vreau puțin timp!”

Kernelul răspunde simplu:

„Acum este rândul altuia.”

Niciun proces obișnuit nu poate monopoliza procesorul după bunul plac.

Această regulă este unul dintre motivele pentru care Linux poate rula sute sau mii de procese fără ca unul singur să le blocheze pe celelalte.

Schimbarea de context

Acum apare una dintre cele mai importante operații dintr-un sistem de operare:

context switch.

Imaginează-ți că citești trei cărți în paralel.

De fiecare dată când schimbi cartea, trebuie să-ți amintești:

- la ce pagină ai rămas;
- ce personaje erau implicate;
- ce idee urmărea autorul.

Procesorul face ceva asemănător.

Înainte să întrerupă un proces, kernelul salvează starea acestuia:

- valorile registrelor;
- contorul de program;
- indicatorul stivei;
- alte informații necesare pentru reluarea execuției.

Când procesul primește din nou procesorul, toate aceste informații sunt restaurate, iar execuția continuă exact din punctul în care fusese întreruptă.

Procesul nu „știe” că a fost oprit.

Pentru el, timpul pare continuu.

O iluzie perfectă

Aceasta este frumusețea multitaskingului.

Fiecare proces are impresia că deține procesorul.

În realitate, procesorul este împărțit între toate procesele active.

Este o iluzie creată de scheduler și de viteza cu care are loc schimbarea de context.

Dar procesoarele moderne?

Până acum am discutat despre un singur nucleu.

Procesoarele moderne au:

- 4 nuclee;
- 8 nuclee;
- 16 nuclee;
- 32 de nuclee sau chiar mai multe.

În acest caz, mai multe procese pot rula cu adevărat în paralel.

Totuși, principiul rămâne același.

Dacă ai mai multe procese decât nuclee — și aproape întotdeauna ai — schedulerul continuă să alterneze execuția lor.

Caseta „În spatele cortinei”

Cât costă un context switch?

Schimbarea de context nu este gratuită.

Kernelul trebuie să salveze și să restaureze starea proceselor, iar acest lucru consumă timp.

Dacă sistemul schimbă prea des procesul activ, o parte semnificativă din timpul procesorului poate fi consumată doar pentru aceste operații administrative.

Acesta este unul dintre motivele pentru care programele bine proiectate și sistemele bine configurate pot avea performanțe sensibil mai bune: nu doar pentru că execută cod eficient, ci și pentru că reduc schimbările inutile de context.

Mai târziu, când vom studia monitorizarea performanței, vom vedea cum putem observa acest fenomen cu instrumente precum `vmstat`, `pidstat`, `sar` și `perf`.

Administrator Notes

Atunci când un server pare „lent”, tentația este să presupui că procesorul este prea slab.

Dar experiența arată că problema poate fi în altă parte:

- prea multe procese care concurează pentru CPU;
- multe schimbări de context;

- procese care așteaptă accesul la disc;
- procese blocate în operații de rețea.

Un administrator experimentat nu întreabă doar „cât este utilizarea CPU?”, ci și „**cum își petrece procesorul timpul?**”

Această întrebare conduce aproape întotdeauna la un diagnostic mai bun.

Gândește ca un inginer

Imaginează-ți un restaurant în care un singur ospătar deservește zece mese.

Dacă aleargă haotic între ele, toți clienții vor fi nemulțumiți.

Dacă își organizează inteligent traseul și timpul, poate crea impresia că este prezent peste tot.

Schedulerul este acel ospătar.

Nu pentru că este cel mai rapid, ci pentru că știe **când** și **unde** să își îndrepte atenția.

Aceasta este o lecție care depășește Linux. În orice sistem complex, performanța înseamnă adesea organizare mai bună, nu doar resurse mai multe.

Ce trebuie să reții

- Un nucleu de procesor execută o singură instrucțiune la un moment dat.
 - Iluzia simultaneității este creată prin alternarea rapidă a proceselor.
 - Schedulerul kernelului decide ce proces rulează și pentru cât timp.
 - Schimbarea de context permite reluarea execuției fără ca procesul să observe întreruperea.
 - Procesoarele multicore execută cu adevărat mai multe procese în paralel, dar schedulerul rămâne esențial.
 - Performanța unui sistem depinde atât de puterea hardware, cât și de modul în care sunt planificate procesele.
-

Notă de autor

Aici aș încheia cu o idee care va reveni de multe ori în carte:

Linux nu încearcă să facă totul în același timp. Linux încearcă să decidă inteligent ce merită făcut în fiecare clipă.

Este o idee tehnică, dar și o lecție de proiectare a sistemelor.

Capitolul 1

1.15 Totul începe cu o întrebare

„Calculatorul nu ghicește niciodată. Fiecare acțiune începe cu o întrebare și continuă cu un răspuns.”

Cum gândește un calculator?

Una dintre cele mai mari diferențe dintre oameni și calculatoare este modul în care iau decizii.

Noi folosim experiența, intuiția și, uneori, instinctul.

Calculatorul nu are nimic din toate acestea.

El nu presupune.

Nu ghicește.

Nu interpretează intenții.

Nu completează informații lipsă.

Pentru fiecare pas pe care îl face există o întrebare foarte precisă.

Și un răspuns foarte precis.

Acest mod de gândire este prezent în fiecare nivel al sistemului de operare.

Să urmărim o comandă

Ai tastat:

```
cat raport.txt
```

Pentru tine, intenția este clară:

„Vreau să citesc fișierul.”

Dar pentru sistem, această intenție trebuie descompusă într-o succesiune de întrebări.

Întrebarea 1

Există comanda `cat`?

Dacă nu există:

```
command not found
```


Dacă există:
mergem mai departe.

Întrebarea 2

Unde se află executabilul?

Bash începe căutarea în PATH.

Dacă îl găsește:

continuă.

Întrebarea 3

Poate fi executat?

Kernelul verifică:

- există?
 - este executabil?
 - utilizatorul are drepturi?
-

Întrebarea 4

Poate fi încărcat în memorie?

Dacă da:

procesul începe.

Întrebarea 5

Programul `cat` întreabă:

Există fișierul `raport.txt`?

Întrebarea 6

Kernelul răspunde:

Da.

sau

Nu.

Întrebarea 7

Dacă există:

Utilizatorul are dreptul să îl citească?

Întrebarea 8

Dacă răspunsul este afirmativ:

Citește primul bloc.

Întrebarea 9

Mai există date?

Dacă da:

continuă.

Dacă nu:

încheie.

Observă ceva extraordinar

Niciun pas nu presupune nimic.

Niciun pas nu este sărit.

Linux este un lanț de întrebări.

Iar fiecare răspuns determină următoarea întrebare.

Acesta este motivul pentru care depanarea unui sistem poate fi abordată metodic.

Nu trebuie să ghicești.

Trebuie doar să descoperi unde lanțul s-a întrerupt.

Întrebările potrivite

De multe ori, un cursant întreabă:

„De ce nu merge?”

Întrebarea este firească, dar prea generală.

Un administrator experimentat o transformă în întrebări mai mici:

- Comanda există?
- Programul pornește?
- Fișierul există?
- Permisunile sunt corecte?
- Sistemul de fișiere este montat?
- Discul răspunde?
- Kernelul a raportat o eroare?

Dintr-o singură întrebare vagă au apărut mai multe întrebări precise.

Aceasta este esența diagnosticării.

De ce este important acest mod de gândire?

Pentru că este universal.

Nu funcționează doar în Bash.

Funcționează în:

- rețelistică;
- baze de date;
- cloud;
- containere;
- securitate;
- dezvoltare software.

Un inginer bun nu încearcă să memoreze toate răspunsurile.

Învăță să formuleze întrebările corecte.

Caseta „În spatele cortinei”

De ce **strace** este atât de valoros?

Mai târziu vom învăța instrumentul **strace**.

Mulți îl descriu simplu:

„Afișează apelurile de sistem.”

Eu prefer altă definiție.

`strace` îți permite să vezi **întrebările** pe care un proces le adresează kernelului și răspunsurile primite.

De exemplu:

```
open(...)  
read(...)  
stat(...)  
close(...)
```

Nu sunt doar funcții.

Sunt dialogul dintre aplicație și kernel.

Când vei privi `strace` în acest fel, nu vei mai vedea o listă de apeluri criptice, ci povestea execuției unei comenzi.

Administrator Notes

Când un utilizator spune:

„Nu merge.”

evită să cauți imediat soluții.

Începe prin a reconstrui lanțul întrebărilor.

Unde apare primul răspuns neașteptat?

Acolo este, de cele mai multe ori, cauza problemei.

Această abordare economisește timp și reduce riscul de a rezolva doar simptomele.

Gândește ca un inginer

Imaginează-ți că un detectiv ajunge la locul unei fapte.

Nu începe cu concluziile.

Începe cu întrebările.

- Cine a fost aici?
- Ce s-a întâmplat prima dată?
- Ce urme există?
- Ce lipsește?
- Ce nu se potrivește?

Un administrator Linux lucrează la fel.

Un sistem lasă urme:

- loguri;
- procese;
- coduri de eroare;
- apeluri de sistem;
- conexiuni;
- permisiuni.

Rolul nostru este să învățăm să le citim.

Ce trebuie să reții

- Linux ia decizii printr-un lanț de întrebări și răspunsuri.
- Fiecare pas al execuției verifică o condiție înainte de a continua.
- Diagnosticarea eficientă înseamnă identificarea punctului în care lanțul se întrerupe.
- Întrebările corecte sunt mai valoroase decât răspunsurile memorate.
- Acest mod de gândire este aplicabil în întreaga inginerie software și de sisteme.

Capitolul 2

2.1 Ce se întâmplă când pornește Bash?

„Înainte ca utilizatorul să tasteze prima comandă, Bash a luat deja o serie de decizii importante. Înțelegerea acestor pași explică de ce două terminale aparent identice se pot comporta diferit.”

Introducere

Până acum am urmărit drumul unei comenzi prin sistem. Am văzut cum colaborează terminalul, Bash, kernelul și programele externe.

Acum schimbăm perspectiva.

În loc să urmărim o comandă, vom urmări chiar **nașterea procesului Bash**.

Este un moment important, deoarece multe dintre caracteristicile pe care le observăm ulterior — variabilele de mediu, aliasurile, funcțiile, promptul sau istoricul comenzilor — sunt stabilite chiar în primele momente ale execuției shell-ului.

Cine pornește Bash?

Mulți utilizatori spun:

„Am deschis un terminal.”

Din punct de vedere tehnic, ceea ce s-a întâmplat este puțin diferit.

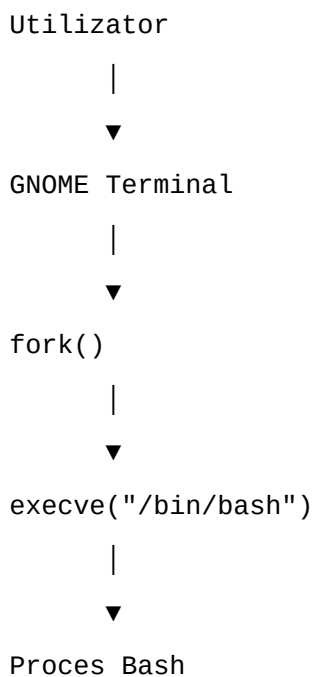
Să presupunem că utilizatorul deschide aplicația **GNOME Terminal**.

Aceasta este doar un emulator de terminal. Ea nu este Bash și nici kernelul.

În momentul în care se creează o fereastră nouă, emulatorul solicită sistemului de operare pornirea unui proces nou.

În majoritatea distribuțiilor Linux, acel proces va fi Bash (sau shell-ul configurat pentru utilizator).

Schema simplificată este:



Din acest moment începe execuția codului sursă al lui Bash.

De unde știe sistemul ce shell să pornească?

Răspunsul nu este „pentru că există Bash”.

Fiecare utilizator are asociat un **shell de autentificare** (*login shell*).

Acesta este memorat în fișierul:

`/etc/passwd`

De exemplu:

`daniel:x:1000:1000:Daniel Cristea:/home/daniel:/bin/bash`

Ultimul câmp reprezintă shell-ul implicit al utilizatorului.

Poate fi:

`/bin/bash`

/bin/sh
/bin/zsh
/bin/ksh
/bin/fish

sau orice alt shell instalat și permis de sistem.

Acesta este motivul pentru care doi utilizatori de pe aceeași mașină pot primi shell-uri diferite imediat după autentificare.

Bash începe prin a se analiza pe sine

Imediat după pornire, Bash nu afișează promptul.

Mai întâi încearcă să înțeleagă **în ce mod a fost lansat**.

Printre întrebările pe care și le pune se numără:

- Sunt un *login shell*?
- Sunt un shell interactiv?
- Execut un script?
- Rulez în modul POSIX?
- Am fost pornit ca `sh`?
- Sunt un shell restricționat?

Răspunsurile la aceste întrebări vor influența întregul comportament ulterior al shell-ului.

Acesta este motivul pentru care aceeași comandă poate avea rezultate diferite în contexte diferite.

De ce este important acest lucru?

Imaginează-ți două ferestre de terminal.

În prima funcționează completarea cu tasta **Tab**, istoricul comenzilor și aliasurile.

În a doua nu.

Prima reacție este:

„Ce s-a stricat?”

De multe ori nu s-a stricat nimic.

Pur și simplu Bash a fost pornit într-un alt mod și a citit alte fișiere de configurare.

Vom studia aceste scenarii în capitolele următoare.

Caseta „În spatele cortinei”

De ce trebuie să știe Bash cum a fost pornit?

Pentru că un shell folosit interactiv și un shell care execută un script au scopuri diferite.

Într-un terminal interactiv sunt utile:

- promptul;
- istoricul comenzilor;
- completarea automată;
- aliasurile.

În schimb, într-un script aceste funcționalități nu doar că sunt inutile, dar pot produce efecte nedorite sau pot încetini execuția.

De aceea Bash își adaptează comportamentul încă din primele momente ale execuției.

Administrator Notes

Una dintre cele mai frecvente situații întâlnite în administrarea Linux este următoarea:

„Scriptul funcționează când îl rulez manual, dar nu funcționează din `cron`.”

În multe cazuri, cauza nu este scriptul, ci faptul că Bash rulează într-un mediu diferit:

- alte variabile de mediu;
- alt `PATH`;
- alte fișiere de inițializare;
- shell neinteractiv.

Înainte să modifice scriptul, verifică întotdeauna **cum a fost pornit shell-ul**.

Experiment

Într-un terminal, rulează:

```
echo "$0"
```

```
echo "$-"
```

```
ps -p $$ -o pid,ppid,cmd
```

Încă nu vom explica în detaliu toate aceste rezultate. Scopul experimentului este să observi că Bash îți oferă informații despre propria identitate și despre modul în care rulează.

Vom reveni asupra fiecărei comenzi în capitolele dedicate.

Ce trebuie să reții

- Bash este un proces pornit de un alt proces.
- Shell-ul implicit al utilizatorului este definit în `/etc/passwd`.
- La pornire, Bash determină modul în care a fost lansat.
- Modul de pornire influențează comportamentul shell-ului și fișierele de configurare citite.
- Multe probleme aparent misterioase sunt cauzate de diferențe între un shell interactiv și unul neinteractiv.

Capitolul 2

2.2 Login Shell, Interactive Shell și Non-Interactive Shell

„Două instanțe Bash pot avea aceeași versiune și același utilizator, dar se pot comporta complet diferit. Motivul nu este întâmplător: Bash își adaptează comportamentul în funcție de modul în care a fost pornit.”

Introducere

Una dintre cele mai frecvente surse de confuzie în administrarea sistemelor Linux apare atunci când aceeași comandă produce rezultate diferite în contexte aparent identice.

Un alias există într-un terminal, dar nu și într-un script.

O variabilă de mediu este prezentă după autentificare, dar lipsește într-o sesiune `cron`.

Promptul este personalizat într-o fereastră de terminal, însă nu și într-o conexiune automată prin SSH.

În majoritatea cazurilor, cauza nu este o eroare a lui Bash, ci faptul că **shell-ul a fost pornit într-un mod diferit**.

Pentru a înțelege aceste diferențe, trebuie să facem distincția între două concepte independente:

- **login shell;**
- **interactive shell.**

Aceste concepte sunt adesea confundate, deși descriu aspecte diferite ale execuției lui Bash.

Două întrebări diferite

La pornire, Bash încearcă să răspundă la două întrebări fundamentale:

Prima întrebare:

Sunt un *login shell*?

A doua întrebare:

Sunt un *interactive shell*?

Observă că nu este aceeași întrebare.

Un shell poate fi:

- login și interactiv;
- login și neinteractiv;
- non-login și interactiv;
- non-login și neinteractiv.

Aceste combinații determină ce fișiere de configurare vor fi citite și ce funcționalități vor fi activate.

Ce este un Login Shell?

Un **login shell** este shell-ul care pornește la începutul unei sesiuni de autentificare.

Scopul lui principal este să pregătească mediul utilizatorului.

În această etapă sunt încărcate informații precum:

- variabile de mediu;
- PATH;
- limba sistemului (LANG, LC_*);
- editorul implicit;
- configurații generale ale sesiunii.

În mod tradițional, un login shell este primul shell pe care îl primești după autentificarea într-un sistem.

Exemple de Login Shell

În mod obișnuit, Bash rulează ca **login shell** în situații precum:

- autentificarea pe consola text (TTY);
- autentificarea prin SSH;
- unele sesiuni grafice (în funcție de distribuție și display manager);
- pornirea explicită cu:

```
bash --login
```

sau

```
bash -l
```

Cum recunoști un Login Shell?

Există mai multe metode.

Una dintre cele mai simple este:

```
echo "$0"
```

Dacă rezultatul începe cu caracterul -, de exemplu:

```
-bash
```

înseamnă că Bash rulează ca **login shell**.

Important: acesta este un indiciu folosit de multe implementări și utilitare Unix, dar nu este singura metodă prin care poate fi identificat. Mai târziu vom vedea și alte tehnici.

Ce este un Interactive Shell?

Acum schimbăm complet perspectiva.

Un **interactive shell** nu este definit prin modul în care a fost pornit.

El este definit prin **modul în care comunică cu utilizatorul**.

Dacă Bash așteaptă comenzi tastate de un om și răspunde imediat la ele, atunci este un shell interactiv.

Acesta oferă funcționalități precum:

- prompt (PS1);
- istoricul comenzilor;
- completarea cu Tab;
- editarea liniei de comandă prin Readline;
- aliasuri;
- controlul joburilor (fg, bg, jobs).

Aceste facilități există pentru că utilizatorul este prezent și interacționează direct cu shell-ul.

Cum verifici dacă shell-ul este interactiv?

Una dintre cele mai elegante metode este:

```
echo "$-"
```

Dacă rezultatul conține litera:

```
i
```

atunci shell-ul este interactiv.

Exemplu:

himBHs

Observă litera **i**.

Ea indică faptul că Bash rulează în mod interactiv.

Ce este un Non-Interactive Shell?

Acesta este cazul întâlnit cel mai des în scripting.

De exemplu:

```
bash backup.sh
```

sau

```
./backup.sh
```

Aici Bash nu așteaptă comenzi de la utilizator.

El citește comenzile dintr-un fișier și le execută una după alta.

Nu are nevoie de:

- prompt;
- istoric;
- completare automată;
- editarea liniei de comandă.

De aceea, multe dintre funcționalitățile specifice unui shell interactiv sunt dezactivate.

O matrice utilă

Până acum avem două caracteristici independente.

Le putem reprezenta astfel:

Login Interactive		Exemplu
Da	Da	Autentificare SSH urmată de lucrul în terminal
Da	Nu	Rar; de exemplu, anumite scripturi lansate într-un context de login
Nu	Da	Deschiderea unei noi ferestre GNOME Terminal într-o sesiune deja autentificată
Nu	Nu	Execuția unui script Bash dintr-un alt proces

Această clasificare este foarte utilă atunci când încerci să înțelegi de ce Bash citește sau ignoră anumite fișiere de configurare.

Caseta „În spatele cortinei”

De ce face Bash această diferență?

Pentru că inițializarea completă a mediului poate consuma timp și poate produce efecte secundare.

Imaginează-ți un script care este executat de mii de ori pe oră.

Dacă la fiecare execuție Bash ar încărca istoricul comenzilor, ar personaliza promptul și ar defini aliasuri destinate utilizatorului, ar irosi resurse și ar introduce comportamente nedorite.

Prin separarea clară între shell-urile de login și cele interactive, Bash poate încărca doar configurațiile relevante pentru contextul în care rulează.

Administrator Notes

Una dintre cele mai frecvente întrebări în administrarea Linux este:

„De ce funcționează în terminal, dar nu funcționează în cron?”

În foarte multe cazuri, răspunsul este simplu:

`crOn` pornește un shell neinteractiv și cu un mediu mult mai restrâns.

Consecințele pot fi:

- un `PATH` diferit;
- lipsa anumitor variabile de mediu;
- aliasuri inexistente;
- funcții Bash indisponibile.

Înainte să modifice scriptul, verifică întotdeauna în ce tip de shell rulează.

Experiment

Încearcă următoarele comenzi în terminal:

```
echo "$0"
```

```
echo "$-"
```

```
shopt login_shell
```

Observă rezultatele.

Nu încerca încă să le memorezi.

În capitolele următoare vom reveni asupra fiecărei informații și o vom explica în contextul fișierelor de inițializare.

Ce trebuie să reții

- **Login shell** și **interactive shell** sunt concepte diferite.
- Un login shell pregătește mediul utilizatorului.
- Un interactive shell este optimizat pentru dialogul direct cu utilizatorul.
- Același Bash poate avea comportamente diferite în funcție de modul în care a fost lansat.
- Înțelegerea acestor diferențe este esențială pentru depanarea problemelor legate de mediu și configurare.

Capitolul 2

2.3 Cum își construiește Bash mediul de lucru?

„Înainte de primul prompt, Bash citește o serie de fișiere de configurare. Ordinea în care le citește explică aproape toate diferențele dintre două sesiuni aparent identice.”

Introducere

Atunci când Bash pornește, nu afișează imediat promptul.

Mai întâi își construiește mediul de lucru.

Acest proces este asemănător cu pregătirea unui birou înainte de începerea unei zile de lucru.

Se deschid sertarele.

Se așază documentele.

Se verifică instrumentele.

Se pregătește spațiul.

Abia apoi începe activitatea.

Și Bash face exact același lucru.

Ce înseamnă "mediu de lucru"?

Prin **mediu de lucru** (*execution environment*) înțelegem totalitatea informațiilor de care Bash are nevoie pentru a funcționa.

Printre acestea se numără:

- variabilele de mediu;
- variabilele shell;
- aliasurile;
- funcțiile;

- promptul (PS1);
- istoricul comenzilor;
- opțiunile Bash (set, shopt);
- completarea automată;
- localizarea (LANG, LC_ALL);
- editorul implicit;
- variabila PATH.

Toate acestea contribuie la comportamentul shell-ului.

De unde provin aceste informații?

Nu sunt compilate în Bash.

Ele provin din fișiere de configurare.

Aceste fișiere sunt citite într-o ordine bine definită.

Această ordine este foarte importantă.

Dacă o înțelegi, vei putea explica aproape orice diferență de comportament între două sesiuni Bash.

Primul nivel – Configurația globală

Primul fișier important este:

`/etc/profile`

Acesta este administrat de sistem și afectează toți utilizatorii.

În general, aici sunt definite configurații comune:

- PATH global;
- variabile de mediu comune;
- setări regionale;
- inițializarea unor aplicații.

Administratorul sistemului îl modifică rar și cu atenție, deoarece orice schimbare afectează toate sesiunile de login.

Al doilea nivel – Configurația utilizatorului

După configurarea globală, Bash caută configurația personală a utilizatorului.

Aici apare un comportament foarte important.

Bash caută, în această ordine:

`~/ .bash_profile`

dacă nu există:

`~/ .bash_login`

iar dacă nici acesta nu există:

`~/ .profile`

Și aici este un detaliu esențial:

Bash **nu le citește pe toate trei**.

Le verifică în ordine și îl folosește pe primul pe care îl găsește.

Aceasta este una dintre cele mai frecvente surse de confuzie.

Mulți utilizatori cred că toate aceste fișiere sunt executate.

Nu este adevărat.

O analogie simplă

Imaginează-ți trei uși.

Bash încearcă prima ușă.

Dacă se deschide, intră și nu le mai verifică pe celelalte.

Dacă este închisă, încearcă a doua.

Dacă și aceasta este închisă, încearcă ultima.

Nu intră niciodată prin toate trei.

Ce se întâmplă cu `.bashrc`?

Foarte mulți utilizatori cred că `.bashrc` este primul fișier citit de Bash.

În realitate, lucrurile sunt diferite.

Pentru un **login shell**, Bash **nu citește automat** fișierul:

`~/ .bashrc`

Acesta este unul dintre cele mai importante lucruri pe care trebuie să le rețină orice administrator.

Atunci de ce pare că `.bashrc` este executat?

Pentru că, în majoritatea distribuțiilor Linux, fișierul:

`~/ .bash_profile`

conține o secțiune asemănătoare cu aceasta:

```
if [ -f ~/ .bashrc ]; then
```



```
. ~/.bashrc  
fi
```

Observă ce se întâmplă.

Nu Bash citește direct `.bashrc`.

Utilizatorul, prin propriul fișier `.bash_profile`, îi spune lui Bash să îl încarce.

Această diferență este subtilă, dar extrem de importantă.

De ce există această separare?

Întrebarea este firească.

De ce nu pune Bash toate configurațiile într-un singur fișier?

Răspunsul ține de responsabilități.

Fișierele de login sunt destinate inițializării sesiunii.

Aici definim lucruri care trebuie făcute o singură dată:

- `PATH`;
- variabile de mediu;
- configurări generale.

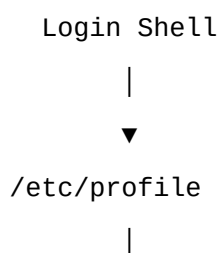
În schimb, `.bashrc` este destinat shell-urilor interactive.

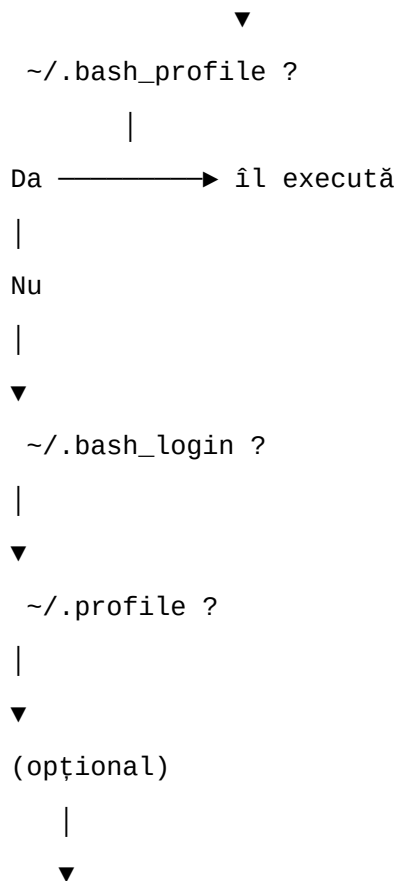
Aici găsim de obicei:

- aliasuri;
- promptul;
- funcții Bash;
- completări;
- configurarea culorilor;
- opțiuni `shopt`;
- configurarea istoricului.

Separarea evită executarea inutilă a acelorași configurații în contexte unde nu sunt necesare.

Diagramă





source ~/.bashrc

Această diagramă merită ținută minte. Ea explică o mare parte din comportamentul unui login shell.

Caseta „În spatele cortinei”

De ce distribuțiile folosesc **source ~/.bashrc**?

Pentru că astfel păstrează o singură copie a configurațiilor interactive.

Imaginează-ți că aliasurile și promptul ar exista și în `.bash_profile`, și în `.bashrc`. Orice modificare ar trebui făcută în două locuri, crescând riscul de inconsistențe.

Prin încărcarea explicită a `.bashrc` din `.bash_profile`, configurațiile interactive rămân centralizate într-un singur fișier.

Este un exemplu elegant de reutilizare și evitare a duplicării configurației.

Capcană frecventă

Un administrator adaugă un alias în:

```
~/.profile
```

și apoi deschide o nouă fereastră GNOME Terminal.

Aliasul nu apare.

Concluzia este:

"Linux nu funcționează."

Concluzia corectă este:

Shell-ul pornit nu a citit fișierul în care a fost definit aliasul.

Înainte să cauți erori, verifică întotdeauna ce fișiere au fost încărcate și în ce ordine.

Experiment

Rulează:

```
ls -la ~ | grep -E '(\.bash_profile|\.bash_login|\.profile|\.bashrc)'
```

Observă ce fișiere există în contul tău.

Apoi afișează conținutul lor:

```
cat ~/.bash_profile
```

```
cat ~/.bashrc
```

Dacă unul dintre fișiere nu există, este perfect normal. Distribuțiile Linux diferă în modul în care își organizează configurațiile implicite.

Ce trebuie să reții

- Bash își construiește mediul de lucru citind fișiere de configurare.
- `/etc/profile` conține configurații globale pentru login shell-uri.
- Bash verifică `~/.bash_profile`, `~/.bash_login` și `~/.profile` în această ordine și execută doar primul găsit.
- `.bashrc` nu este citit automat de un login shell; de regulă este încărcat explicit din `~/.bash_profile`.
- Separarea între fișierele de login și cele interactive permite o organizare mai clară și evită executarea inutilă a configurațiilor.

Capitolul 2

2.4 Cum demonstrezi că Bash citește fișierele de inițializare?

„În inginerie nu este suficient să afirmi că un mecanism există. Trebuie să poți demonstra funcționarea lui.”

Introducere

Până acum am discutat despre ordinea în care Bash citește fișierele de configurare.

Dar apare o întrebare firească.

Cum putem verifica faptul că Bash chiar citește aceste fișiere?

Nu presupunem.

Nu memorăm.

Demonstrăm.

Acesta va fi principiul întregii cărți.

Primul experiment

Vom modifica temporar unul dintre fișierele de inițializare.

De exemplu:

```
echo "Pornesc ~/.bashrc"
```

adăugat la sfârșitul fișierului:

```
~/.bashrc
```

Acum închide terminalul și deschide unul nou.

Dacă mesajul apare imediat după deschiderea terminalului, ai demonstrat că fișierul a fost executat.

Nu pentru că scrie într-o carte.

Ci pentru că l-ai văzut.

Dar dacă mesajul nu apare?

Foarte bine.

Acum începe adevărata investigație.

Întrebările sunt:

- Terminalul pornește un login shell?
- Bash citește `.bash_profile`?
- `.bash_profile` execută `.bashrc`?
- Există eroare de sintaxă?
- Fișierul este citit deloc?

Observă diferența.

Nu mai ghicim.

Construim o ipoteză și o verificăm.

Al doilea experiment

Adaugă în:

```
~/.bash_profile
```

următoarea linie:

```
echo "Pornesc ~/.bash_profile"
```

Deschide o nouă sesiune SSH.

Dacă mesajul apare înaintea promptului, ai demonstrat că login shell-ul citește acel fișier.

Al treilea experiment

Acum creează un script simplu:

```
#!/bin/bash
```

```
echo "Script pornit"
```

Rulează-l astfel:

```
./script.sh
```

Observă că mesajele din `.bashrc` nu apar.

De ce?

Pentru că Bash rulează acum într-un context diferit.

Aceasta este o dovadă practică că nu toate instanțele Bash citesc aceleași fișiere.

O observație importantă

În această carte vom evita pe cât posibil demonstrațiile bazate exclusiv pe afirmații.

În schimb, vom încerca să răspundem prin experimente.

Acest lucru are două avantaje:

1. Cititorul capătă încredere în mecanism.
2. Învăță cum să investigheze singur sistemul.

Acest mod de învățare este mult mai valoros decât memorarea.

Instrumentele investigatorului

Un administrator Linux seamăna foarte mult cu un detectiv.

Nu presupune.

Caută dovezi.

În Bash, printre cele mai utile instrumente de investigație se numără:

type

which

command -V

declare

set

shopt

env

printenv

history

help

compgen

Aceste comenzi nu rezolvă probleme.

Ele îți arată cum funcționează Bash.

Vom învăța fiecare dintre ele în profunzime.

Caseta „În spatele cortinei”

De ce prefer experimentul?

Pentru că sistemele Linux evoluează.

Distribuțiile diferă.

Versiunile Bash diferă.

Dar mecanismele fundamentale rămân aceleași.

Dacă înveți să verifici singur comportamentul sistemului, nu mai depinzi de memoria unei comenzi sau de un articol găsit pe internet.

Aceasta este una dintre cele mai importante abilități ale unui administrator.

Capcană frecventă

Foarte mulți utilizatori modifică:

`~/ .bashrc`

și apoi spun:

"Nu s-a schimbat nimic."

Problema nu este neapărat fișierul.

Poate fi:

- shell-ul deja pornit;
- lipsa reîncărcării configurației;
- modificarea unui alt fișier decât cel folosit de sesiunea curentă;
- o eroare de sintaxă care întrerupe execuția.

Înainte de a trage concluzii, verifică întotdeauna fiecare ipoteză.

Administrator Notes

În producție, evită să modifici simultan mai multe fișiere de inițializare.

Fă o singură modificare.

Testează-o.

Documentează-o.

Abia apoi treci la următoarea.

Această disciplină simplifică enorm depanarea și reduce riscul de a introduce probleme greu de identificat.

Experiment

În această carte vom avea două tipuri de experimente:

Experimente de observare

Scopul lor este să vezi cum funcționează sistemul.

De exemplu:

`echo "$-"`

sau

`type cd`

Experimente de modificare

Aici schimbăm controlat comportamentul sistemului.

De exemplu:

- modificăm PATH;
- adăugăm un alias;
- redefinim o funcție;
- schimbăm promptul;
- observăm efectul.

Aceste experimente trebuie făcute cu atenție și, atunci când este cazul, într-o mașină virtuală sau într-un mediu de test.

Ce trebuie să reții

- În această carte afirmațiile importante vor fi susținute prin experimente.
- Bash poate fi investigat folosind propriile sale instrumente (`type`, `declare`, `set`, `shopt`, `env` etc.).
- Problemele trebuie abordate prin formularea și verificarea ipotezelor.
- O modificare controlată este mai ușor de înțeles și de depanat decât mai multe modificări simultane.

Capitolul 2

2.5 De ce funcționează într-un terminal, dar nu într-un script?

„Este una dintre cele mai frecvente întrebări din lumea Bash. Răspunsul nu ține de script, ci de mediul în care acesta rulează.”

Problema

Ai definit un alias:

```
alias ll='ls -aF'
```

În terminal funcționează perfect:

```
ll
```

Dar creezi scriptul:

```
#!/bin/bash
```


ll

și obții:

ll: command not found

Prima reacție este:

"Aliasul nu funcționează."

Nu.

Aliasul funcționează exact cum a fost proiectat.

Problema este alta.

Primul pas

Un inginer nu începe cu soluția.

Începe cu întrebările.

În cazul nostru:

1. Rulează același Bash?
2. Rulează în același mod?
3. Au fost citite aceleași fișiere?
4. Există același mediu de execuție?

Abia apoi încercăm să explicăm diferența.

Ce se întâmplă în terminal?

Într-o sesiune interactivă, Bash citește de regulă:

~/ .bashrc

Acolo găsește:

- aliasuri;
- funcții;
- configurarea promptului;
- opțiuni shopt;
- completarea comenzilor.

Astfel, înainte să tastezi prima comandă, aliasul ll există deja.

Ce se întâmplă într-un script?

Acum rulezi:

```
./backup.sh
```

Kernelul pornește un nou proces Bash.

Acest Bash este, în mod normal:

- **non-login;**
- **non-interactive.**

În acest context, fișierul `~/ .bashrc` nu este încărcat automat.

Prin urmare, aliasul nu există.

Nu pentru că Bash l-a uitat.

Ci pentru că nimeni nu i l-a definit.

Demonstrație

În terminal:

```
alias ll='ls -aF'
```

```
type ll
```

Rezultatul:

```
ll is aliased to `ls -aF'
```

Acum creează scriptul:

```
#!/bin/bash
```

```
type ll
```

Rezultatul va fi asemănător cu:

```
type: ll: not found
```

Nu trebuie să ne crezi pe cuvânt.

Sistemul îți arată singur diferența.

Dar funcțiile?

Mulți presupun că funcțiile Bash sunt diferite.

Hai să verificăm.

În terminal:

```
salut() {
```

```
    echo "Salut!"  
}
```

Acum:

```
type salut
```

vei obține:

```
salut is a function
```

Creează apoi un script:

```
#!/bin/bash
```

```
salut
```

Rezultatul:

```
salut: command not found
```

Și funcțiile lipsesc.

Din același motiv.

Ele au fost definite doar în shell-ul interactiv.

De ce variabilele de mediu par diferite?

Acum apare o întrebare foarte interesantă.

Unele variabile există și în script.

Altele nu.

De exemplu:

```
export NUME="Daniel"
```

și apoi:

```
#!/bin/bash
```

```
echo "$NUME"
```

funcționează.

Dar:

```
NUME="Daniel"
```

fără `export`

nu funcționează în procesul copil.

Vom dedica un capitol întreg acestui mecanism, deoarece este unul dintre cele mai importante concepte din Bash.

Deocamdată reține doar ideea:

Nu toate variabilele sunt moștenite de procesele copil.

Caseta „În spatele cortinei”

De ce aliasurile nu sunt exportate?

Pentru că aliasurile sunt un mecanism destinat utilizatorului.

Ele simplifică tastarea comenzilor.

Nu reprezintă informații necesare unui proces copil.

În schimb, variabile precum:

PATH

HOME

LANG

TERM

descriu mediul de execuție și trebuie să poată fi transmise altor procese.

Aceasta este una dintre diferențele fundamentale dintre aliasuri și variabilele de mediu.

Greșeală din lumea reală

Un administrator creează:

```
alias rm='rm -i'
```

pentru a evita ștergerile accidentale.

Apoi scrie un script și presupune că și acolo fiecare comandă `rm` va cere confirmare.

Scriptul șterge fișiere fără nicio întrebare.

De ce?

Pentru că aliasul nu există în acel shell.

Dacă protecția este esențială, ea trebuie implementată în script sau prin alte mecanisme, nu bazându-ne pe aliasurile din sesiunea interactivă.

Administrator Notes

Dacă un script depinde de aliasuri, funcții sau setări specifice utilizatorului, întreabă-te dacă aceasta este într-adevăr intenția ta.

În general, scripturile de administrare ar trebui să fie cât mai independente de configurația interactivă a unui utilizator.

Un script robust folosește căi explicite, verifică dependențele și nu presupune existența unor aliasuri sau funcții locale.

Acest lucru îl face mai ușor de rulat din `cron`, `systemd`, SSH sau alte medii automate.

Laborator 2.2

Obiectiv

Înțelegerea diferenței dintre un shell interactiv și unul neinteractiv.

Durata

20–30 minute.

Cerințe

1. Creează un alias `ll`.
 2. Creează o funcție `salut()`.
 3. Definește o variabilă simplă și o variabilă exportată.
 4. Rulează comenzile atât în terminal, cât și dintr-un script.
 5. Notează diferențele.
-

Întrebări

1. Care elemente sunt disponibile în ambele contexte?
 2. Care dispar?
 3. De ce?
 4. Ce rol are `export`?
 5. Cum ai face ca un script să nu depindă de configurația interactivă?
-

Soluția laboratorului

Întrebarea 1

Care elemente sunt disponibile în ambele contexte?

Variabilele exportate și alte informații din mediul de execuție sunt moștenite de procesul copil.

Întrebarea 2

Care dispar?

Aliasurile, funcțiile nedefinite în noul shell și variabilele care nu au fost exportate.

Întrebarea 3

De ce?

Pentru că fiecare proces Bash are propriul spațiu de memorie. Procesul copil moștenește doar ceea ce părintele transmite explicit sau ceea ce Bash reconstruiește la pornire.

Întrebarea 4

Ce rol are `export`?

Marchează o variabilă pentru a fi inclusă în mediul transmis proceselor copil. Fără `export`, variabila rămâne locală shell-ului curent.

Întrebarea 5

Cum ai face ca un script să nu depindă de configurația interactivă?

- Folosește căi absolute sau verifică `PATH`.
- Nu te baza pe aliasuri.
- Definește în script funcțiile de care ai nevoie.
- Verifică explicit existența programelor necesare.
- Documentează cerințele de mediu.

Ce trebuie să reții

- Un shell interactiv și unul neinteractiv nu pornesc cu același mediu.
- Aliasurile și funcțiile sunt, în mod obișnuit, specifice sesiunii în care au fost definite.
- `export` permite transmiterea variabilelor către procesele copil.
- Scripturile robuste trebuie să fie cât mai independente de configurația interactivă.

Capitolul 2

2.6 Cum decide Bash ce execută?

„Când tastezi o comandă, Bash nu merge direct în `/usr/bin`. El urmează un algoritm foarte precis. Înțelegerea acestui algoritm explică numeroase situații întâlnite în administrarea sistemelor Linux.”

Introducere

Să presupunem că tastezi:

`ls`

Întrebarea pare banală.

Cum știe Bash ce înseamnă `ls`?

Merge direct în:

`/usr/bin/ls`

?

Nu.

De fapt, Bash începe o căutare.

Și această căutare urmează o ordine foarte bine definită.

Dacă înțelegi această ordine, vei putea explica de ce aceeași comandă poate executa lucruri diferite pe sisteme diferite.

Algoritmul lui Bash

Atunci când introduci un cuvânt care poate reprezenta o comandă, Bash îl caută în următoarea ordine:

1. Alias
2. Funcție Bash
3. Comandă internă (builtin)
4. Executabil găsit prin PATH

Dacă găsește o potrivire, căutarea se oprește.

Nu mai verifică pașii următori.

Această regulă este foarte importantă.

Pasul 1 – Aliasurile

Mai întâi, Bash verifică dacă numele introdus este un alias.

Exemplu:

```
alias ls='ls --color=auto'
```

Acum:

`ls`

nu mai înseamnă doar `ls`.

Bash îl rescrie intern în:

```
ls --color=auto
```

Abia apoi continuă procesarea.

Observă că aliasul nu execută nimic.

El doar rescrie linia de comandă înainte de analizarea ei completă.

Acesta este motivul pentru care aliasurile sunt foarte simple și foarte rapide.

Pasul 2 – Funcțiile Bash

Dacă nu există un alias, Bash verifică existența unei funcții.

Exemplu:

```
ls() {  
    echo "Ai apelat funcția ls"  
}
```

Acum:

```
ls
```

nu mai execută programul din `/usr/bin`.

Execută funcția.

Aceasta este o diferență majoră.

O funcție poate conține:

- alte comenzi;
- bucle;
- condiții;
- apeluri către alte funcții.

Practic, poate înlocui complet comportamentul unei comenzi.

Pasul 3 – Builtin-urile

Dacă nu există nici alias, nici funcție, Bash verifică dacă numele aparține unei comenzi interne.

Exemple:

```
cd
```

```
echo
```

```
pwd
```

```
export
```



```
read
source
history
help
```

Aceste comenzi sunt implementate chiar în codul sursă al lui Bash.

Nu există fișiere precum:

```
/usr/bin/cd
```

Pur și simplu pentru că `cd` trebuie să modifice directorul procesului Bash curent. Dacă ar fi un executabil extern, ar schimba directorul unui proces copil, iar efectul s-ar pierde imediat ce acel proces s-ar încheia.

Acesta este un exemplu excelent în care arhitectura sistemului explică de ce o comandă este builtin.

Pasul 4 – Căutarea în PATH

Abia dacă niciuna dintre etapele anterioare nu produce un rezultat, Bash începe să caute un executabil.

Cum?

Parcurgând directoarele din variabila:

```
echo "$PATH"
```

de la stânga la dreapta.

Dacă `PATH` este:

```
/usr/local/bin:/usr/bin:/bin
```

și există două executabile numite `mytool`:

```
/usr/local/bin/mytool
```

```
/usr/bin/mytool
```

va fi executat primul.

Ordinea directoarelor din `PATH` reprezintă, de fapt, o ordine de prioritate.

Demonstrație

Creează un alias:

```
alias testcmd='echo Alias'
```

Apoi o funcție:

```
testcmd() {
    echo Funcție
```

```
}
```

Acum rulează:

```
type testcmd
```

Rezultatul va arăta că Bash vede mai întâi aliasul.

El nici măcar nu ajunge să verifice funcția.

Elimină aliasul:

```
unalias testcmd
```

și rulează din nou:

```
type testcmd
```

Acum Bash identifică funcția.

Prin acest experiment ai demonstrat că ordinea de căutare nu este doar o teorie, ci un comportament observabil.

type – una dintre cele mai utile comenzi

În multe situații, primul instrument pe care ar trebui să îl folosești este:

```
type nume_comandă
```

De exemplu:

```
type cd
```

Rezultatul:

```
cd is a shell builtin
```

Sau:

```
type ls
```

Rezultatul poate fi:

```
ls is aliased to 'ls --color=auto'
```

sau

```
ls is /usr/bin/ls
```

în funcție de configurația sistemului.

type îți spune **ce va executa Bash**, nu doar unde există un fișier pe disc.

Caseta „În spatele cortinei”

De ce which și type pot afișa rezultate diferite?

Aceasta este o întrebare foarte frecventă.

Comanda:

```
which ls
```

caută, în general, executabile în directoarele din PATH.

În schimb:

```
type ls
```

întreabă chiar shell-ul Bash cum ar interpreta comanda.

Prin urmare, `type` poate afișa aliasuri, funcții și builtin-uri, în timp ce `which` poate să nu le vadă sau să le trateze diferit, în funcție de implementare.

Recomandare practică: când depanezi comportamentul lui Bash, preferă `type` și `command -V`, deoarece descriu modul în care Bash va rezolva efectiv comanda.

Greșeală din lumea reală

Un administrator instalează o versiune nouă a unui utilitar în:

```
/usr/local/bin
```

Dar în PATH, directorul apare după:

```
/usr/bin
```

Rezultatul?

Se execută în continuare versiunea veche.

Administratorul este convins că instalarea a eșuat.

În realitate, problema este doar ordinea directoarelor din PATH.

Administrator Notes

Când o comandă pare să execute „altceva” decât te aștepti, verifică întotdeauna:

1. Există un alias?
2. Există o funcție cu același nume?
3. Este o comandă builtin?
4. Ce spune `type`?
5. Care este ordinea directoarelor din PATH?

Aceste cinci verificări rezolvă un număr surprinzător de mare de situații aparent misterioase.

Cum aș depana această problemă?

Problema:

Comanda `foo` execută alt program decât mă aștept.

Pașii:

```
type foo
```

↓

```
alias | grep foo
```

↓

```
declare -f foo
```

↓

```
command -V foo
```

↓

```
echo "$PATH"
```

↓

```
which foo
```

Abia după aceste verificări are sens să concluzionezi dacă problema este în `PATH`, într-un alias sau într-o funcție.

Laborator 2.3

Obiectiv

Înțelegerea algoritmului de rezolvare a comenzilor în Bash.

Durata

25–30 minute.

Cerințe

1. Creează un alias numit `demo`.
2. Verifică-l cu `type demo`.
3. Elimină aliasul.
4. Creează o funcție cu același nume.
5. Verifică din nou cu `type`.
6. Elimină funcția.
7. Creează un script executabil `demo` într-un director personal și adaugă acel director la începutul lui `PATH`.
8. Observă cum se schimbă rezultatul.

Întrebări

1. În ce ordine a ales Bash fiecare variantă?
2. Ce comandă ți-a oferit cea mai clară explicație?
3. De ce `type` este mai util decât `which` în acest context?
4. Cum influențează ordinea din `PATH` alegerea executabilului?

Soluția laboratorului

1. Bash a ales întotdeauna primul element găsit conform algoritmului: alias → funcție → builtin → executabil.
2. `type` a oferit cele mai complete informații deoarece reflectă modul în care Bash interpretează efectiv comanda.
3. `which` se concentrează în principal pe executabilele din `PATH`, în timp ce `type` cunoaște și aliasurile, funcțiile și builtin-urile.
4. Directorul aflat primul în `PATH` are prioritate. Dacă două executabile au același nume, Bash îl va alege pe primul găsit în ordinea directoarelor.

Ce trebuie să reții

- Bash urmează un algoritm determinist pentru rezolvarea comenzilor.
- Aliasurile și funcțiile pot ascunde executabile cu același nume.
- Builtin-urile există deoarece anumite operații trebuie executate în procesul Bash curent.
- `PATH` stabilește prioritatea executabilelor externe.
- `type` este unul dintre cele mai importante instrumente pentru înțelegerea și depanarea comportamentului lui Bash.

Capitolul 2

2.7 PATH – De ce există și cum funcționează cu adevărat?

„Variabila `PATH` nu este doar o listă de directoare. Ea reprezintă mecanismul prin care Bash transformă un nume scurt, precum `ls`, într-un executabil concret aflat pe disc.”

Introducere

Scrii:

```
ls
```

și funcționează.

Nu ai scris:

```
/usr/bin/ls
```

și totuși Bash găsește programul.

Cum?

Răspunsul este:

PATH.

Dar înainte de a explica mecanismul, să ne punem o întrebare mai importantă.

De ce există PATH?

Imaginează-ți că PATH nu ar exista.

Fiecare comandă ar trebui scrisă complet.

Astfel:

```
/usr/bin/ls
```

```
/usr/bin/cat
```

```
/usr/bin/find
```

```
/usr/bin/grep
```

```
/bin/mkdir
```

```
/usr/bin/sort
```

Calculatorul ar funcționa.

Dar utilizatorul ar pierde timp și ar introduce mult mai ușor greșeli.

PATH a fost creat pentru a separa două lucruri:

- **numele logic al unei comenzi;**
- **locația fizică a executabilului.**

Această separare este una dintre ideile elegante moștenite din Unix.

Ce este PATH?

PATH este o variabilă de mediu.

Poate fi afișată astfel:

```
echo "$PATH"
```

Exemplu:

`/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin`

Observă separatorul.

Nu este spațiul.

Nu este virgula.

Este caracterul:

:

Fiecare element reprezintă un director în care Bash va căuta executabile.

Cum caută Bash?

Să presupunem că PATH conține:

`/usr/local/bin:/usr/bin:/bin`

și utilizatorul tastează:

`grep`

Bash procedează astfel:

Pasul 1

Verifică:

`/usr/local/bin/grep`

Există?

Dacă da...

căutarea se oprește.

Nu mai verifică nimic.

Dacă nu...

continuă.

Pasul 2

Verifică:

`/usr/bin/grep`

Există?

Dacă da...

îl execută.

Dacă nu...

continuă.

Pasul 3

Verifică:

`/bin/grep`

Și tot așa.

Observă ceva foarte important.

PATH nu este o listă.

Este o **ordine de prioritate**.

Acesta este unul dintre cele mai importante lucruri pe care trebuie să le rețină cititorul.

Demonstrație practică

Creează două directoare:

```
mkdir ~/test1
```

```
mkdir ~/test2
```

În fiecare creează un executabil numit:

demo

În primul:

```
#!/bin/bash
```

```
echo "Rulează test1"
```

În al doilea:

```
#!/bin/bash
```

```
echo "Rulează test2"
```

Acordă drept de execuție:

```
chmod +x ~/test1/demo
```

```
chmod +x ~/test2/demo
```

Acum setează:

```
export PATH="$HOME/test1:$HOME/test2:$PATH"
```

Rulează:

demo

Rezultatul:

Rulează test1

Schimbă ordinea:

```
export PATH="$HOME/test2:$HOME/test1:$PATH"
```

Rulează din nou:

demo

Rezultatul:

Rulează test2

Nu s-a schimbat programul.

Nu s-a schimbat Bash.

S-a schimbat doar **prioritatea directoarelor**.

Acest experiment valorează mai mult decât o pagină de definiții.

De ce primul director are prioritate?

Imaginează-ți o bibliotecă.

Îi ceri bibliotecarului:

„Adu-mi cartea *Linux*.”

El merge la primul raft.

Dacă o găsește, ți-o aduce.

Nu mai verifică celelalte rafturi.

PATH funcționează exact la fel.

PATH și securitatea

Acum apare o întrebare foarte importantă.

De ce nu este recomandat să adaugi:

.

la începutul lui PATH?

Adică:

```
export PATH=".: $PATH"
```

Imaginează-ți că intri într-un director primit de la altcineva.

În acel director există un fișier executabil numit:

ls

Dacă directorul curent este primul în PATH, atunci simpla comandă:

ls

nu va mai executa programul din:

/usr/bin/ls

ci pe cel aflat în directorul curent.

Acesta este un risc de securitate bine cunoscut.

Din acest motiv, majoritatea distribuțiilor moderne evită să includă `.` în PATH sau îl plasează, dacă este folosit, la sfârșitul listei.

Caseta „În spatele cortinei”

Ce este hash?

De fiecare dată când Bash găsește un executabil prin PATH, poate memora rezultatul într-un tabel intern numit **hash table**.

Scopul este simplu.

Data viitoare când rulezi aceeași comandă, Bash nu mai parcurge toate directoarele din PATH.

Consultă mai întâi acest tabel.

Poți vedea conținutul lui astfel:

hash

Exemplu:

hits	command
12	/usr/bin/ls
8	/usr/bin/grep

Observă coloana **hits**.

Ea indică de câte ori Bash a folosit acea intrare din tabelul de hash.

De ce există hash -r?

Să presupunem că instalezi o versiune nouă a unei comenzi într-un director aflat mai sus în PATH.

Bash poate continua să folosească vechea locație memorată.

În acest caz:

hash -r

șterge întregul tabel de hash.

La următoarea execuție, Bash caută din nou executabilul și reconstruiește informațiile.

Acum înțelegi de ce această comandă există.

Nu este o comandă „misterioasă”, ci o consecință firească a mecanismului de optimizare.

Greșeală din lumea reală

Un administrator instalează o versiune nouă a `terraform` în `/usr/local/bin`.

Modifică `PATH`.

Dar Bash continuă să execute versiunea veche.

El verifică `PATH` de mai multe ori și nu găsește problema.

În realitate, Bash folosea locația memorată în tabelul de hash.

După:

```
hash -r
```

noua versiune este executată imediat.

Acesta este un exemplu clasic în care înțelegerea mecanismului economisește mult timp.

Administrator Notes

Ori de câte ori ai impresia că Bash execută „programul greșit”, verifică în această ordine:

```
type comandă
```

↓

```
echo "$PATH"
```

↓

```
hash
```

↓

```
hash -r
```

↓

```
type comandă
```

Această secvență simplă rezolvă numeroase situații legate de `PATH` și de memorarea executabilelor.

Cum aş depana această problemă?

Problema:

După instalarea unei versiuni noi a unei comenzi, Bash continuă să ruleze versiunea veche.

Pașii:

1. Verific ce execută Bash (`type comandă`).
2. Verific ordinea directoarelor din `PATH`.
3. Verific tabelul de hash (`hash`).
4. Golesc tabelul (`hash -r`).
5. Verific din nou (`type comandă`).
6. Dacă problema persistă, verific existența mai multor executabile cu același nume.

Laborator 2.4

Obiectiv

Înțelegerea rolului lui `PATH` și al tabelului de hash.

Durata

30–40 minute.

Cerințe

1. Creează două directoare (`test1` și `test2`) și câte un executabil `demo` în fiecare.
2. Modifică ordinea directoarelor în `PATH` și observă ce se execută.
3. Rulează `hash` și urmărește apariția comenzii `demo`.
4. Schimbă ordinea directoarelor fără să goleşti hash-ul și observă rezultatul.
5. Rulează `hash -r` și repetă testul.
6. Explică diferențele observate.

Soluția laboratorului

- `PATH` stabilește ordinea în care Bash caută executabilele.
- Primul executabil găsit este ales.
- Bash poate memora rezultatul în tabelul de hash pentru a evita căutările repetate.
- `hash -r` obligă Bash să refacă această căutare la următoarea execuție.
- Înțelegerea acestui mecanism este esențială pentru depanarea problemelor legate de executabile multiple sau modificări recente ale mediului.

Ce trebuie să reții

- `PATH` nu este doar o listă de directoare, ci o ordine de prioritate.

- Bash caută executabilele de la stânga la dreapta.
- Tabelul de hash optimizează căutarea comenzilor utilizate frecvent.
- `hash -r` golește acest tabel și forțează o nouă rezolvare a comenzilor.
- `PATH` și `hash` trebuie analizate împreună atunci când investighezi probleme de rezolvare a comenzilor.

Capitolul 3

3.1 Variabilele – Memoria lui Bash

„O variabilă nu este doar un nume asociat unei valori. În Bash, variabilele reprezintă mecanismul prin care shell-ul își amintește informații și le transmite, atunci când este necesar, altor procese.”

Introducere

Imaginează-ți că Bash nu ar avea variabile.

De fiecare dată când ai schimba directorul, ar trebui să reții manual unde te afli.

De fiecare dată când ai executa o comandă, ar trebui să specifici complet toate informațiile necesare.

Nu ar exista:

- utilizatorul curent;
- directorul `HOME`;
- `PATH`;
- istoricul ultimelor directoare;
- codul ultimei comenzi.

Practic, fiecare comandă ar trebui să înceapă de la zero.

Variabilele există tocmai pentru a permite shell-ului să păstreze informații între două comenzi.

Ce este o variabilă?

O variabilă este o asociere între:

- un nume;
- o valoare.

Exemplu:

```
NUME="Daniel"
```

Acum Bash memorează:

NUME

|



"Daniel"

Ori de câte ori folosești:

echo "\$NUME"

Bash înlocuiește expresia \$NUME cu valoarea memorată.

Rezultatul:

Daniel

Observă un detaliu important.

Variabila nu este înlocuită de echo.

Bash face această înlocuire **înainte** ca echo să fie executată.

Acesta este primul contact al cititorului cu ideea de **expansiune**, pe care o vom studia în profunzime mai târziu.

Unde sunt păstrate variabilele?

Aceasta este întrebarea pe care aproape nimeni nu și-o pune.

Mulți își imaginează că variabilele sunt "în Bash", fără să se întrebe ce înseamnă asta.

Realitatea este mai interesantă.

Variabilele sunt păstrate **în memoria procesului Bash**.

Imaginează-ți procesul Bash ca pe un spațiu de lucru care conține mai multe categorii de informații:

+-----+	
	Proces Bash
+-----+	
	Aliasuri
	Funcții
	Variabile Bash
	Variabile exportate
	Istoric
	Opțiuni Bash
	Joburi
+-----+	

Aceste informații există atât timp cât procesul Bash există.

Când procesul se încheie, memoria lui este eliberată, iar aceste informații dispar.

Acesta este motivul pentru care variabilele pe care le definești într-un terminal nu mai există după închiderea lui.

Variabile Bash și variabile de mediu

Acum ajungem la una dintre cele mai importante distincții.

Nu toate variabilele sunt egale.

Există două categorii principale:

Variabile Bash

Sunt cunoscute doar de procesul Bash curent.

Exemplu:

```
NUME="Daniel"
```

Poți verifica existența ei:

```
echo "$NUME"
```

Funcționează.

Dar dacă pornești un proces copil:

```
bash
```

și rulezi:

```
echo "$NUME"
```

rezultatul va fi gol.

De ce?

Pentru că noul proces nu a primit această informație.

Variabile de mediu

Acum:

```
export NUME="Daniel"
```

Se întâmplă ceva nou.

Variabila nu mai este doar o informație internă a shell-ului.

Ea devine parte din **mediul de execuție** (*environment*) transmis proceselor copil.

Acum:

```
bash
```

urmat de:

```
echo "$NUME"
```

va afișa:

Daniel

Acum ai demonstrat practic rolul lui `export`.

De ce există `export`?

Întrebarea este foarte importantă.

De ce nu transmite Bash automat toate variabilele?

Imaginează-ți că un shell conține:

- sute de variabile temporare;
- contoare;
- rezultate intermediare;
- informații folosite doar într-un script.

Dacă toate ar fi copiate în fiecare proces copil, memoria ar fi încărcată inutil, iar mediul ar deveni greu de controlat.

`export` este mecanismul prin care utilizatorul spune:

"Această variabilă este importantă și trebuie transmisă mai departe."

Este o decizie explicită.

Și este o idee foarte elegantă.

Demonstrație

```
VAR1="locală"
```

```
export VAR2="exportată"
```

```
bash
```

În shell-ul copil:

```
echo "$VAR1"
```

```
echo "$VAR2"
```

Rezultatul:

exportată

Prima variabilă nu există.

A doua există.

Acest experiment este suficient pentru a înțelege diferența.

Caseta „În spatele cortinei”

Ce transmite, de fapt, `execve()`?

Ne întoarcem la un concept introdus în primul capitol.

Când Bash execută:

```
execve( ... )
```

unul dintre argumentele apelului de sistem este chiar **lista variabilelor de mediu**.

Aceasta explică de ce procesele copil primesc doar variabilele exportate.

Nu este o regulă arbitrară a lui Bash.

Este modul în care funcționează interfața dintre Bash și kernel.

Aici putem face o trimitere internă:

Vezi Capitolul 1.8 – `fork()` și `execve()`.

Acesta este exact genul de legătură între capitole pe care ne-am propus să o construim.

Greșeală din lumea reală

Un administrator definește:

```
JAVA_HOME=/opt/jdk
```

Apoi pornește:

```
mvn package
```

Maven spune că nu găsește Java.

Problema?

Variabila nu a fost exportată.

Soluția:

```
export JAVA_HOME=/opt/jdk
```

Nu este o problemă Maven.

Nu este o problemă Java.

Este o problemă de mediu de execuție.

Administrator Notes

Ori de câte ori o aplicație spune:

"Nu găsesc această variabilă."

verifică în această ordine:

1. Există variabila?
2. A fost exportată?
3. A fost definită înainte de pornirea aplicației?
4. Rulează aplicația într-un alt proces sau alt utilizator?

În practică, aceste întrebări rezolvă majoritatea problemelor legate de variabilele de mediu.

Cum aş depana această problemă?

Problema:

O aplicație nu vede variabila JAVA_HOME.

Pași:

```
echo "$JAVA_HOME"
```

↓

```
export -p | grep JAVA_HOME
```

↓

```
env | grep JAVA_HOME
```

↓

```
printenv JAVA_HOME
```

↓

Repornește aplicația dacă variabila a fost definită după lansarea ei.

Laborator 3.1

Obiectiv

Înțelegerea diferenței dintre variabilele Bash și variabilele de mediu.

Cerințe

1. Creează o variabilă simplă.

2. Creează o variabilă exportată.
 3. Pornește un nou proces Bash.
 4. Compară rezultatele.
 5. Folosește `env`, `printenv` și `export -p` pentru a observa diferențele.
-

Soluția laboratorului

- Variabilele simple rămân locale shell-ului curent.
 - Variabilele exportate sunt incluse în mediul transmis proceselor copil.
 - `env` și `printenv` afișează mediul de execuție, nu toate variabilele Bash.
 - `export -p` listează variabilele marcate pentru export.
-

Ce trebuie să reții

- Variabilele Bash și variabilele de mediu sunt concepte diferite.
- `export` nu creează o variabilă nouă, ci schimbă modul în care aceasta este transmisă proceselor copil.
- Procesele copil nu văd automat toate variabilele shell-ului părinte.
- Relația dintre `export` și `execve()` explică mecanismul de moștenire a mediului.

Capitolul 3

3.2 Expansiunea variabilelor – Momentul în care Bash începe să „gândească”

„Înainte ca o comandă să fie executată, Bash o transformă. Înlocuiește variabile, extinde caractere speciale și pregătește forma finală pe care o va primi programul. Acest proces poartă numele de expansiune.”

Introducere

Să presupunem că ai definit:

```
NUME="Daniel"
```

și apoi execuți:

```
echo "$NUME"
```

Programul `echo` nu știe ce este `NUME`.

Nu caută variabile.

Nu citește memoria lui Bash.

Totuși afișează:

Daniel

Cum este posibil?

Răspunsul este unul dintre cele mai importante mecanisme ale shell-ului.

Bash modifică linia de comandă înainte ca aceasta să fie executată.

Ce vede Bash?

Utilizatorul scrie:

```
echo "$NUME"
```

Dar aceasta nu este comanda care ajunge la `echo`.

Mai întâi Bash observă simbolul:

```
$
```

Acest caracter îi spune:

„Urmează o expansiune de variabilă.”

Bash caută valoarea variabilei și rescrie intern comanda.

Procesul este:

Utilizator

|

▼

```
echo "$NUME"
```

|

▼

Bash găsește variabila

|

▼

```
echo "Daniel"
```

|

▼

```
execve(...)
```

|

▼

Programul `echo`

Observă un lucru esențial.

`echo` nu vede niciodată textul `$NUME`.

Primește direct:

Daniel

Cum putem demonstra?

Vom folosi o comandă foarte simplă.

```
NUME="Linux"
```

```
printf '<%s>\n' "$NUME"
```

Rezultatul:

```
<Linux>
```

Acum schimbă valoarea:

```
NUME="Bash"
```

```
printf '<%s>\n' "$NUME"
```

Rezultatul devine:

```
<Bash>
```

Programul `printf` nu s-a modificat.

S-a modificat doar textul pe care Bash i l-a transmis.

Expansiunea are loc înaintea execuției

Acesta este unul dintre cele mai importante concepte ale cărții.

Putem reprezenta procesul astfel:

Linia tastată

|

▼

Analiza sintactică (parsing)

|

▼

Expansiuni

|

▼

Redirecționări

|



Execuția comenzii

Vom studia fiecare etapă separat.

Deocamdată este suficient să înțelegem că expansiunea este un pas intermediar între ceea ce tastează utilizatorul și ceea ce execută sistemul.

De ce există acoladele?

Foarte des vei vedea:

```
${NUME}
```

în loc de:

```
$NUME
```

De ce?

Să presupunem că ai:

```
NUME="linux"
```

și scrii:

```
echo "$NUMEscript"
```

Bash caută o variabilă numită:

```
NUMEscript
```

care nu există.

Rezultatul va fi un șir gol.

Forma corectă este:

```
echo "${NUME}script"
```

Acum Bash știe exact unde se termină numele variabilei.

Rezultatul:

```
linuxscript
```

O recomandare de stil

Din punct de vedere sintactic, ambele forme sunt corecte:

```
$NUME
```

și

`${NUME}`

Totuși, în această carte vom recomanda folosirea acoladelor ori de câte ori variabila este lipită de alt text.

Exemplu:

```
LOGFILE="${HOME}/backup.log"
```

Această formă este mai clară, mai ușor de citit și reduce riscul de erori.

Caseta „În spatele cortinei”

De ce Bash face expansiunea?

Pentru că majoritatea programelor Unix nu cunosc sintaxa shell-ului.

Programul `grep` nu știe ce înseamnă `$HOME`.

Nici `cat`.

Nici `cp`.

Nici `tar`.

Ele primesc doar argumente.

Bash este responsabil pentru transformarea comenzilor scrise de utilizator într-o listă de argumente pe care programele o pot înțelege.

Acesta este unul dintre motivele pentru care shell-ul există ca strat între utilizator și kernel.

Greșeală din lumea reală

Un administrator scrie:

```
BACKUP=/backup
```

```
cp raport.txt $BACKUP_vechi
```

Intenția era:

```
/backup_vechi
```

Dar Bash caută variabila:

```
BACKUP_vechi
```

care nu există.

Rezultatul este neașteptat.

Forma corectă este:

```
cp raport.txt "${BACKUP}_vechi"
```

Această eroare apare frecvent în scripturile complexe și este ușor de evitat prin utilizarea consecventă a acoladelor.

Administrator Notes

În scripturile de producție, folosește forma:

```
"${VARIABILA}"
```

chiar și atunci când nu este strict necesar.

Este mai ușor de citit, mai sigură și previne numeroase erori atunci când scripturile evoluează.

Este o convenție adoptată în multe proiecte open-source și recomandată în ghidurile moderne de stil pentru shell scripting.

Cum aş depana această problemă?

Problema:

Variabila pare că nu este înlocuită corect.

Pași:

1. Verific dacă variabila există:

```
declare -p NUME
```

2. Afișez valoarea:

```
printf '<%s>\n' "$NUME"
```

3. Verific dacă numele variabilei este delimitat corect:

```
"${NUME}"
```

4. Activez temporar modul de depanare:

```
set -x
```

și observ cum Bash rescrie comenzile înainte de execuție.

Acesta este unul dintre cele mai utile instrumente pentru depanarea scripturilor și îi vom dedica un capitol separat.

Laborator 3.2

Obiectiv

Înțelegerea mecanismului de expansiune a variabilelor.

Durata

20–30 minute.

Cerințe

1. Creează variabila NUME.
2. Afișează-o folosind `echo` și `printf`.
3. Concatenează text cu și fără acolade.
4. Activează `set -x` și observă comenzile executate.
5. Notează diferențele dintre forma `$VAR` și `${VAR}`.

Soluția laboratorului

- Bash înlocuiește variabilele înainte de lansarea programului.
- `echo` și `printf` primesc deja valoarea expandată.
- Acoladele delimitează explicit numele variabilei și evită ambiguitățile.
- `set -x` este un instrument foarte util pentru observarea comenzilor după expansiune și înainte de execuție.

Ce trebuie să reții

- Expansiunea variabilelor este realizată de Bash, nu de programele externe.
- Simbolul `$` inițiază o expansiune.
- Forma `${VAR}` este recomandată atunci când variabila este urmată de alt text.
- Expansiunea are loc înainte de execuția comenzii.
- `set -x` este unul dintre cele mai valoroase instrumente pentru înțelegerea și depanarea scripturilor Bash.

Capitolul 3

3.3 Marea transformare – Cum rescrie Bash fiecare comandă

„Comanda pe care o tastezi și comanda pe care o execută kernelul nu sunt întotdeauna aceeași. Între ele există un proces de transformare realizat de Bash.”

Introducere

Să presupunem că tastezi:

```
cp ~/Documente/${NUME}/*.txt "$HOME/arhiva/"
```

Pare o comandă simplă.

În realitate, Bash vede această linie ca pe un șir de elemente care trebuie interpretate și transformate.

Înainte ca programul `cp` să fie lansat, Bash trebuie să răspundă la mai multe întrebări:

- Ce înseamnă `~`?
- Ce valoare are `${NUME}`?
- La ce fișiere se referă `*.txt`?
- Ce reprezintă `$HOME`?

Abia după ce toate aceste întrebări au primit răspuns, comanda poate fi executată.

Bash este un traducător

Îmi place să privesc Bash ca pe un traducător simultan.

Utilizatorul vorbește într-un limbaj convenabil pentru oameni.

Kernelul și programele externe înțeleg doar argumente concrete.

Bash face traducerea dintre cele două lumi.

De exemplu:

Utilizatorul scrie:

```
echo ~
```

Bash transformă intern în:

```
echo /home/daniel
```

Programul `echo` nu știe ce este `~`.

Primește doar șirul de caractere:

```
/home/daniel
```

Un exemplu mai complex

Presupunem:

```
NUME="proiect"
```

și directorul conține:

```
raport1.txt
```

```
raport2.txt
```

```
raport3.txt
```

Utilizatorul scrie:

```
ls ${NUME}*.txt
```

Dacă variabila conține:

```
raport
```

Bash transformă intern:

```
ls raport*.txt
```

Apoi extinde caracterul `*` și obține:

```
ls raport1.txt raport2.txt raport3.txt
```

Aceasta este comanda pe care o primește `ls`.

Nu cea tastată de utilizator.

Etapele transformării

În capitolele următoare vom studia fiecare etapă separat.

Pentru moment este suficient să cunoaștem imaginea de ansamblu.

În mod simplificat, Bash procesează o comandă astfel:

Linia tastată

|

▼

Analiza sintactică (parsing)

|

▼

Expansiuni

|

▼

Eliminarea ghilimelelor (quote removal)

|

▼

Redirecționări

|

▼

Rezolvarea comenzii

|

▼

Execuția

Notă: În capitolele dedicate vom rafina această schemă și vom prezenta ordinea completă a expansiunilor exact așa cum este descrisă în documentația Bash. Deocamdată folosim o versiune simplificată pentru a construi modelul mental al cititorului.

De ce este importantă ordinea?

Să privim exemplul:

```
echo "$HOME/*.*txt"
```

Mulți utilizatori se așteaptă ca Bash să afișeze toate fișierele `.*txt`.

Dar nu se întâmplă.

De ce?

Pentru că ghilimelele modifică regulile de expansiune.

Acesta este un subiect suficient de important încât îi vom dedica mai multe capitole.

Deocamdată este suficient să reținem că **ordinea și contextul contează**.

Caseta „În spatele cortinei”

Bash nu interpretează totul deodată

Aceasta este o idee foarte importantă.

Bash nu citește o linie și o execută instantaneu.

El parcurge mai multe faze succesive.

Fiecare fază are o responsabilitate clară.

Acest mod de proiectare face codul Bash mai ușor de întreținut și explică de ce unele combinații de caractere speciale produc efecte diferite în funcție de context.

Înțelegerea acestor faze este cheia pentru scrierea scripturilor robuste.

Greșeală din lumea reală

Un administrator creează un director:

```
Backup iulie
```

Apoi scrie:

```
cp $DIR/*.sql /backup
```

unde:

```
DIR="Backup iulie"
```

Rezultatul este neașteptat deoarece valoarea variabilei conține un spațiu.

Forma recomandată este:

```
cp "${DIR}"/* .sql /backup
```

Vom dedica un capitol întreg acestui subiect atunci când vom studia quoting și împărțirea în cuvinte (*word splitting*).

Administrator Notes

Ori de câte ori o comandă produce un rezultat surprinzător, pune-ți întrebarea:

„Ce comandă a ajuns, de fapt, la program?”

De foarte multe ori problema nu este în `cp`, `grep` sau `find`.

Problema este că Bash a transformat linia de comandă într-un mod diferit de cel așteptat.

Această întrebare simplă schimbă complet modul de depanare.

Cum aş depana această problemă?

Problema:

Comanda pare corectă, dar programul primește argumente greșite.

Pași:

1. Afișez valorile variabilelor implicate.
2. Verific folosirea ghilimelelor.
3. Activez:

```
set -x
```

și observ forma finală a comenzii.

4. Simplific expresia până când identific transformarea care produce efectul neașteptat.
-

Laborator 3.3

Obiectiv

Observarea transformărilor realizate de Bash înainte de execuția unei comenzi.

Durata

30 de minute.

Cerințe

1. Creează un director cu fișiere `.txt`.
 2. Definește o variabilă care conține numele directorului.
 3. Rulează comenzi folosind:
 - `~`
 - `${VAR}`
 - `*`
 4. Activează `set -x`.
 5. Compară comanda tastată cu cea executată efectiv.
-

Soluția laboratorului

- Bash efectuează mai multe tipuri de expansiuni înainte de lansarea programului.
 - Programul extern primește argumentele deja transformate.
 - `set -x` este un instrument excelent pentru observarea rezultatului acestor transformări.
 - O mare parte dintre erorile din scripturi provin din interpretarea greșită a acestor etape, nu din funcționarea programelor externe.
-

Ce trebuie să reții

- Bash transformă linia de comandă înainte de execuție.
- Expansiunile fac parte dintr-un proces mai amplu de procesare.
- Ordinea acestor etape influențează rezultatul final.
- Programele externe nu interpretează sintaxa Bash; ele primesc argumentele deja pregătite.

Capitolul 3

3.4 Ordinea oficială a expansiunilor în Bash

„Atunci când o linie de comandă conține mai multe tipuri de expansiuni, Bash nu le execută aleatoriu. Există o ordine precisă, definită de implementare și documentată în manualul oficial. Înțelegerea acestei ordini explică multe comportamente aparent surprinzătoare.”

Introducere

Să analizăm următoarea comandă:

```
echo ~/Proiecte/${NUME}{1,2}*.txt
```

La prima vedere pare doar o combinație de caractere speciale.

În realitate, Bash trebuie să rezolve mai multe tipuri de expansiuni:

- `{1,2}`
- `~`
- `${NUME}`
- `*`

Întrebarea este:

Pe care o rezolvă prima?

Răspunsul este foarte important, deoarece rezultatul final depinde de această ordine.

Ordinea oficială

Conform documentației Bash, expansiunile au loc, în linii mari, în următoarea ordine:

1. **Brace Expansion**
2. **Tilde Expansion**
3. **Parameter Expansion**
4. **Command Substitution**
5. **Arithmetic Expansion**
6. **Word Splitting**
7. **Pathname Expansion (Globbing)**
8. **Quote Removal**

Aceasta este ordinea pe care o vom folosi și în întreaga carte.

De ce contează?

Să presupunem:

```
NUME="raport"
```

și executăm:

```
echo ${NUME}{1,2}
```

Rezultatul este:

raport1 raport2

Ce s-a întâmplat?

Mai întâi:

`${NUME}{1,2}`

devine:

`${NUME}1`

`${NUME}2`

(abia aici intervine **Brace Expansion**)

Apoi fiecare expresie trece prin **Parameter Expansion**:

raport1

raport2

Acest exemplu arată că fiecare etapă pregătește terenul pentru următoarea.

O analogie inginerescă

Imaginează-ți o linie de asamblare într-o fabrică.

Prima stație montează șasiul.

A doua instalează motorul.

A treia montează instalația electrică.

Dacă ai încerca să montezi instalația electrică înainte de șasiu, procesul nu ar mai avea sens.

Expansiunile Bash funcționează la fel.

Ordinea nu este arbitrară.

Ea este parte din proiectarea shell-ului.

Quote Removal – de ce este ultima?

Aceasta este o întrebare excelentă.

Mulți presupun că Bash elimină ghilimelele imediat.

Nu.

Le păstrează până aproape de final.

De ce?

Pentru că ghilimelele influențează etapele anterioare.

Exemplu:

```
echo "*.txt"
```

Asteriscul nu este expandat.

De ce?

Pentru că ghilimelele îi spun lui Bash:

„Tratează acest text ca pe un șir literal.”

Abia după ce toate celelalte etape s-au încheiat, ghilimelele sunt eliminate.

Programul `echo` nu primește ghilimelele, ci doar textul dintre ele.

Caseta „În spatele cortinei”

Brace Expansion este o facilitare a shell-ului

Este important să subliniem un detaliu.

Construcția:

```
file{1,2,3}.txt
```

nu este înțeleasă de kernel.

Nici de `cp`.

Nici de `ls`.

Este o funcționalitate implementată exclusiv în Bash (și în alte shell-uri compatibile).

Acest lucru explică de ce aceeași comandă poate avea rezultate diferite în shell-uri diferite.

Greșeală din lumea reală

Un administrator scrie:

```
echo "$HOME"/*.log
```

și obține lista fișierelor.

Apoi schimbă comanda în:

```
echo "$HOME/*.log"
```

și nu mai primește aceeași listă.

Nu s-a schimbat `echo`.

S-au schimbat regulile de expansiune.

Acest exemplu este un motiv excelent pentru care vom dedica un capitol separat mecanismului de quoting.

Administrator Notes

Nu încerca să memorezi această ordine ca pe o poezie.

Înțelege responsabilitatea fiecărei etape.

Când vei depana un script, vei putea spune:

„Problema apare înainte de *word splitting*.”

sau

„Aici globbing-ul nu mai are loc deoarece expresia este între ghilimele.”

Acesta este nivelul de înțelegere pe care ni-l dorim.

Cum aş depana această problemă?

Problema:

O comandă produce un rezultat diferit de cel așteptat.

Metodă:

1. Identific toate tipurile de expansiuni prezente în comandă.
2. Analizez ordinea în care Bash le aplică.
3. Verific dacă quoting-ul modifică vreuna dintre etape.
4. Folosesc:

`set -x`

pentru a observa forma finală a comenzii.

Laborator 3.4

Obiectiv

Observarea ordinii expansiunilor.

Cerințe

Execută și explică rezultatul următoarelor comenzi:

```
echo file{1,2}.txt
```

```
NUME=test  
echo ${NUME}{1,2}
```

```
touch a.txt b.txt  
echo *.txt
```

```
echo "*.txt"
```

```
echo ~
```

```
echo "~"
```

Pentru fiecare exemplu, răspunde:

- Ce tip de expansiune apare?
- În ce etapă?
- De ce rezultatul este acesta?

Soluția laboratorului

echo file{1,2}.txt

Brace Expansion

Rezultat:

```
file1.txt file2.txt
```

echo \${NUME}{1,2}

Mai întâi are loc **Brace Expansion**, apoi **Parameter Expansion**, rezultând:

```
test1 test2
```

echo *.txt

Are loc **Pathname Expansion**, iar *.txt este înlocuit cu lista fișierelor care corespund modelului.

echo "*.txt"

Nu are loc **Pathname Expansion** deoarece modelul este protejat de ghilimele.

Rezultatul este:

*.txt

echo ~

Are loc **Tilde Expansion**, iar ~ este înlocuit cu directorul HOME al utilizatorului.

echo "~"

Nu are loc **Tilde Expansion** deoarece caracterul ~ este între ghilimele și este tratat ca text literal.

Ce trebuie să reții

- Bash aplică expansiunile într-o ordine bine definită.
- Ordinea este parte din proiectarea shell-ului și influențează rezultatul final.
- Quote Removal este ultima etapă.
- Ghilimelele nu sunt doar caractere decorative; ele modifică modul în care Bash interpretează comanda.
- Înțelegerea ordinii expansiunilor este esențială pentru scrierea și depanarea scripturilor Bash.

Capitolul 3

3.5 Quoting – Cum îi spui lui Bash să nu interpreteze ceea ce scrii

„Caracterele speciale fac Bash puternic. Ghilimelele îți permit să controlezi această putere și să decizi ce trebuie interpretat și ce trebuie tratat ca text literal.”

Introducere

Până acum am învățat că Bash interpretează multe caractere speciale.

De exemplu:

- \$ inițiază expansiunea unei variabile;
- * reprezintă un model de căutare;
- ~ este înlocuit cu directorul HOME;
- {} pot genera mai multe șiruri de caractere.

Dar apare imediat o întrebare firească.

Ce fac dacă vreau să folosesc aceste caractere ca text obișnuit?

Cum îi spun lui Bash:

"Nu interpreta. Afișează exact ceea ce am scris."

Răspunsul este:

quoting.

Ce este quoting?

Quoting reprezintă mecanismul prin care utilizatorul controlează interpretarea liniei de comandă.

Cu alte cuvinte, îi spune lui Bash:

- acest text poate fi interpretat;
- acest text trebuie lăsat neschimbat.

Fără quoting, aproape orice comandă complexă ar deveni dificil de scris și de înțeles.

Cele trei mecanisme principale

În Bash există trei metode utilizate frecvent pentru quoting:

- caracterul backslash (\);
- ghilimelele simple (' . . .');
- ghilimelele duble (" . . .").

Vom studia fiecare dintre ele separat.

1. Escape Character (\)

Backslash-ul protejează caracterul imediat următor.

Exemplu:

```
echo \$HOME
```

Rezultatul:

```
$HOME
```

De ce?

Pentru că semnul \$ nu mai este interpretat ca început al unei expansiuni.

Același lucru este valabil pentru:

```
echo \*
```

Rezultatul:

```
*
```

și nu lista fișierelor din director.

Când este util?

Atunci când vrem să protejăm **un singur caracter**.

De exemplu:

```
mkdir Director\ Nou
```

crează un director numit:

```
Director Nou
```

fără să fie nevoie de ghilimele.

2. Ghilimelele simple

Acestea sunt cea mai puternică formă de quoting.

Tot ceea ce se află între ele este tratat ca text literal.

Exemplu:

```
echo '$HOME'
```

Rezultatul:

\$HOME

Nu există:

- expansiune de variabile;
- expansiune ~;
- substituție de comenzi;
- expansiune aritmetică.

Bash tratează conținutul exact așa cum a fost scris.

Demonstrație

Presupunem:

```
NUME="Daniel"
```

Executăm:

```
echo '$NUME'
```

Rezultatul:

```
$NUME
```

Nu:

```
Daniel
```

Acesta este comportamentul dorit.

3. Ghilimelele duble

Acum lucrurile devin mai interesante.

Ghilimelele duble protejează textul doar parțial.

Exemplu:

```
echo "$NUME"
```

Rezultatul:

```
Daniel
```

Observă diferența.

Variabila este expandată.

În schimb:

```
echo "*.txt"
```

afișează:

```
*.txt
```

Nu lista fișierelor.

De ce?

Pentru că în interiorul ghilimelelor duble **globbing-ul** este dezactivat.

Tabel comparativ

Expansiune	Fără quoting	' . . . '	" . . . "
Variabile (\$VAR)	✓	✗	✓
Globbering (*)	✓	✗	✗
Spații	separator	păstrate	păstrate
Command substitution (\$(. . .))	✓	✗	✓
Arithmetic expansion (\${ (. . .) })	✓	✗	✓

Acest tabel merită reținut. Vom reveni la el de mai multe ori în capitolele următoare.

De ce sunt importante ghilimelele?

Să presupunem:

```
DIR="Backup iulie"
```

și executăm:

```
cp $DIR/raport.txt /backup
```

Bash vede:

```
cp Backup iulie/raport.txt /backup
```

Observă problema.

Spațiul desparte argumentele.

Programul `cp` primește:

Argumentul 1:

Backup

Argumentul 2:

iulie/raport.txt

Rezultatul este o eroare.

Forma corectă este:

```
cp "$DIR/raport.txt" /backup
```

Acum Bash transmite un singur argument:

```
Backup iulie/raport.txt
```

Acesta este unul dintre cele mai importante motive pentru care recomandăm folosirea consecventă a ghilimelelor duble în jurul variabilelor.

Caseta „În spatele cortinei”

De ce recomandăm "\${VAR}"?

Poate ai observat că în toate exemplele folosim forma:

```
"${VAR}"
```

și nu:

```
$VAR
```

Nu este doar o preferință stilistică.

Este o măsură de siguranță.

Ea previne probleme cauzate de:

- spații;
- caractere speciale;
- împărțirea în cuvinte (*word splitting*);
- interpretarea accidentală a unor modele.

În multe proiecte open-source, aceasta este regula implicită de scriere a scripturilor Bash.

Greșeală din lumea reală

Un administrator scrie:

```
rm -rf $DIR/*
```

unde:

```
DIR=""
```

din cauza unei erori anterioare.

Comanda devine:

```
rm -rf /*
```

Consecințele pot fi dezastruoase.

Acesta este unul dintre motivele pentru care scripturile robuste validează întotdeauna variabilele înainte de a le utiliza și folosesc quoting în mod consecvent.

Administrator Notes

Există o recomandare pe care merită să o transformăm într-o regulă de lucru:

Dacă ai un motiv întemeiat să nu pui o variabilă între ghilimele, explică acel motiv în comentariul scriptului. În toate celelalte situații, folosește ghilimele duble.

Această disciplină reduce foarte multe erori și face intenția autorului clară pentru cei care vor întreține scriptul peste ani.

Cum aş depana această problemă?

Problema:

Scriptul funcționează pentru unele directoare, dar eșuează pentru cele care conțin spații.

Pași:

1. Afișez valoarea variabilei:

```
printf '<%s>\n' "$DIR"
```

2. Activez:

```
set -x
```

3. Observ argumentele transmise către comandă.
 4. Verific dacă toate variabilele sunt între ghilimele.
 5. Repet testul cu un director care conține spații.
-

Laborator 3.5

Obiectiv

Înțelegerea diferențelor dintre cele trei mecanisme de quoting.

Cerințe

1. Creează variabila:

```
NUME="Linux Bash"
```

2. Rulează:

```
echo $NUME
echo "$NUME"
echo '$NUME'
```

3. Creează fișiere `.txt` și compară:

```
echo *.txt
echo "*.txt"
echo '*.txt'
```

4. Creează un director cu spații în nume și testează comenzile `cp` și `ls` cu și fără quoting.
 5. Explică fiecare rezultat.
-

Soluția laboratorului

- Backslash-ul protejează caracterul imediat următor.
 - Ghilimelele simple dezactivează aproape toate expansiunile.
 - Ghilimelele duble permit expansiunea variabilelor și a substituțiilor de comenzi, dar protejează textul împotriva împărțirii în cuvinte și a globbing-ului.
 - Folosirea consecventă a ghilimelelor duble în jurul variabilelor este una dintre cele mai bune practici în scripting.
-

Ce trebuie să reții

- Quoting controlează modul în care Bash interpretează linia de comandă.
- `\`, `' . . . '` și `" . . . "` au scopuri și comportamente diferite.
- Ghilimelele duble reprezintă alegerea implicită recomandată în majoritatea scripturilor.
- Multe erori aparent misterioase sunt cauzate de lipsa quoting-ului.

Capitolul 3

3.6 Word Splitting – Mecanismul invizibil care produce cele mai multe erori în Bash

*„Foarte multe scripturi nu eșuează din cauza comenzilor folosite, ci din cauza modului în care Bash împarte textul în argumente. Acest mecanism, numit **Word Splitting**, este una dintre cele mai importante surse de erori în scripting.”*

Introducere

Să presupunem că avem variabila:

```
NUME="Linux Bash"
```

și executăm:

```
echo $NUME
```

Rezultatul este:

```
Linux Bash
```

Pare perfect.

Dar întrebarea este:

Câte argumente a primit echo?

Unul?

Două?

Această întrebare este cheia întregului capitol.

Ce este Word Splitting?

După ce Bash termină expansiunea variabilelor, urmează o etapă foarte importantă.

Dacă rezultatul unei expansiuni nu este protejat de quoting, Bash îl poate împărți în mai multe cuvinte.

Acest proces se numește **Word Splitting**.

Este controlat, în principal, de variabila specială:

```
IFS
```

(Internal Field Separator)

Primul experiment

Definim:

```
NUME="Linux Bash"
```

Acum rulăm:

```
printf '<%s>\n' $NUME
```

Rezultatul este:

```
<Linux>
<Bash>
```

Observă ce s-a întâmplat.

Expansiunea a produs şirul:

```
Linux Bash
```

Dar, pentru că variabila nu era între ghilimele, Bash l-a împărţit în două argumente.

Programul `printf` nu a făcut această împărţire.

Bash a făcut-o înainte de execuţie.

Acelaşi experiment cu quoting

Acum executăm:

```
printf '<%s>\n' "$NUME"
```

Rezultatul este:

```
<Linux Bash>
```

De această dată există un singur argument.

Ghilimelele au împiedicat Word Splitting.

Aceasta este una dintre cele mai importante idei din întregul Bash.

De ce se întâmplă asta?

Pentru că Bash trebuie să transforme textul într-o listă de argumente.

Dacă utilizatorul scrie:

```
cp fisier1 fisier2 backup/
```

Bash trebuie să identifice:

Argumentul 1:

```
fișier1
```

Argumentul 2:

```
fișier2
```

Argumentul 3:

```
backup/
```

Word Splitting este mecanismul prin care Bash separă aceste argumente atunci când este cazul.

Rolul lui IFS

Variabila:

```
echo "$IFS"
```

poate părea goală atunci când este afișată, deoarece conține caractere speciale (spațiu, tab și linie nouă).

Pentru a vedea valoarea ei într-un mod mai clar, putem folosi:

```
printf '%q\n' "$IFS"
```

sau:

```
declare -p IFS
```

În configurația implicită, IFS conține:

- spațiu;
- tab;
- caracterul de sfârșit de linie.

Aceste caractere sunt folosite de Bash pentru a separa cuvintele rezultate după expansiune.

Notă tehnică: În capitolul dedicat lui IFS vom explica în detaliu toate particularitățile sale și efectele modificării acestei variabile.

Demonstrație

Definim:

```
LISTA="unu doi trei"
```

Acum:

```
for x in $LISTA
do
    printf "[%s]\n" "$x"
done
```

Rezultatul:

```
[unu]
[doi]
[trei]
```

Același exemplu:

```
for x in "$LISTA"
do
    printf "[%s]\n" "$x"
done
```

produce:

```
[unu doi trei]
```

Observă cât de mult schimbă comportamentul doar două ghilimele.

Un exemplu din producție

Presupunem că administratorul dorește să arhiveze un director:

```
DIR="/home/daniel/Proiect Linux"
```

și scrie:

```
tar -czf backup.tar.gz $DIR
```

După expansiune:

```
tar -czf backup.tar.gz /home/daniel/Proiect Linux
```

Bash vede două argumente:

```
/home/daniel/Proiect
```

și

```
Linux
```

`tar` nu mai poate interpreta corect comanda.

Forma corectă este:

```
tar -czf backup.tar.gz "$DIR"
```

Caseta „În spatele cortinei”

De ce Word Splitting este separat de Parameter Expansion?

Aceasta este o întrebare foarte interesantă.

Parameter Expansion produce un șir de caractere.

Word Splitting decide dacă acel șir trebuie tratat ca un singur argument sau împărțit în mai multe.

Separarea acestor etape oferă Bash o flexibilitate foarte mare și permite quoting-ului să controleze comportamentul fără a modifica valoarea variabilei.

Greșeală din lumea reală

Un script conține:

```
for FILE in $(find . -name "*.txt")
do
    cp $FILE backup/
done
```

Scriptul funcționează perfect...

până când apare un fișier numit:

```
Raport final.txt
```

Din acel moment, Word Splitting împarte numele în două cuvinte și scriptul începe să dea erori.

Notă: aici nu vom explica încă soluția completă (`find -print0`, `read -d ''`, `xargs -0` etc.). Vom face doar o trimitere înainte. Vom dedica un capitol întreg procesării sigure a numelor de fișiere.

Administrator Notes

În foarte multe situații, regula practică este simplă:

Dacă expandezi o variabilă și intenția este să reprezinte un singur argument, pune-o între ghilimele.

Există excepții.

Le vom studia.

Dar această regulă previne majoritatea erorilor întâlnite în scripturile Bash.

Cum aş depana această problemă?

Problema:

Scriptul funcţionează pentru majoritatea fişierelor, dar eşuează atunci când apar spaţii în nume.

Paşi:

1. Activez:

```
set -x
```

2. Afişez argumentele folosind:

```
printf '<%s>\n' "$@"
```

(sau alte tehnici potrivite contextului).

3. Verific fiecare expansiune de variabilă.
 4. Identific variabilele care nu sunt între ghilimele.
 5. Testez din nou folosind directoare şi fişiere cu spaţii în nume.
-

Laborator 3.6

Obiectiv

Înţelegerea mecanismului Word Splitting.

Cerinţe

1. Creează variabila:

```
TEXT="unu doi trei"
```

2. Compară rezultatele:

```
printf '<%s>\n' $TEXT
```

şi

```
printf '<%s>\n' "$TEXT"
```

3. Creează un director care conține spații în nume.
 4. Folosește acel director într-un script cu și fără quoting.
 5. Explică diferențele observate.
-

Soluția laboratorului

Întrebarea 1

De ce primul exemplu afișează trei linii?

Pentru că după Parameter Expansion, Bash aplică Word Splitting și transformă șirul în trei argumente distincte.

Întrebarea 2

De ce al doilea exemplu afișează o singură linie?

Pentru că ghilimelele dezactivează Word Splitting asupra rezultatului expansiunii.

Întrebarea 3

De ce apar erori la directoarele cu spații?

Pentru că Bash împarte numele directorului în mai multe argumente atunci când variabila nu este protejată prin quoting.

Ce trebuie să reții

- Word Splitting este o etapă separată de Parameter Expansion.
- Ea are loc după expansiunea variabilelor și înainte de Pathname Expansion.
- Ghilimelele duble împiedică Word Splitting.
- Lipsa quoting-ului este una dintre cele mai frecvente cauze ale erorilor din scripturile Bash.
- IFS controlează modul implicit de separare a cuvintelor.

Capitolul 3

3.7 Pathname Expansion (Globbing) – Cum găsește Bash fișierele fără să le numești

„Atunci când scrii `*.txt`, Bash nu caută un fișier cu acest nume. El construiește o listă cu toate fișierele care se potrivesc modelului și transmite această listă programului care urmează să fie executat.”

Introducere

Mulți utilizatori cred că programe precum `ls`, `cp` sau `rm` înțeleg caractere precum:

```
*  
?  
[]
```

În realitate, în majoritatea situațiilor, ele nu le înțeleg deloc.

Ele primesc deja lista fișierelor.

Cel care face căutarea este Bash.

Această idee este fundamentală.

Ce este globbing?

Globbering este denumirea folosită pentru mecanismul prin care Bash transformă un model într-o listă de nume de fișiere.

Exemplu:

Utilizatorul scrie:

```
ls *.txt
```

Presupunem că directorul conține:

```
note.txt  
raport.txt  
backup.tar
```

Înainte de execuție, Bash transformă comanda în:

```
ls note.txt raport.txt
```

Programul `ls` nu primește niciodată:

```
*.txt
```

Primește:

```
note.txt  
raport.txt
```

Demonstrație

Vom demonstra folosind `printf`, deoarece afișează fiecare argument separat.

Creează câteva fișiere:

```
touch a.txt b.txt c.log
```

Acum rulează:

```
printf '<%s>\n' *.txt
```

Rezultatul:

```
<a.txt>
```

```
<b.txt>
```

Observă.

`printf` nu știe ce este `*.txt`.

Primește deja două argumente.

Acesta este unul dintre motivele pentru care preferăm `printf` în laboratoare. El ne permite să vedem exact câte argumente au fost generate de Bash.

Caracterul *

Caracterul:

*

înseamnă:

orice secvență de caractere (inclusiv secvența vidă).

Exemple:

*

→ toate fișierele (cu excepția celor ascunse, în configurația implicită).

`*.txt`

→ toate fișierele care se termină în `.txt`.

`backup*`

→ toate fișierele care încep cu `backup`.

Caracterul ?

Caracterul:

?

înseamnă:

exact un caracter.

Exemplu:

a?.txt

se potrivește cu:

ab.txt

a1.txt

ax.txt

dar nu cu:

a.txt

abc.txt

Clase de caractere

Putem restrânge căutarea folosind:

[]

Exemplu:

capitol[123].txt

se potrivește cu:

capitol1.txt

capitol2.txt

capitol3.txt

dar nu și cu:

capitol4.txt

Putem folosi și intervale:

[a-z]

sau

[0-9]

Aceste expresii sunt foarte utile în scripturi și în administrarea fișierelor.

Ce se întâmplă dacă nu există nicio potrivire?

Aici apare un detaliu foarte important.

Presupunem că rulăm:

```
ls *.pdf
```

într-un director care nu conține fișiere PDF.

În configurația implicită a Bash, modelul **nu este eliminat**.

Comanda transmisă va fi:

```
ls "*.pdf"
```

Iar `ls` va încerca să găsească un fișier cu numele literal `*.pdf`, afișând de obicei:

```
ls: cannot access '*.pdf': No such file or directory
```

Acesta este comportamentul implicit.

Mai târziu vom studia opțiunile `nullglob`, `failglob` și `nocaseglob`, care pot modifica acest comportament.

Globbering și quoting

Acum apare o întrebare foarte importantă.

Ce se întâmplă dacă scriem:

```
echo "*.txt"
```

Rezultatul este:

```
*.txt
```

Nu lista fișierelor.

De ce?

Pentru că ghilimelele dezactivează etapa de **Pathname Expansion**.

Aceasta este una dintre cele mai importante relații dintre quoting și globbing.

Caseta „În spatele cortinei”

De ce programele nu implementează ele însele globbing-ul?

Pentru că acesta este rolul shell-ului.

Dacă fiecare program ar implementa propriul mecanism de interpretare pentru *, ? și [], comportamentul ar fi diferit de la un utilitar la altul.

Unix a ales o abordare elegantă:

Shell-ul rezolvă modelele.

Programele primesc doar lista finală de argumente.

Această separare simplifică implementarea utilităților și asigură un comportament consecvent.

Greșeală din lumea reală

Un administrator scrie:

```
scp *.log server:/backup/
```

și observă că uneori sunt copiate toate fișierele, iar alteori comanda eșuează.

Explicația este simplă.

Modelul *.log este expandat de shell-ul local înainte ca scp să fie executat.

Dacă nu există potriviri, scp primește argumentul literal *.log.

Acest comportament explică multe situații aparent ciudate atunci când lucrăm cu utilitare de rețea.

Administrator Notes

Nu presupune niciodată că un program "înțelege" caracterele * sau ?.

În cele mai multe cazuri, shell-ul a rezolvat deja aceste modele înainte ca programul să înceapă execuția.

Atunci când depanezi o comandă, întreabă-te întotdeauna:

Ce argumente a primit programul?

Nu:

Ce am tastat eu?

Aceasta este diferența dintre perspectiva utilizatorului și cea a sistemului.

Cum aş depana această problemă?

Problema:

*.log nu produce rezultatul aşteptat.

Paşii:

1. Verific dacă există fişiere care se potrivesc modelului:

```
printf '<%s>\n' *.log
```

2. Verific dacă modelul este între ghilimele.
3. Verific opţiunile relevante:

```
shopt | grep glob
```

4. Testez comportamentul cu şi fără quoting.
 5. Verific dacă shell-ul este într-adevăr Bash sau un alt shell cu reguli diferite.
-

Laborator 3.7

Obiectiv

Înţelegerea mecanismului de Pathname Expansion.

Cerinţe

1. Creează fişiere:

```
touch a.txt  
touch b.txt  
touch c.log  
touch capitol1.txt  
touch capitol2.txt  
touch capitol3.txt
```

2. Testează:

```
printf '<%s>\n' *.txt  
  
printf '<%s>\n' a?.txt  
  
printf '<%s>\n' capitol[12].txt  
  
printf '<%s>\n' "*.txt"
```

3. Explică fiecare rezultat.

Soluția laboratorului

Exercițiul 1

`*.txt`

→ Bash generează lista tuturor fișierelor `.txt`.

Exercițiul 2

`a?.txt`

→ Se potrivesc doar numele care conțin exact un caracter între `a` și `.txt`.

Exercițiul 3

`capitol[12].txt`

→ Sunt selectate doar `capitol1.txt` și `capitol2.txt`.

Exercițiul 4

`"*.txt"`

→ Nu are loc globbing deoarece modelul este protejat de ghilimele.

Testează-te

Ce va afișa următoarea comandă?

Presupunem că directorul conține:

`a.txt`
`b.txt`
`c.log`

Comanda este:

```
printf '<%s\n' "*.txt"
```

A)

`<a.txt>`
`<b.txt>`

B)

`<*.txt>`

C)

Eroare.

Răspuns

Varianta B.

Explicație:

Ghilimelele duble împiedică Pathname Expansion. `printf` primește un singur argument, și anume șirul literal `*.txt`.

Ce trebuie să reții

- Pathname Expansion este realizată de Bash, nu de programele externe.
- `*`, `?` și `[]` sunt modele, nu expresii regulate.
- Ghilimelele împiedică globbing-ul.
- În lipsa unei potriviri, comportamentul implicit este păstrarea modelului neschimbat.
- Pentru a înțelege o comandă, trebuie să știi ce argumente au fost generate după Pathname Expansion.

Capitolul 3

3.8 Command Substitution – Cum folosește Bash rezultatul unei comenzi ca argument pentru alta

*„În Bash, ieșirea unei comenzi poate deveni intrarea altei comenzi înainte ca aceasta să fie executată. Acest mecanism se numește **Command Substitution** și reprezintă una dintre cele mai elegante caracteristici ale shell-ului.”*

Introducere

Până acum am văzut că Bash poate înlocui:

- variabile;
- `~`;
- modele de fișiere.

Dar apare o întrebare firească.

Poate Bash să înlocuiască și rezultatul unei comenzi?

Răspunsul este da.

Și aceasta este una dintre cele mai puternice facilități ale shell-ului.

Primul exemplu

Executăm:

```
echo "$(date)"
```

Rezultatul poate fi:

```
Mon Aug 3 10:15:42 EEST 2026
```

Întrebarea este:

Primește `echo` expresia:

```
$(date)
```

?

Nu.

Bash execută mai întâi:

```
date
```

Colectează ieșirea standard (*standard output*), elimină caracterul final de linie nouă și apoi înlocuiește expresia `$(date)` cu textul rezultat.

Intern, procesul poate fi imaginat astfel:

```
echo "$(date)"
  ↓
rulează comanda date
  ↓
obține textul rezultat
  ↓
echo "Mon Aug 3 10:15:42 EEST 2026"
  ↓
execve(...)
```

Observă din nou modelul pe care îl întâlnim în tot Bash-ul.

Programul `echo` nu știe nimic despre `$(...)`.

El primește doar argumentul deja pregătit de shell.

De ce există Command Substitution?

Să presupunem că vrem să creăm un director care conține data curentă.

Fără Command Substitution, ar trebui:

1. să rulăm `date`;
2. să copiem rezultatul;
3. să îl introducem manual în comanda `mkdir`.

Command Substitution permite automatizarea acestui proces.

Exemplu:

```
mkdir "backup-$(date +%F)"
```

Dacă data este `2026-08-04`, Bash transformă intern comanda în:

```
mkdir "backup-2026-08-04"
```

Forma recomandată

În trecut era folosită frecvent sintaxa:

```
`date`
```

(backticks)

Ea funcționează și astăzi, însă recomandarea modernă este:

```
$(date)
```

De ce?

Pentru că este:

- mai ușor de citit;
- mai ușor de imbricat;
- mai clară vizual;
- mai puțin predispusă la erori.

În această carte vom folosi exclusiv sintaxa `$(...)`, iar atunci când întâlnim backticks în documentații sau scripturi vechi, le vom explica și vom arăta echivalentul modern.

Demonstrație

Executăm:

```
printf '<%s>\n' "$(hostname)"
```

Rezultatul poate fi:

```
<server01>
```

Acum:

```
printf '<%s>\n' "$(pwd)"
```

Rezultatul:

```
</home/daniel>
```

Observă că `printf` nu execută `hostname` și nici `pwd`.

El primește doar textul rezultat.

Command Substitution și quoting

Acesta este un punct foarte important.

Presupunem că executăm:

```
FILES=$(ls)
```

și apoi:

```
printf '<%s>\n' $FILES
```

Dacă rezultatul conține spații sau linii noi, Bash poate aplica **Word Splitting** și poate împărți textul în mai multe argumente.

În schimb:

```
printf '<%s>\n' "$FILES"
```

tratează rezultatul ca un singur argument.

Notă: este important să subliniem că folosirea `ls` pentru procesarea numelor de fișiere este, în general, o practică nerecomandată în scripturi. Vom dedica un capitol separat acestui subiect și vom prezenta alternative precum `find`, `printf '%s\0'` și citirea sigură a numelor de fișiere.

Command Substitution și codul de ieșire

Aici merită făcută o precizare importantă.

Command Substitution înlocuiește **ieșirea standard** (*stdout*) a comenzii.

Nu transmite automat și codul de ieșire (*exit status*).

Dacă avem nevoie să verificăm succesul unei comenzi, vom folosi mecanisme precum:

```
if comando; then
    ...
fi
```

sau vom analiza variabila specială:

\$?

Vom studia aceste concepte într-un capitol dedicat controlului execuției.

Caseta „În spatele cortinei”

Ce se întâmplă în interior?

Atunci când Bash întâlnește expresia:

\$(comandă)

el creează un context de execuție pentru acea comandă, colectează ieșirea standard și o transformă într-un șir de caractere care va participa la etapele următoare ale procesării liniei de comandă.

Acest lucru explică de ce rezultatul unei substituții poate fi ulterior afectat de **Word Splitting** și de quoting.

Greșeală din lumea reală

Un administrator scrie:

```
for FILE in $(find . -name "*.txt")
do
    ...
done
```

Scriptul funcționează până când apare un fișier cu spații în nume.

Problema nu este `find`.

Problema este că rezultatul substituției este supus etapelor ulterioare de procesare, inclusiv **Word Splitting**.

În capitolul dedicat procesării sigure a listelor de fișiere vom prezenta soluția corectă.

Administrator Notes

Folosește Command Substitution pentru a obține informații, nu pentru a procesa liste complexe de fișiere.

Dacă rezultatul poate conține spații, linii noi sau alte caractere speciale, analizează cu atenție modul în care va fi utilizat și aplică quoting atunci când intenția este de a transmite un singur argument.

Cum aş depana această problemă?

Problema:

Rezultatul unei Command Substitution pare să fie împărțit în mai multe argumente.

Pași:

1. Afișez rezultatul folosind:

```
printf '<%s>\n' "$REZULTAT"
```

2. Verific dacă expresia este între ghilimele.
3. Activez:

```
set -x
```

și observ forma finală a comenzii.

4. Analizez dacă după substituție are loc Word Splitting.
-

Laborator 3.8

Obiectiv

Înțelegerea mecanismului Command Substitution.

Cerințe

1. Afișează data folosind `$(date)`.
 2. Creează un director care conține data curentă în nume.
 3. Afișează numele calculatorului folosind `$(hostname)`.
 4. Compară comportamentul unei variabile obținute prin Command Substitution cu și fără ghilimele.
 5. Explică fiecare rezultat.
-

Soluția laboratorului

- Bash execută mai întâi comanda din `$(...)`.
 - Rezultatul este introdus în linia de comandă înainte de execuția comenzii principale.
 - Programul apelat nu știe că argumentul provine dintr-o substituție de comandă.
 - Rezultatul substituției poate participa la etapele ulterioare ale procesării, inclusiv Word Splitting.
-

Testează-te

Ce primește, de fapt, `echo`?

Comanda:

```
echo "Utilizator: $(whoami)"
```

A)

Expresia `$(whoami)`.

B)

Numele utilizatorului rezultat în urma executării comenzii `whoami`.

C)

Obiectul procesului `whoami`.

Răspuns: B.

`echo` primește doar textul produs de comanda `whoami`; substituția este rezolvată de Bash înainte de execuția lui `echo`.

Ce trebuie să reții

- Command Substitution înlocuiește expresia `$(...)` cu ieșirea standard a comenzii executate.
- Sintaxa recomandată este `$(...)`, nu backticks.
- Rezultatul substituției participă la etapele ulterioare de procesare ale Bash-ului.
- Quoting rămâne esențial atunci când rezultatul trebuie tratat ca un singur argument.
- Command Substitution este unul dintre mecanismele fundamentale pentru automatizare și compunerea comenzilor.

Capitolul 3

3.9 Arithmetic Expansion – Cum efectuează Bash calcule înainte de execuția unei comenzi

*„Bash tratează în mod obișnuit toate valorile ca șiruri de caractere. Atunci când avem nevoie de calcule, trebuie să îi indicăm explicit că urmează o expresie aritmetică. Acest mecanism poartă numele de **Arithmetic Expansion**.”*

Introducere

Să presupunem că scriem:

```
A=10  
B=20
```

```
echo "$A+$B"
```

Mulți începători se așteaptă să obțină:

```
30
```

Rezultatul real este:

```
10+20
```

De ce?

Pentru că până în acest moment Bash a făcut doar **Parameter Expansion**.

A înlocuit variabilele cu valorile lor.

Dar nu a efectuat niciun calcul.

Cum îi spunem lui Bash că dorim un calcul?

Prin sintaxa:

```
$(( expresie ))
```

Acum:

```
A=10  
B=20
```

```
echo $((A+B))
```

rezultatul este:

30

Observă diferența.

Nu `echo` face calculul.

Bash îl calculează înainte.

Intern putem imagina procesul astfel:

```
echo $((A+B))  
  |  
  ▼  
Arithmetic Expansion  
  |  
  ▼  
echo 30  
  |  
  ▼  
execve(...)
```

Același model apare din nou.

Programul executat primește deja rezultatul final.

Bash tratează variabilele numeric

Un aspect elegant al lui Bash este că în interiorul unei expresii aritmetice nu mai este obligatoriu să folosim caracterul `$`.

Ambele variante sunt corecte:

```
echo $((A+B))
```

și

```
echo $(($A+$B))
```

Totuși, prima este mai clară și este recomandată.

De ce?

Pentru că în interiorul lui `$((...))`, Bash știe deja că identificatorii reprezintă variabile numerice.

Aceasta este o convenție pe care o vom folosi în întreaga carte.

Operatorii disponibili

Bash oferă majoritatea operatorilor întâlniți în limbajele C și Java.

Adunare

```
echo $((5+3))
```

Rezultat:

8

Scădere

```
echo $((10-4))
```

Rezultat:

6

Înmulțire

```
echo $((6*7))
```

Rezultat:

42

Împărțire

```
echo $((20/4))
```

Rezultat:

5

Este important de observat că Bash lucrează, în mod implicit, cu **numere întregi**.

Exemplu:

```
echo $((5/2))
```

Rezultatul este:

2

Nu:

Pentru calcule în virgulă mobilă vom utiliza, de regulă, utilitare precum `bc` sau `awk`, pe care le vom studia în capitolele dedicate.

Restul împărțirii

Operatorul:

`%`

calculează restul împărțirii.

Exemplu:

```
echo $((17%5))
```

Rezultatul:

2

Acest operator este foarte util în scripturi pentru verificarea numerelor pare și impare sau pentru cicluri repetitive.

Incrementare și decrementare

Bash suportă:

```
((CONTOR++))
```

și

```
((CONTOR--))
```

Aceste forme sunt foarte utile în bucle și în numărarea evenimentelor.

Notă pentru autor: când vom ajunge la capitolul despre `for`, `while` și `until`, vom reveni asupra acestor operatori și vom explica diferența dintre forma prefixată (`++i`) și cea postfixată (`i++`).

Demonstrație practică

Construim un contor simplu:

```
COUNT=1
```

```
echo "$COUNT"
((COUNT++))
echo "$COUNT"
((COUNT++))
echo "$COUNT"
```

Rezultatul:

```
1
2
3
```

Observă că variabila este modificată de expresia aritmetică.

Caseta „În spatele cortinei”

De ce Bash nu calculează automat?

Pentru că Bash este, înainte de toate, un shell.

Majoritatea informațiilor cu care lucrează sunt șiruri de caractere:

- nume de fișiere;
- directoare;
- utilizatori;
- variabile de mediu;
- argumente ale comenzilor.

Dacă orice expresie de forma:

```
10+20
```

ar fi interpretată automat ca un calcul, ar exista numeroase ambiguități.

Prin folosirea sintaxei:

```
$( ( . . . ) )
```

utilizatorul îi spune explicit lui Bash:

„De aici începe o expresie aritmetică.”

Această alegere face limbajul mai previzibil și reduce interpretările greșite.

Greșeală din lumea reală

Un administrator scrie:

```
COUNT=1
```

```
echo "$COUNT+1"
```

și se întreabă de ce rezultatul este:

```
1+1
```

Explicația este simplă.

Bash nu vede o expresie aritmetică.

Vede doar un șir de caractere.

Forma corectă este:

```
echo $((COUNT+1))
```

Administrator Notes

Atunci când citești un script, încearcă să identifici în ce „mod” interpretează Bash fiecare expresie.

Întreabă-te:

- Este un șir de caractere?
- Este o variabilă?
- Este o expresie aritmetică?
- Este o substituție de comandă?

Această disciplină mentală te ajută să anticipezi rezultatul înainte de a executa scriptul.

Cum aș depana această problemă?

Problema:

Un calcul produce un rezultat neașteptat.

Pași:

1. Verific dacă expresia este încadrată în:

```
$(( ... ))
```

2. Verific valorile variabilelor implicate:

declare -p A B

3. Verific dacă lucrez cu numere întregi sau am nevoie de precizie în virgulă mobilă.
 4. Testez expresia într-o comandă simplă `echo` înainte de a o integra într-un script.
-

Laborator 3.9

Obiectiv

Înțelegerea mecanismului Arithmetic Expansion.

Cerințe

1. Creează variabilele:

A=15

B=4

2. Calculează:

- suma;
- diferența;
- produsul;
- câtul;
- restul împărțirii.

3. Creează un contor și incrementează-l de cinci ori.

4. Verifică rezultatul împărțirii:

`echo $( (7/3) )`

și explică de ce nu obții `2.333...`

Soluția laboratorului

- Arithmetic Expansion este declanșată prin sintaxa `$( ( . . . ) )`.
 - Bash efectuează calculele înainte de execuția comenzii.
 - Variabilele pot fi folosite direct în expresiile aritmetice.
 - Împărțirea implicită este întreagă.
 - Operatorul `%` calculează restul împărțirii.
-

Testează-te

Ce va afișa următorul script?

A=8
B=3

```
echo "$A+$B"
```

```
echo $((A+B))
```

```
echo $((A/B))
```

```
echo $((A%B))
```

A)

11
2.66
2

B)

8+3
11
2
2

C)

Eroare.

Răspuns

Varianta B.

Explicație:

- Prima comandă realizează doar **Parameter Expansion**.
 - A doua execută **Arithmetic Expansion**.
 - Împărțirea este întreagă.
 - Ultima comandă calculează restul împărțirii.
-

Ce trebuie să reții

- Bash nu interpretează automat expresiile matematice.
- Arithmetic Expansion este declanșată prin `$((...))`.
- În interiorul expresiilor aritmetice, variabilele pot fi folosite fără caracterul `$`.

- Bash lucrează implicit cu numere întregi.
- Înainte de a căuta o eroare într-un script, verifică dacă Bash interpretează expresia în modul pe care îl intenționezi.

Capitolul 3

3.10 Brace Expansion – Cum generează Bash zeci sau sute de argumente fără bucle

„Brace Expansion este una dintre puținele expansiuni care nu depinde de existența fișierelor, de variabile sau de comenzi. Ea generează pur și simplu combinații de șiruri de caractere. Este o facilitate de productivitate implementată exclusiv în shell.”

Introducere

Presupunem că vrem să creăm trei fișiere:

```
raport1.txt  
raport2.txt  
raport3.txt
```

Putem scrie:

```
touch raport1.txt raport2.txt raport3.txt
```

Funcționează.

Dar este repetitiv.

Bash oferă o alternativă elegantă:

```
touch raport{1,2,3}.txt
```

Rezultatul este identic.

Cum?

Ce este Brace Expansion?

Brace Expansion este mecanismul prin care Bash generează automat mai multe șiruri de caractere pornind de la un model.

Exemplu:

```
echo file{1,2,3}
```

Rezultatul:

```
file1  
file2  
file3
```

Observă ceva foarte important.

Nu există niciun fișier pe disc.

Și totuși funcționează.

Aceasta demonstrează că Brace Expansion nu caută fișiere.

Generează text.

Prima demonstrație

Executăm:

```
printf '<%s>\n' server{01,02,03}
```

Rezultatul:

```
<server01>  
<server02>  
<server03>
```

Dacă foloseam `echo`, vedeam doar textul.

`printf` ne arată însă foarte clar că Bash a generat **trei argumente distincte**.

Din nou, `printf` este un aliat excelent pentru înțelegerea mecanismelor interne.

Secvențe

Brace Expansion poate genera și intervale.

Exemplu:

```
echo {1..5}
```

Rezultatul:

```
1 2 3 4 5
```

Putem folosi și litere:

```
echo {a..f}
```

Rezultatul:

a b c d e f

Și chiar pași:

```
echo {0..20..5}
```

Rezultatul:

```
0 5 10 15 20
```

Acest mecanism este foarte util pentru generarea rapidă de liste.

Brace Expansion și variabilele

Aici apare una dintre cele mai interesante întrebări.

Presupunem:

N=5

```
echo {1..$N}
```

Mulți se așteaptă la:

```
1 2 3 4 5
```

Dar rezultatul este:

```
{1..5}
```

sau chiar expresia rămâne neexpandată, în funcție de context.

De ce?

Pentru că **Brace Expansion are loc înainte de Parameter Expansion**.

În momentul în care Bash încearcă să proceseze `{1..$N}`, valoarea lui `$N` nu este încă cunoscută.

Acesta este unul dintre cele mai bune exemple care demonstrează că ordinea expansiunilor nu este doar o informație teoretică, ci influențează direct rezultatul.

Caseta „În spatele cortinei”

De ce Brace Expansion este prima?

Pentru că este complet independentă.

Nu are nevoie:

- de sistemul de fișiere;

- de variabile;
- de comenzi;
- de procese.

Primește un șablon și produce o listă de șiruri.

Această independență explică de ce Bash o execută înaintea tuturor celorlalte expansiuni.

Greșeală din lumea reală

Un administrator încearcă să creeze directoare pentru un număr variabil de utilizatori:

```
MAX=10
```

```
mkdir user{1..$MAX}
```

Scriptul nu produce rezultatul dorit.

Problema nu este `mkdir`.

Problema este că Brace Expansion este evaluată înainte ca Bash să înlocuiască valoarea variabilei `MAX`.

În acest caz este necesară o altă abordare, de exemplu o buclă `for`, pe care o vom studia în capitolele dedicate structurilor de control.

Administrator Notes

Folosește Brace Expansion atunci când lista este cunoscută în momentul scrierii comenzii.

Dacă lista depinde de valori calculate în timpul execuției, de intrarea utilizatorului sau de conținutul sistemului de fișiere, o buclă sau o altă tehnică este, de regulă, soluția potrivită.

Cum aș depana această problemă?

Problema:

Brace Expansion nu generează rezultatul așteptat.

Pașii:

1. Verific dacă expresia conține variabile.
2. Îmi amintesc ordinea expansiunilor: Brace Expansion este prima.
3. Testez expresia cu `printf '%s>\n'`.
4. Dacă am nevoie de valori dinamice, aleg o structură de control în locul Brace Expansion.

Laborator 3.10

Obiectiv

Înțelegerea mecanismului Brace Expansion.

Cerințe

1. Creează fișierele:

```
touch raport{1,2,3}.txt
```

2. Afișează:

```
echo {a..g}
```

3. Afișează:

```
echo {10..100..10}
```

4. Testează:

```
N=5
```

```
echo {1..$N}
```

5. Explică de ce ultimul exemplu nu produce rezultatul intuitiv.
-

Soluția laboratorului

- Brace Expansion generează text, nu caută fișiere.
 - Intervalele pot fi numerice sau alfabetice.
 - Se pot specifica pași.
 - Brace Expansion este executată înainte de Parameter Expansion.
 - Din acest motiv, variabilele nu pot controla direct intervalele generate de Brace Expansion.
-

Testează-te

Ce va afișa următoarea comandă?

```
NUME=test
```

```
printf '<%s>\n' ${NUME},prod}
```

A)

```
<test>  
<prod>
```

B)

```
<${NUME}>  
<prod>
```

C)

Eroare.

Răspuns corect: B.

Explicație:

Mai întâi are loc **Brace Expansion**, care produce două șiruri:

```
${NUME}  
prod
```

Abia apoi **Parameter Expansion** înlocuiește `${NUME}` cu valoarea sa. Acest exemplu este foarte valoros pentru că obligă cititorul să aplice ordinea expansiunilor în loc să încerce să ghicească rezultatul.

Capitolul 3

3.11 Cum citește Bash o linie de comandă – Un studiu complet

„Un administrator experimentat nu citește o comandă de la stânga la dreapta. El vede etapele prin care aceasta va trece înainte de a fi executată. Acest capitol își propune să formeze exact această abilitate.”

Introducere

Până acum am învățat separat:

- Parameter Expansion
- Command Substitution
- Arithmetic Expansion
- Brace Expansion
- Word Splitting
- Pathname Expansion

- Quoting

Acum le vom combina.

Pentru că în lumea reală nu vei executa:

```
echo "$HOME"
```

de o mie de ori.

Vei scrie comenzi complexe.

Și trebuie să poți anticipa rezultatul.

Exemplul 1

Pornim de la ceva simplu.

Presupunem:

```
HOME=/home/daniel
```

Comanda este:

```
echo "$HOME"
```

Etapa 1

Parsing.

Bash identifică:

- comanda `echo`
 - un argument
-

Etapa 2

Parameter Expansion

Rezultatul devine:

```
/home/daniel
```

Etapa 3

Word Splitting

Nu are loc.

De ce?

Argumentul este între ghilimele.

Etapa 4

Pathname Expansion

Nu există modele.

Nu se întâmplă nimic.

Etapa 5

Quote Removal

Ghilimelele sunt eliminate.

Programul primește:

echo

Argument:

/home/daniel

Observă.

Acum nu mai descriem doar rezultatul.

Descriem întreaga călătorie.

Exemplul 2

Acum ceva mai interesant.

Presupunem:

DIR="Documente Personale"

Comanda:

```
printf '<%s>\n' $DIR
```

Parsing

↓

Parameter Expansion

↓

Rezultă:

Documente Personale

↓

Word Splitting

↓

Rezultă:

Argumentul 1

Documente

Argumentul 2

Personale

↓

Quote Removal

↓

Execuție.

Acum aceeași comandă:

```
printf '<%s>\n' "$DIR"
```

Parcurge exact aceleași etape.

Dar Word Splitting este inhibat.

Rezultatul este:

un singur argument.

Aici cititorul începe să observe că două comenzi aproape identice produc rezultate diferite doar pentru că una modifică o etapă din procesare.

Acesta este exact tipul de înțelegere pe care îl căutăm.

Exemplul 3

Acum combinăm trei mecanisme.

Presupunem:

PREFIX=server

Comanda:

```
printf '<%s>\n' ${PREFIX}{01,02}
```

Brace Expansion

↓

Produce:

```
${PREFIX}01  
${PREFIX}02
```

↓

Parameter Expansion

↓

Produce:

```
server01  
server02
```

↓

Word Splitting

Nu apare.

↓

Pathname Expansion

Nu apare.

↓

Execuție.

Acesta este un exemplu extraordinar.

Pentru că demonstrează că ordinea expansiunilor nu este doar teorie.

Produce efecte reale.

Exemplul 4

Acum introducem și Command Substitution.

Comanda:

```
echo "$(hostname)-$(date +%Y)"
```

Parsing

↓

Command Substitution

↓

Prima comandă produce:

server01

A doua produce:

2026

↓

Parameter Expansion

Nu există.

↓

Word Splitting

Nu apare deoarece rezultatul este între ghilimele.

↓

Quote Removal

↓

Execuție.

Programul primește:

server01-2026

Exemplul 5

Acesta este unul dintre cele mai frumoase exemple.

```
FILES=$(printf '%s\n' *.txt)
printf '<%s>\n' "$FILES"
```

Aici cititorul trebuie să înceapă să recunoască singur etapele.

Nu le mai explicăm pe toate.

Le cerem lui.

Acesta este începutul autonomiei.

Caseta „În spatele cortinei”

Din acest moment începi să citești Bash diferit

La începutul cărții citeai:

```
cp "$HOME"/*.txt backup/
```

ca pe o comandă.

Acum începi să vezi:

- parsing;
- parameter expansion;
- pathname expansion;
- quote removal;
- `execve()`.

Acesta este momentul în care începi să gândești asemenea shell-ului.

Greșeală din lumea reală

Un administrator modifică o comandă prin adăugarea unor ghilimele.

Rezultatul se schimbă complet.

El presupune că ghilimelele afectează doar afișarea.

În realitate, ele modifică una dintre etapele fundamentale ale procesării.

Problema nu este comanda.

Problema este modelul mental incomplet.

Administrator Notes

Când întâlnești o comandă dificilă, nu încerca să o înțelegi dintr-o singură privire.

Descompune-o.

Întreabă-te:

- Ce expansiuni conține?
- În ce ordine apar?
- Ce argumente vor exista înainte de execuție?

Această metodă este mult mai eficientă decât încercarea de a memora exemple.

Laborator 3.11

Obiectiv

Înțelegerea completă a procesării unei linii de comandă.

Cerințe

Analizează următoarele comenzi fără să le execuți inițial.

Pentru fiecare:

1. Identifică toate expansiunile.
2. Scrie ordinea în care apar.
3. Descrie argumentele finale.
4. Abia apoi execută comanda și compară rezultatul cu predicția.

Comenzi:

```
echo "${HOME}/${hostname}"
```

```
printf '<%s>\n' ${USER}{1,2}
```

```
echo "$HOME/*.txt"
```

```
echo "$((5+3))"
```

```
printf '<%s>\n' *
```

Testează-te

Întrebare

Dacă ai înțeles cu adevărat capitolele precedente, ar trebui să poți răspunde fără să execuți comenzile.

Ce se întâmplă prima dată?

A)

Parameter Expansion

B)

Brace Expansion

C)

Word Splitting

Răspuns:

B.

Nu pentru că trebuie memorat.

Ci pentru că acum înțelegi arhitectura Bash.

Ce trebuie să reții

- O comandă Bash este procesată în etape distincte.
- Fiecare etapă are un rol bine definit și poate modifica rezultatul final.
- Quoting nu este un detaliu de sintaxă, ci un mecanism care influențează procesarea.
- Înțelegerea ordinii expansiunilor permite anticiparea comportamentului unei comenzi complexe.
- Un administrator experimentat analizează o comandă etapă cu etapă, nu doar rezultatul ei.

Capitolul 3

3.12 Studii de caz – Cum analizează un administrator Linux o comandă Bash

„În capitolele anterioare am învățat mecanisme separate. În acest capitol vom face ceea ce face un administrator în fiecare zi: vom lua comenzi reale și le vom analiza pas cu pas.”

Introducere

Până acum am studiat:

- Parameter Expansion
- Command Substitution
- Arithmetic Expansion
- Brace Expansion
- Word Splitting
- Pathname Expansion
- Quoting

Acum le vom combina.

Nu pentru a învăța ceva nou.

Ci pentru a transforma cunoștințele în deprinderi.

În acest capitol nu ne interesează doar rezultatul unei comenzi.

Ne interesează **raționamentul** care conduce la acel rezultat.

Studiul de caz 1

Presupunem:

```
HOME=/home/daniel
```

și executăm:

```
echo "$HOME"
```

Întrebarea

Ce vede echo?

Mulți răspund:

Variabila HOME.

Nu.

Acesta este primul lucru pe care vreau să-l corectăm.

echo nu vede niciodată variabila.

El vede doar:

/home/daniel

Analiza

Parsing

↓

Parameter Expansion

↓

Quote Removal

↓

Execuție

Concluzie:

Programul nu știe că argumentul provine dintr-o variabilă.

Studiul de caz 2

Presupunem:

DIR="Documente Personale"

Comanda:

```
printf '<%s>\n' $DIR
```

Întrebare

De ce apar două linii?

Mulți răspund:

Pentru că `printf` împarte textul.

Greșit.

`printf` nu împarte nimic.

Analiza

Parameter Expansion

↓

Rezultă:

Documente Personale

↓

Word Splitting

↓

Două argumente

↓

Execuție

Concluzie:

Împărțirea este făcută de Bash.

Studiul de caz 3

Directorul conține:

a.txt
b.txt
c.log

Comanda:

```
printf '<%s>\n' *.txt
```

Întrebare

Cine găsește fișierele?

Mulți spun:

`printf`.

Nu.

Analiza

Pathname Expansion

↓

Rezultă:

a.txt

b.txt

↓

Execuție

Concluzie:

`printf` primește deja două argumente.

Studiul de caz 4

Comanda:

`echo "$(hostname)"`

Întrebare

Cine execută `hostname`?

Nu `echo`.

Analiza

Command Substitution

↓

Rezultă:

server01

↓

Execuție

Concluzie:

`echo` primește rezultatul, nu comanda.

Studiul de caz 5

Comanda:

```
echo $((8+4))
```

Întrebare

Cine face calculul?

Nu echo.

Analiza

Arithmetic Expansion

↓

Rezultă:

12

↓

Execuție

Concluzie:

Programul nu știe că a existat o expresie matematică.

Un exercițiu de gândire

Să analizăm:

```
echo "$HOME"/*.txt
```

Nu executa încă.

Ia o foaie și răspunde:

1. Ce tipuri de expansiuni există?
2. Care apare prima?
3. Care este inhibată?
4. Câte argumente va primi echo?

Abia după aceea rulează comanda.

Acest exercițiu este mai valoros decât memorarea rezultatului.

Caseta „În spatele cortinei”

Un administrator nu citește doar comenzi

Pe măsură ce câștigi experiență, vei observa o schimbare interesantă.

La început vezi:

```
cp "$HOME"/* .txt backup/
```

După o vreme începi să vezi:

- o variabilă care va fi expandată;
- un model care va fi rezolvat prin globbing;
- un argument protejat de quoting;
- o comandă care va primi o listă de fișiere.

Această schimbare de perspectivă apare treptat.

Nu trebuie grăbită.

Ea este rezultatul exercițiului și al înțelegerii mecanismelor.

Greșeală din lumea reală

Un administrator vede o comandă lungă și încearcă să o înțeleagă dintr-o singură privire.

Rezultatul este adesea confuz.

O metodă mai eficientă este:

- identifică expansiunile;
- stabilește ordinea lor;
- determină argumentele finale;
- abia apoi analizează programul executat.

Această metodă reduce foarte mult timpul de depanare.

Administrator Notes

Atunci când întâlnești o comandă dificilă, nu încerca să o memorezi.

Descompune-o.

Fiecare comandă Bash poate fi analizată prin aceeași metodă.

Dacă îți formezi acest reflex, vei putea înțelege și comenzi pe care nu le-ai văzut niciodată.

Laborator 3.12

Obiectiv

Aplicarea tuturor mecanismelor studiate până acum.

Cerințe

Pentru fiecare comandă:

- identifică expansiunile;
- stabilește ordinea lor;
- scrie argumentele finale;
- execută comanda și verifică dacă analiza ta a fost corectă.

```
echo "$HOME"
```

```
printf '<%s>\n' *.txt
```

```
echo "$(hostname)"
```

```
echo $((15+7))
```

```
printf '<%s>\n' ${USER}{1,2}
```

Testează-te

Care dintre afirmațiile următoare este adevărată?

A)

Programul executat realizează expansiunea variabilelor.

B)

Programul executat primește argumentele după ce Bash a terminat procesarea liniei de comandă.

C)

echo știe să interpreteze \$(. . .).

Răspuns corect: B.

Ce trebuie să reții

- Bash procesează linia de comandă înainte de execuția programului.

- Mecanismele studiate nu sunt independente; ele colaborează într-o ordine bine definită.
- Majoritatea erorilor pot fi înțelese analizând etapele de procesare, nu doar rezultatul final.
- În practică, o metodă sistematică de analiză este mai valoroasă decât memorarea unor cazuri particulare.

Dacă ar fi să rezum tot ce am învățat până acum într-o singură idee, aş spune că Bash este un interpret care pregăteşte argumentele înainte ca programele să înceapă execuția. Variabilele, substituțiile de comenzi, calculele, modelele de fişiere şi quoting-ul sunt toate mecanisme prin care Bash transformă ceea ce scriem într-o listă de argumente pe care programele o pot înţelege. Dacă înţelegi această idee, multe dintre comportamentele aparent surprinzătoare ale shell-ului devin logice.

Capitolul 4

Redirecționări și fluxurile standard – Cum comunică programele între ele și cu utilizatorul

„Unul dintre cele mai elegante concepte din Unix este că programele nu trebuie să ştie dacă citesc de la tastatură, dintr-un fişier sau din rezultatul altei comenzi. Ele citesc şi scriu prin fluxuri standard. Shell-ul este cel care decide unde sunt conectate aceste fluxuri.”

Introducere

Până acum am studiat modul în care Bash pregăteşte o comandă înainte de execuție.

Dar, odată ce programul porneşte, apare o altă întrebare:

Cum comunică acel program cu lumea exterioară?

Cum citeşte date?

Cum afişează rezultate?

Cum raportează erorile?

Răspunsul stă într-un concept fundamental al sistemelor Unix:

fluxurile standard (*standard streams*).

Acest capitol este unul dintre cele mai importante din întreaga carte. Dacă înţelegi bine fluxurile standard, vei înţelege mult mai uşor redirecţionările, pipe-urile, scriptingul şi chiar multe API-uri moderne.

O idee care vine din filosofia Unix

La începuturile Unix, creatorii sistemului și-au pus o întrebare simplă:

Cum putem scrie programe care să fie cât mai independente de mediul în care rulează?

Dacă un program citește direct de la tastatură și scrie direct pe ecran, devine greu de reutilizat.

Soluția a fost elegantă.

Fiecărui proces i se oferă trei canale de comunicare standard.

Programul nu trebuie să știe unde duc aceste canale.

El doar citește și scrie.

Shell-ul este cel care decide conexiunile.

Această separare a responsabilităților este unul dintre motivele pentru care Unix a rămas atât de flexibil timp de decenii.

Cele trei fluxuri standard

Orice proces Unix pornește, în mod normal, cu trei descriptori de fișiere deja deschiși.

Descriptor	Nume	Rol
0	Standard Input (stdin)	De unde citește programul datele.
1	Standard Output (stdout)	Unde trimite rezultatele normale.
2	Standard Error (stderr)	Unde trimite mesajele de eroare și diagnostic.

Aceste numere nu sunt alese întâmplător.

Ele sunt convenții istorice ale sistemelor Unix și sunt folosite în toate shell-urile moderne.

Prima demonstrație

Executăm:

```
cat
```

Programul pare că "nu face nimic".

De fapt, el citește de pe **stdin** și scrie imediat pe **stdout**.

Dacă tastăm:

```
Salut
```

și apăsăm **Enter**, vom vedea:

- căile către fișierele găsite;
- mesaje de tipul:

Permission denied

Dacă amestecăm cele două categorii de informații, devine dificil să procesăm rezultatul.

Prin existența lui **stderr**, putem păstra separate:

- datele utile;
- mesajele de diagnostic.

Această decizie de proiectare s-a dovedit extrem de inspirată și este folosită și astăzi.

Caseta „În spatele cortinei”

De ce se numesc „descriptori de fișiere”?

Pentru că, în Unix, aproape totul este tratat ca un fișier sau ca un obiect care poate fi accesat prin aceeași interfață: fișiere obișnuite, terminale, pipe-uri, socket-uri și multe alte resurse.

Un descriptor de fișier este un identificator numeric prin care un proces accesează unul dintre aceste obiecte.

De aceea, redirecționarea către un fișier și redirecționarea către un terminal folosesc același mecanism intern.

Această uniformitate este una dintre cele mai elegante idei din arhitectura Unix.

Greșeală din lumea reală

Un administrator execută:

```
find / -name "*.conf" > rezultate.txt
```

Deschide fișierul și observă că lipsesc mesajele de eroare.

Concluzia lui este că nu au existat erori.

În realitate, mesajele au fost afișate pe terminal prin **stderr** și nu au fost redirecționate.

Abia când înțelegi existența celor două fluxuri de ieșire începi să înțelegi de ce anumite redirecționări nu produc rezultatul așteptat.

Administrator Notes

Când un program produce atât rezultate, cât și erori, întreabă-te întotdeauna:

Pe ce flux a fost trimis fiecare mesaj?

Foarte multe situații aparent misterioase devin clare odată ce identifici dacă informația a fost scrisă pe **stdout** sau pe **stderr**.

Cum aş depana această problemă?

Problema:

Am redirecționat ieșirea într-un fișier, dar mesajele de eroare apar în continuare pe ecran.

Pașii:

1. Verific dacă programul folosește `stderr` pentru mesaje de eroare.
2. Confirm că am redirecționat doar `stdout`.
3. Testez separat redirecționarea fiecărui flux.
4. Abia apoi combin redirecționările atunci când este necesar.

Laborator 4.1

Obiectiv

Observarea fluxurilor standard.

Cerințe

1. Rulează:

```
cat
```

Introdu câteva linii de text și observă comportamentul.

2. Rulează:

```
ls
```

Observă că rezultatele apar pe ecran.

3. Rulează:

```
ls fisier-care-nu-exista
```

Observă că mesajul de eroare apare tot pe ecran, dar are o natură diferită.

4. Răspunde:

- Care comandă citește de la **stdin**?
 - Care comandă scrie pe **stdout**?
 - Care comandă produce informații pe **stderr**?
-

Soluția laboratorului

- **cat** citește de la **stdin** și scrie pe **stdout**.
- **ls** afișează lista fișierelor pe **stdout**.
- **ls** trimite mesajele de eroare pe **stderr** atunci când apare o problemă.

Ce trebuie să reții

- Orice proces Unix pornește, în mod normal, cu trei fluxuri standard.
- **stdin** este sursa de intrare.
- **stdout** transportă rezultatele normale.
- **stderr** transportă mesajele de eroare și diagnostic.
- Shell-ul conectează aceste fluxuri și le poate redirecționa fără ca programul să fie modificat.

Capitolul 4

4.2 Prima redirecționare – Ce înseamnă, de fapt, caracterul >

„Caracterul > nu îi spune unui program să scrie într-un fișier. Îi spune shell-ului să conecteze fluxul standard de ieșire al programului către un fișier înainte ca programul să înceapă execuția.”

Introducere

Acum știm că orice proces pornește cu trei fluxuri standard.

În mod normal:

stdin ← tastatură

stdout → terminal

stderr → terminal

Dar Bash poate modifica aceste conexiuni.

Acesta este rolul redirectionărilor.

Primul exemplu

Executăm:

```
echo "Salut"
```

Fluxurile sunt:

```
stdout
  |
  ▼
Terminal
```

Vedem:

```
Salut
```

Acum executăm:

```
echo "Salut" > salut.txt
```

Ce s-a schimbat?

Programul echo?

Nu.

Ce vede Bash?

Înainte de execuție, Bash observă caracterul:

```
>
```

Și spune:

„În loc să conectez stdout la terminal, îl voi conecta la fișierul `salut.txt`.”

Abia după aceea pornește programul.

Modelul devine:

```
          stdout
echo -----> salut.txt
```

În primul caz:

stdout

↓

terminal

În al doilea:

stdout

↓

fișier

Programul `echo` nu știe că scrie într-un fișier.

El continuă să scrie în descriptorul **1**.

Numai destinația s-a schimbat.

Demonstrație

Executăm:

```
echo "Linux" > exemplu.txt
```

Acum:

```
cat exemplu.txt
```

afișează:

```
Linux
```

Observă că `echo` nu are o opțiune precum:

```
--save-to-file
```

Nici nu are nevoie.

Shell-ul a făcut conexiunea înainte.

Un model mental

Imaginează-ți instalația electrică dintr-o casă.

Ai aceeași lampă.

Dacă schimbi poziția întrerupătorului, lumina ajunge într-o altă încăpere.

Nu ai modificat becul.

Ai modificat traseul curentului.

Exact așa funcționează și redirectionările.

Programul nu se schimbă.

Se schimbă traseul datelor.

De ce este elegantă această soluție?

Pentru că același program poate fi folosit în zeci de contexte.

De exemplu:

```
echo "Salut"
```

poate scrie:

- pe ecran;
- într-un fișier;
- într-un pipe;
- într-un socket;
- într-un dispozitiv.

Fără ca autorul programului să fi prevăzut toate aceste situații.

Aceasta este una dintre cele mai elegante idei din filosofia Unix:

Programele rezolvă o problemă. Shell-ul decide cum sunt conectate între ele și cu exteriorul.

Caseta „În spatele cortinei”

Cine creează fișierul?

Mulți utilizatori cred că:

```
echo
```

creează fișierul.

Nu.

În cele mai multe cazuri, **Bash** este cel care deschide (și, dacă este necesar, creează sau trunchiază) fișierul înainte de execuția programului.

Apoi transmite programului un descriptor de fișier deja conectat.

Această diferență este foarte importantă.

Dacă shell-ul nu poate deschide fișierul (de exemplu, din lipsă de permisiuni), programul nici măcar nu mai pornește.

Vom demonstra acest lucru într-un laborator ulterior.

Greșeală din lumea reală

Un administrator execută:

```
echo "Test" > /root/test.txt
```

ca utilizator obișnuit.

Primește:

```
Permission denied
```

și presupune că eroarea vine din `echo`.

În realitate, `echo` nu a fost executat niciodată.

Shell-ul a încercat să deschidă fișierul și a eșuat înainte de lansarea programului.

Această observație schimbă complet modul de depanare.

Administrator Notes

Atunci când o redirectionare produce o eroare, întreabă-te mai întâi:

A reușit shell-ul să deschidă fișierul?

Nu presupune automat că problema este în program.

Foarte des, programul nici nu a început execuția.

Cum aș depana această problemă?

Problema:

Comanda cu `>` nu funcționează.

Pașii:

1. Verific dacă directorul există.
2. Verific permisiunile asupra directorului și fișierului.

3. Verific utilizatorul curent (`whoami`).
4. Abia apoi analizez programul executat.

Foarte des vei descoperi că problema este în redirectionare, nu în comandă.

Laborator 4.2

Obiectiv

Înțelegerea rolului operatorului `>`.

Cerințe

1. Creează un fișier:

```
echo "Linux" > linux.txt
```

2. Verifică conținutul:

```
cat linux.txt
```

3. Repetă comanda cu alt text și observă ce se întâmplă cu fișierul existent.
 4. Încearcă să redirectionezi către un director pentru care nu ai drept de scriere și observă mesajul afișat.
 5. Explică cine produce eroarea: shell-ul sau programul.
-

Soluția laboratorului

- `>` redirectionează fluxul **stdout**.
 - Shell-ul deschide fișierul înainte de execuția programului.
 - Dacă fișierul există, conținutul lui este înlocuit.
 - Dacă shell-ul nu poate deschide fișierul, programul nu este executat.
-

Ce trebuie să reții

- `>` nu schimbă comportamentul programului, ci conexiunea fluxului **stdout**.
- Shell-ul gestionează redirectionarea înainte de lansarea programului.
- Erorile de permisiuni pot apărea înainte ca programul să înceapă execuția.
- Înțelegerea acestui mecanism simplifică foarte mult depanarea.

Capitolul 4

4.3 De ce există >>? – Diferența dintre suprascriere și adăugare

„În administrarea sistemelor există o diferență fundamentală între a înlocui informația și a o păstra. Operatorii > și >> există tocmai pentru a răspunde acestor două nevoi diferite.”

Introducere

În capitolul anterior am folosit:

```
echo "Linux" > linux.txt
```

și am observat că fișierul a fost creat.

Acum executăm:

```
echo "Bash" > linux.txt
```

Mulți utilizatori se așteaptă ca fișierul să conțină:

```
Linux  
Bash
```

Dar rezultatul este:

```
Bash
```

Ce s-a întâmplat?

Unde a dispărut prima linie?

Această întrebare este punctul de plecare al acestui capitol.

Ce înseamnă, de fapt, >?

Operatorul > nu înseamnă:

„Scrie într-un fișier.”

El înseamnă:

„Deschide fișierul pentru scriere și începe de la începutul lui.”

Dacă fișierul există deja, conținutul anterior este eliminat înainte ca programul să înceapă să scrie.

Acest proces se numește **trunchiere** (*truncation*).

Este un termen pe care îl vom folosi și în capitolele despre loguri și administrarea fișierelor.

Prima demonstrație

Executăm:

```
echo "Prima linie" > jurnal.txt
```

Verificăm:

```
cat jurnal.txt
```

Rezultat:

```
Prima linie
```

Acum:

```
echo "A doua linie" > jurnal.txt
```

Rezultatul este:

```
A doua linie
```

Prima linie nu mai există.

Nu pentru că `echo` a șters-o.

Ci pentru că Bash a deschis fișierul în modul de suprascriere înainte de execuția lui `echo`.

Atunci de ce există `>>`?

În multe situații nu vrem să înlocuim informația.

Vrem să o păstrăm și să adăugăm informații noi.

Exemplu:

- jurnale de activitate;
- fișiere de log;
- rapoarte;
- audit;
- istoricul unei aplicații.

Pentru aceste cazuri există:

>>

Demonstrație

Executăm:

```
echo "Prima linie" > jurnal.txt
```

apoi:

```
echo "A doua linie" >> jurnal.txt
```

și:

```
echo "A treia linie" >> jurnal.txt
```

Acum:

```
cat jurnal.txt
```

afișează:

```
Prima linie  
A doua linie  
A treia linie
```

Observă că programul este același.

S-a schimbat doar modul în care Bash a deschis fișierul.

Un model mental

Imaginează-ți un caiet.

Operatorul > înseamnă:

Rupe toate paginile și începe un caiet nou.

Operatorul >> înseamnă:

Deschide caietul la ultima pagină și continuă să scrii.

Această analogie este simplă, dar descrie foarte bine diferența de comportament.

Caseta „În spatele cortinei”

Cine decide poziția în fișier?

Din nou, nu programul.

Shell-ul deschide fișierul în modul potrivit și îi oferă programului un descriptor deja poziționat.

În cazul lui `>`:

- poziția este la începutul fișierului;
- dacă fișierul există, acesta este trunchiat.

În cazul lui `>>`:

- poziția este la sfârșitul fișierului;
- conținutul existent este păstrat.

Programul continuă să scrie pe `stdout` fără să cunoască aceste detalii.

De ce este util acest mecanism?

Pentru că același program poate fi folosit în contexte complet diferite.

Exemplu:

`date`

Poate:

afișa ora pe ecran:

`date`

crea un fișier nou:

`date > raport.txt`

sau completa un jurnal:

`date >> raport.txt`

Autorul programului `date` nu a implementat trei moduri diferite.

Shell-ul este cel care schimbă conexiunea fluxului de ieșire.

Aceasta este încă o demonstrație a filozofiei Unix:

Programele produc date. Shell-ul decide unde ajung aceste date.

Greșeală din lumea reală

Un administrator dorește să păstreze istoricul unei copii de siguranță și scrie:

```
date > backup.log
```

în interiorul unui script executat zilnic.

După câteva săptămâni observă că fișierul conține doar ultima dată.

Problema nu este în script.

Problema este alegerea operatorului.

În acest caz:

```
date >> backup.log
```

era alegerea potrivită.

Administrator Notes

Întreabă-te întotdeauna:

Vreau să înlocuiesc informația sau vreau să o păstrez?

Răspunsul la această întrebare îți spune imediat dacă trebuie să folosești `>` sau `>>`.

Această alegere pare minoră, dar în practică poate însemna diferența dintre păstrarea istoricului și pierderea lui.

Cum aş depana această problemă?

Problema:

Fișierul meu de log conține doar ultima execuție.

Pașii:

1. Verific dacă scriptul folosește `>` sau `>>`.
2. Confirm dacă intenția este suprascrierea sau completarea.
3. Testez manual cele două variante pe un fișier temporar.
4. Verific dacă aplicația însăși nu rescrie fișierul.

În multe situații, problema se reduce la alegerea operatorului de redirectionare.

Laborator 4.3

Obiectiv

Înțelegerea diferenței dintre `>` și `>>`.

Cerințe

1. Creează un fișier:

```
echo "Linia 1" > test.log
```

2. Adaugă o nouă linie folosind:

```
echo "Linia 2" >> test.log
```

3. Verifică rezultatul:

```
cat test.log
```

4. Repetă ultimul pas folosind `>` în loc de `>>`.
 5. Explică diferența observată.
-

Soluția laboratorului

- `>` creează fișierul sau îl deschide prin eliminarea conținutului existent.
 - `>>` păstrează conținutul și adaugă informația nouă la sfârșit.
 - Alegerea operatorului depinde de scopul urmărit: înlocuire sau completare.
-

Ce trebuie să reții

- `>` și `>>` modifică modul în care Bash deschide fișierul, nu modul în care funcționează programul.
- `>` produce trunchierea fișierului dacă acesta există.
- `>>` păstrează conținutul existent și adaugă informații noi.
- Alegerea corectă a operatorului este esențială în scripturile care generează loguri sau rapoarte.

Unix separă responsabilitățile în mod riguros. Programul produce informația. Shell-ul decide destinația ei. Această separare permite reutilizarea aceluiași programe în nenumărate contexte fără modificarea codului lor. Este unul dintre motivele pentru care utilitare aparent simple, precum `echo`, `cat` sau `date`, pot participa la construcția unor sisteme complexe.

Capitolul 4

4.4 Standard Error – De ce mesajele de eroare urmează un drum separat

„Una dintre cele mai inspirate decizii din Unix a fost separarea rezultatelor normale de mesajele de eroare. Această separare permite automatizarea fără a pierde informațiile necesare diagnosticării.”

Introducere

În capitolele precedente am învățat că fiecare proces pornește cu trei fluxuri standard:

Descriptor	Nume	Rol
0	stdin	Date de intrare
1	stdout	Rezultate normale
2	stderr	Mesaje de eroare

Până acum am lucrat aproape exclusiv cu **stdout**.

Dar apare o întrebare firească.

De ce avem nevoie de un al doilea flux de ieșire?

Nu ar fi fost suficient unul singur?

Răspunsul îl vom descoperi prin câteva exemple simple.

Primul experiment

Executăm:

```
ls
```

Rezultatul poate fi:

```
documente  
imagini  
script.sh
```

Tot ceea ce vedem reprezintă **ieșirea normală** a programului.

Acum executăm:

```
ls fisier-inexistent
```

Rezultatul este:

```
ls: cannot access 'fisier-inexistent': No such file or directory
```

Și această informație apare pe ecran.

La prima vedere pare același lucru.

Dar nu este.

În primul caz, `ls` a scris pe **stdout**.

În al doilea, a scris pe **stderr**.

Terminalul afișează ambele fluxuri în același loc, motiv pentru care diferența nu este încă vizibilă.

De ce nu observăm diferența?

Pentru că, în mod implicit, Bash conectează atât **stdout**, cât și **stderr** la același terminal.

Putem reprezenta situația astfel:

```
          +-----+
stdout  -----> |
          | Terminal |
stderr  -----> |
          +-----+
```

Deși cele două fluxuri ajung în același loc, ele rămân independente.

Această independență va deveni extrem de importantă în capitolele următoare.

Un exemplu din viața reală

Presupunem că executăm:

```
find /etc -name "*.conf"
```

Rezultatul conține două tipuri de informații.

Pe de o parte:

```
/etc/ssh/sshd_config
/etc/fstab
/etc/...
```

Acestea sunt datele pe care le căutăm.

Pe de altă parte, dacă nu avem acces la anumite directoare, putem vedea mesaje precum:

```
Permission denied
```


Aceste mesaje nu reprezintă rezultatul căutării.

Ele reprezintă informații de diagnostic.

Din acest motiv sunt trimise pe **stderr**.

De ce este această separare atât de importantă?

Imaginează-ți că vrem să procesăm automat lista fișierelor găsite.

Dacă mesajele de eroare ar fi amestecate printre rezultate, un alt program nu ar mai putea distinge cu ușurință:

- informația utilă;
- informația de diagnostic.

Separarea fluxurilor rezolvă elegant această problemă.

Caseta „În spatele cortinei”

Date versus diagnostic

Aceasta este o idee pe care merită să o reținem.

stdout transportă informația pe care programul promite să o producă.

stderr transportă informația despre modul în care programul și-a desfășurat execuția.

Această distincție pare subtilă.

În practică este fundamentală.

Demonstrație

Executăm:

```
grep root /etc/passwd
```

Rezultatul:

```
root:x:0:0:root:/root:/bin/bash
```

Acum:

```
grep root fisier-care-nu-exista
```

Rezultatul:

```
grep: fisier-care-nu-exista: No such file or directory
```

În primul caz avem date.

În al doilea avem diagnostic.

Aceasta este diferența dintre cele două fluxuri.

Greșeală din lumea reală

Un administrator construiește un script care analizează ieșirea unei comenzi.

Scriptul funcționează perfect luni de zile.

Într-o zi apare o eroare de permisiuni.

Scriptul începe să proceseze mesajele de eroare ca și cum ar fi rezultate valide.

Problema nu este scriptul.

Problema este că autorul nu a ținut cont de existența celor două fluxuri.

Administrator Notes

Ori de câte ori scrii un script care prelucrează rezultatul unei comenzi, pune-ți o întrebare simplă:

Procesez datele produse de program sau și mesajele sale de diagnostic?

Foarte multe bug-uri provin din ignorarea acestei diferențe.

Cum aş depana această problemă?

Problema:

Scriptul meu citește informații pe care nu mă așteptam să le primească.

Pașii:

1. Verific dacă programul produce și mesaje pe stderr.
2. Testez comanda într-un terminal și observ dacă apar mesaje de eroare.
3. Identific dacă problema este în datele produse sau în diagnosticul execuției.
4. Abia apoi decid dacă este necesară o redirectionare.

Observă că încă nu discutăm sintaxa.

Întâi înțelegem problema.

Laborator 4.4

Obiectiv

Diferențierea dintre `stdout` și `stderr`.

Cerințe

Execută următoarele comenzi:

```
ls
```

```
ls fisier-inexistent
```

```
grep root /etc/passwd
```

```
grep root fisier-inexistent
```

Pentru fiecare comandă răspunde:

1. Care este rezultatul normal?
2. Există un mesaj de diagnostic?
3. Pe ce flux consideri că a fost trimis fiecare?

Nu folosi încă redirectionări.

Scopul laboratorului este observarea comportamentului.

Soluția laboratorului

- `ls` și `grep` folosesc **`stdout`** pentru rezultatele normale.
 - Atunci când apare o problemă (fișier inexistent, lipsă permisiuni etc.), mesajele sunt trimise pe **`stderr`**.
 - Terminalul afișează ambele fluxuri, dar ele rămân distincte.
-

Ce trebuie să reții

- Nu orice text afișat pe ecran provine din același flux.
- **`stdout`** și **`stderr`** sunt două canale independente.

- Separarea lor permite automatizarea și depanarea eficientă.
- Înainte de a învăța cum se redirecționează aceste fluxuri, este important să înțelegi de ce există.

Capitolul 4

4.5 Redirecționarea lui Standard Error – Cum separăm rezultatele de mesajele de eroare

„Până acum am observat că rezultatele normale și mesajele de eroare urmează căi diferite. În acest capitol vom învăța cum îi putem spune shell-ului să schimbe destinația fluxului de eroare, fără a afecta fluxul de rezultate.”

Introducere

În capitolul anterior am descoperit că există două fluxuri de ieșire:

- **stdout** (descriptorul 1)
- **stderr** (descriptorul 2)

În mod implicit, ambele ajung pe terminal.

Dar dacă dorim să păstrăm separat mesajele de eroare?

Sau să le analizăm ulterior?

Sau să construim un jurnal de diagnostic?

Pentru aceste situații Bash oferă posibilitatea redirecționării fluxului **stderr**.

Primul experiment

Executăm:

```
ls fisier-inexistent
```

Rezultatul:

```
ls: cannot access 'fisier-inexistent': No such file or directory
```

Acum încercăm:

```
ls fisier-inexistent > rezultat.txt
```

Mulți cursanți se așteaptă ca mesajul să ajungă în fișier.

Dar nu se întâmplă.

Mesajul apare în continuare pe terminal.

De ce?

Pentru că operatorul `>` redirecționează doar **stdout**.

În cazul nostru, `ls` nu a produs niciun rezultat normal.

A produs doar un mesaj pe **stderr**.

Cum îi spunem shell-ului că vrem să redirecționeze stderr?

Prin folosirea descriptorului numeric al fluxului.

Am învățat deja:

Flux **Descriptor**

stdin 0

stdout 1

stderr 2

De aceea putem scrie:

```
ls fisier-inexistent 2> erori.txt
```

Acum Bash procedează astfel:

1. Deschide fișierul `erori.txt`.
2. Conectează descriptorul 2 la acel fișier.
3. Abia apoi execută `ls`.

Rezultatul este interesant:

Terminalul nu mai afișează mesajul de eroare.

Dacă verificăm:

```
cat erori.txt
```

vom găsi:

```
ls: cannot access 'fisier-inexistent': No such file or directory
```

Observă un detaliu important

Comanda:

```
ls fisier-inexistent 2> erori.txt
```

nu modifică programul `ls`.

`ls` continuă să scrie pe descriptorul 2.

Doar că acum descriptorul 2 nu mai este conectat la terminal.

Este conectat la un fișier.

Aceasta este o idee care merită repetată de multe ori în carte:

Programele nu știu unde ajung datele lor. Ele scriu în descriptori. Shell-ul decide destinația descriptorilor.

Demonstrație practică

Să executăm două comenzi:

```
ls /etc 2> erori.txt
```

și

```
ls fisier-inexistent 2> erori.txt
```

În primul caz:

- fișierul `erori.txt` rămâne gol.

În al doilea:

- conține mesajul de eroare.

Această demonstrație arată că `stderr` este folosit numai atunci când există informații de diagnostic.

Caseta „În spatele cortinei”

De ce se scrie `2>` și nu doar `>`?

Pentru că `>` este, de fapt, o prescurtare.

În mod implicit:

```
>
```

înseamnă:

1>

adică:

redirecționează descriptorul 1 (stdout).

Atunci când dorim un alt descriptor, îl precizăm explicit.

Astfel:

2>

înseamnă:

redirecționează descriptorul 2 (stderr).

Această explicație îi ajută pe cititori să nu memoreze două simboluri diferite, ci să înțeleagă că este același mecanism aplicat unor descriptori diferiți.

Greșeală din lumea reală

Un administrator execută:

```
find / -name "*.conf" > rezultate.txt
```

și presupune că toate informațiile sunt acum în fișier.

Mai târziu observă că pe terminal au apărut sute de mesaje:

```
Permission denied
```

Nu este un bug.

Este comportamentul normal.

Doar stdout a fost redirecționat.

stderr a rămas conectat la terminal.

Administrator Notes

Nu redirecționa automat stderr doar pentru că poți.

Mesajele de eroare conțin adesea informații importante despre probleme reale.

În multe situații este mai util să le vezi pe ecran decât să le ascunzi într-un fișier.

Întreabă-te întotdeauna:

Vreau să păstrez aceste informații pentru analiză sau vreau să le observ imediat?

Cum aş depana această problemă?

Problema:

Mesajele de eroare nu ajung în fişierul în care am redirecţionat rezultatele.

Paşii:

1. Verific dacă am folosit doar >.
 2. Îmi amintesc că > afectează numai stdout.
 3. Dacă vreau mesajele de eroare, folosesc 2>.
 4. Verific apoi conţinutul fişierului de erori.
-

Laborator 4.5

Obiectiv

Separarea rezultatelor normale de mesajele de eroare.

Cerinţe

Execută:

```
ls /etc > rezultate.txt
```

Verifică:

```
cat rezultate.txt
```

Apoi:

```
ls fisier-inexistent 2> erori.txt
```

Verifică:

```
cat erori.txt
```

La final execută:

```
ls /etc
ls fisier-inexistent
```

şi compară comportamentul cu situaţia în care ai folosit redirecţionările.

Soluția laboratorului

- `>` redirecționează fluxul **stdout**.
 - `2>` redirecționează fluxul **stderr**.
 - Cele două fluxuri sunt independente și pot avea destinații diferite.
 - Shell-ul realizează conexiunile înainte ca programul să înceapă execuția.
-

Ce trebuie să reții

- `>` este echivalent cu `1>`.
- `2>` redirecționează fluxul de eroare.
- Programele nu trebuie modificate pentru a schimba destinația fluxurilor.
- Separarea dintre `stdout` și `stderr` este una dintre cele mai puternice idei din arhitectura Unix.

Capitolul 4

4.6 Citirea din fișiere – Prima întâlnire cu operatorul `<`

„Dacă operatorul `>` schimbă destinația ieșirii standard, atunci este firesc să ne întrebăm dacă putem schimba și sursa din care un program citește date. Răspunsul este da, iar operatorul folosit este `<`.”

Introducere

Până acum am schimbat destinația fluxului **stdout**.

Am văzut că același program poate scrie:

- pe terminal;
- într-un fișier.

Dar apare întrebarea firească:

Poate un program să citească dintr-un fișier fără să știe că acel fișier există?

Răspunsul este unul dintre cele mai elegante concepte din Unix.

Da.

Primul experiment

Executăm:

```
cat
```

Programul așteaptă date.

Scriem:

```
Linux
```

Apoi:

```
Bash
```

La fiecare apăsare de **Enter**, `cat` afișează imediat ceea ce primește.

De unde citește?

De la tastatură.

Mai exact:

din **stdin**.

Acum schimbăm sursa

Creăm un fișier:

```
echo "Linux" > curs.txt  
echo "Bash" >> curs.txt  
echo "LPIC" >> curs.txt
```

Conținutul este:

```
Linux  
Bash  
LPIC
```

Acum executăm:

```
cat < curs.txt
```

Rezultatul este:

```
Linux  
Bash  
LPIC
```

Observă ceva foarte important.

Nu i-am spus lui `cat`:

„Deschide fișierul `curs.txt`.”

I-am spus shell-ului:

„Conectează `stdin` la fișierul `curs.txt`.”

Ce face Bash?

Intern putem imagina următoarele etape.

În mod normal:

Tastatură
↓
`stdin (0)`
↓
`cat`

Cu operatorul `<`:

`curs.txt`
↓
`stdin (0)`
↓
`cat`

Programul `cat` nu știe dacă citește:

- de la tastatură;
- dintr-un fișier;
- dintr-un pipe.

El citește pur și simplu din descriptorul `0`.

Aceasta este frumusețea modelului Unix.

O observație interesantă

Probabil ai observat că putem scrie și:

```
cat curs.txt
```

Rezultatul este același.

Atunci apare întrebarea:

De ce mai există <?

Răspunsul este simplu.

cat știe să deschidă fișiere.

Dar multe programe nu primesc numele fișierului ca argument.

Ele citesc exclusiv din stdin.

Pentru acelea, operatorul < este esențial.

Vom vedea astfel de exemple în capitolele dedicate filtrelor și scripturilor.

Un exemplu foarte practic

Presupunem că avem un program care citește comenzile de la tastatură.

În loc să tastăm manual fiecare linie, putem pregăti un fișier și îl putem folosi ca intrare.

Această idee stă la baza:

- automatizării;
 - testării scripturilor;
 - procesării în lot (*batch processing*).
-

Caseta „În spatele cortinei”

Cine deschide fișierul?

Exact ca în cazul operatorului >.

Nu programul.

Shell-ul.

Ordinea este:

1. Bash deschide fișierul.
2. Conectează descriptorul 0 la acel fișier.
3. Execută programul.

Dacă fișierul nu poate fi deschis, programul nici măcar nu pornește.

Observă cât de consecvent este modelul.

Este același lucru pe care l-am învățat pentru redirecționarea ieșirii.

Greșeală din lumea reală

Un administrator execută:

```
cat < fisier-care-nu-exista
```

și primește:

```
No such file or directory
```

El presupune că eroarea aparține lui `cat`.

În realitate, `cat` nu a început execuția.

Shell-ul nu a reușit să deschidă fișierul pentru `std`in.

Acesta este exact același mecanism pe care l-am întâlnit la `>`.

Administrator Notes

Atunci când folosești operatorul `<`, întreabă-te:

Programul citește dintr-un argument sau din `std`in?

Nu toate utilitățile folosesc același model.

Înțelegerea acestei diferențe te ajută să alegi forma corectă a comenzii și să înțelegi documentația (`man`) mai ușor.

Cum aș depana această problemă?

Problema:

Programul nu citește datele pe care mă așteptam să le proceseze.

Pașii:

1. Verific dacă programul citește din `std`in sau din argumente.
 2. Confirm existența și permisiunile fișierului folosit cu `<`.
 3. Testez mai întâi programul cu intrare de la tastatură.
 4. Apoi repet testul folosind redirectionarea.
-

Laborator 4.6

Obiectiv

Înțelegerea redirectionării fluxului de intrare.

Cerințe

1. Creează fișierul:

```
echo "Linux" > curs.txt  
echo "Bash" >> curs.txt  
echo "LPIC" >> curs.txt
```

2. Rulează:

```
cat curs.txt
```

3. Rulează:

```
cat < curs.txt
```

4. Compară rezultatele.

5. Încearcă:

```
cat < fisier-inexistent
```

și explică de ce apare eroarea.

Soluția laboratorului

- < redirectionează **stdin**.
 - Shell-ul deschide fișierul înainte de lansarea programului.
 - Programul citește din descriptorul 0 fără să știe dacă sursa este tastatura, un fișier sau alt mecanism.
 - Dacă fișierul nu poate fi deschis, programul nu este executat.
-

Ce trebuie să reții

- < schimbă sursa fluxului **stdin**.
- Bash realizează redirectionarea înainte de execuția programului.
- Programele bine proiectate citesc din descriptorul 0 și rămân independente de sursa datelor.
- Acest mecanism face posibilă automatizarea fără modificarea programelor.

Observă simetria:

	Flux	Implicit	Redirecționare
stdin (0)	Tastatură	<	
stdout (1)	Terminal	> / >>	
stderr (2)	Terminal	2>	

Capitolul 5

Pipe-uri – Cum cooperează programele în Unix

„Un program bun rezolvă bine o singură problemă. Un sistem bun permite acestor programe să colaboreze. Pipe-ul este mecanismul prin care Unix transformă multe programe mici într-o soluție mare.”

Introducere

Până acum am învățat:

- cum un program citește din **stdin**;
- cum scrie pe **stdout**;
- cum trimite mesajele de eroare pe **stderr**;
- cum shell-ul poate conecta aceste fluxuri la fișiere.

Dar apare o întrebare fascinantă.

Dacă stdout-ul unui program poate fi conectat la un fișier, poate fi conectat și la stdin-ul altui program?

Răspunsul este:

Da.

Și exact acesta este rolul operatorului:

|

numit **pipe**.

O idee care a schimbat lumea Unix

Să presupunem că avem un program care produce 10.000 de linii.

Avem apoi un alt program care știe să sorteze texte.

Și un al treilea care știe să numere liniile.

În loc să scriem un program uriaș care să facă toate aceste lucruri, Unix spune:

Lasă fiecare program să facă un singur lucru foarte bine. Apoi conectează-le.

Aceasta este una dintre cele mai importante idei din întreaga filozofie Unix.

Primul experiment

Executăm:

```
ls
```

Rezultatul apare pe ecran.

Acum executăm:

```
ls | cat
```

Rezultatul pare identic.

Mulți cursanți întreabă imediat:

"Atunci la ce folosește pipe-ul?"

Este o întrebare excelentă.

Ce se întâmplă în realitate?

În primul caz:

```
ls
|
▼
stdout
|
▼
Terminal
```

În al doilea:

```
ls
|
▼
```



```
stdout
|
▼
stdin
|
cat
|
▼
stdout
|
▼
Terminal
```

Rezultatul este același.

Dar traseul datelor este complet diferit.

Acesta este primul lucru pe care trebuie să-l înțeleagă cititorul.

De ce funcționează?

Pentru că `cat` nu știe cine produce datele.

El știe doar să citească din `stdin`.

În primul capitol despre `stdin` spuneam:

Programul nu știe dacă citește de la tastatură, dintr-un fișier sau din alt program.

Acum vedem și ultima variantă.

`cat` citește din `stdout`-ul lui `ls`.

Fără să știe acest lucru.

Un exemplu mai interesant

Executăm:

```
ls | wc -l
```

Rezultatul poate fi:

```
42
```

Ce s-a întâmplat?

```
ls
```

produce lista fișierelor.

↓

`wc -l`

numără liniile.

Niciunul dintre cele două programe nu știe că există celălalt.

Shell-ul le conectează.

Caseta „În spatele cortinei”

Cine creează pipe-ul?

Nu `ls`.

Nu `wc`.

Ci `Bash`.

Ordinea este următoarea:

1. `Bash` creează un pipe în kernel.
2. Pornește primul proces și conectează `stdout` la capătul de scriere al pipe-ului.
3. Pornește al doilea proces și conectează `stdin` la capătul de citire al pipe-ului.
4. Cele două procese rulează simultan.

Acesta este un detaliu foarte important.

Pipe-ul nu este implementat de programe.

Este implementat de sistemul de operare și orchestrat de shell.

Un model mental

Imaginează-ți două persoane.

Prima produce bilețele.

A doua citește bilețelele.

Între ele există un tub.

Prima persoană introduce bilețelele în tub.

A doua le scoate și le citește.

Niciuna nu trebuie să știe cine este cealaltă.

Trebuie doar să respecte aceeași regulă:

una scrie,

cealaltă citește.

Exact aceasta este ideea pipe-ului.

Greșeală din lumea reală

Un administrator execută:

```
cat fisier.log | grep ERROR
```

Funcționează.

Mai târziu află că putea scrie:

```
grep ERROR fisier.log
```

Și se întreabă:

Care variantă este mai bună?

Răspunsul nu este:

"Prima este greșită."

Răspunsul este:

În acest caz, `grep` știe deja să citească fișiere.

Pipe-ul nu aduce niciun avantaj.

Dar există situații în care programul din dreapta citește exclusiv din `stdin`.

Atunci pipe-ul devine indispensabil.

Aceasta este o nuanță importantă și îi învață pe cursanți să aleagă instrumentul potrivit, nu doar să reproducă modele.

Administrator Notes

Întreabă-te întotdeauna:

Programul din dreapta poate primi date ca argument sau citește în mod natural din `stdin`?

Răspunsul te ajută să alegi între:

```
grep text fisier.txt
```

și

```
cat fisier.txt | grep text
```

Nu există un răspuns universal.

Există un răspuns potrivit contextului.

Cum aş depana această problemă?

Problema:

Pipe-ul nu produce rezultatul aşteptat.

Paşii:

1. Testez separat prima comandă.
2. Testez separat a doua comandă.
3. Verific dacă prima produce date pe stdout.
4. Verific dacă a doua citeşte din stdin.
5. Abia apoi investighez pipe-ul.

Această metodă simplifică foarte mult depanarea lanţurilor lungi de comenzi.

Laborator 5.1

Obiectiv

Înţelegerea funcţionării unui pipe.

Cerinţe

Execută:

```
ls
```

Apoi:

```
ls | cat
```

Compară rezultatele.

Execută:

```
ls | wc -l
```

Apoi:

```
ls | sort
```

În fiecare caz răspunde:

1. Care program produce datele?
 2. Care program le consumă?
 3. Cine creează conexiunea dintre ele?
-

Soluția laboratorului

- `ls` produce informația pe `stdout`.
 - `cat`, `wc` și `sort` citesc din `stdin`.
 - Bash creează pipe-ul și conectează cele două procese înainte de execuția lor.
-

Ce trebuie să reții

- Pipe-ul conectează `stdout`-ul unui proces la `stdin`-ul altui proces.
- Programele rămân independente unele de altele.
- Shell-ul creează și configurează conexiunile.
- Pipe-ul este unul dintre mecanismele fundamentale care permit compunerea utilităților Unix în soluții complexe.

Capitolul 5

5.2 Pipe-urile conectează procese, nu comenzi

„Atunci când scriem două comenzi separate prin caracterul `|`, avem impresia că shell-ul le execută una după alta. În realitate, el creează două procese independente care rulează în același timp și comunică printr-un canal de date.”

Introducere

Până acum am privit pipe-ul ca pe un mecanism prin care rezultatul unei comenzi devine intrarea alteia.

Această descriere este corectă.

Dar este incompletă.

Pentru a înțelege cu adevărat pipe-urile trebuie să răspundem la o întrebare:

Ce conectează, de fapt, operatorul `|` ?

Răspunsul este:

Nu conectează comenzi.

Conectează procese.

Această diferență pare subtilă.

În realitate, este una dintre ideile fundamentale ale sistemelor Unix.

Un exemplu simplu

Executăm:

```
ls | wc -l
```

La prima vedere pare că:

1. rulează `ls`;
2. apoi rulează `wc`.

Dar nu așa funcționează.

Bash creează două procese.

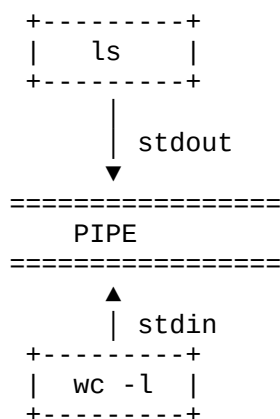
Unul execută `ls`.

Celălalt execută `wc`.

Cele două rulează aproape simultan.

Cum arată acest lucru?

Putem reprezenta situația astfel:



Observă că pipe-ul nu este parte din `ls`.

Și nici din `WC`.

El este creat de shell și există între cele două procese.

O întrebare firească

Mulți cursanți întreabă:

„Dar dacă `ls` încă nu a terminat, cum poate `WC` să înceapă să numere?”

Aceasta este o întrebare excelentă.

Răspunsul este:

Nu trebuie să aștepte.

Pe măsură ce `ls` produce date, acestea sunt introduse în pipe.

În același timp, `WC` citește datele deja disponibile.

Cele două procese lucrează în paralel.

Aceasta este una dintre marile forțe ale pipe-urilor.

O analogie

Imaginează-ți o bandă transportoare într-o fabrică.

Prima persoană pune cutii pe bandă.

A doua le numără.

Nu este nevoie ca prima să termine toate cutiile înainte ca a doua să înceapă.

În momentul în care apare prima cutie, a doua persoană poate începe deja să lucreze.

Pipe-ul funcționează exact la fel.

Caseta „În spatele cortinei”

Pipe-ul are un buffer

Kernelul nu transmite fiecare caracter imediat.

Între procese există o zonă de memorie numită **buffer**.

Procesul care produce date scrie în acest buffer.

Procesul care consumă date citește din el.

Dacă bufferul se umple, procesul producător este suspendat temporar până când consumatorul eliberează spațiu.

Dacă bufferul este gol, consumatorul așteaptă apariția datelor.

Acest mecanism permite cooperarea eficientă dintre procese fără ca programele să fie conștiente una de cealaltă.

Notă: nu este nevoie să discutăm aici dimensiunea bufferului sau detalii dependente de implementarea kernelului. Acestea sunt subiecte pentru LPIC-2 sau pentru programare de sistem.

Greșeală din lumea reală

Un cursant afirmă:

„Pipe-ul execută mai întâi prima comandă și apoi a doua.”

Aceasta este o idee intuitivă, dar incorectă.

În realitate, cele două procese rulează concurent.

Această diferență explică multe comportamente observate în practică, mai ales când lucrăm cu volume mari de date.

Administrator Notes

Când construiești un lanț de pipe-uri, gândește-te la el ca la o linie de producție.

Fiecare program are un rol clar:

- unul produce informația;
- altul o filtrează;
- altul o transformă;
- ultimul o afișează sau o salvează.

Această abordare conduce, de regulă, la comenzi mai simple și mai ușor de întreținut decât un singur script foarte complex.

Cum aş depana această problemă?

Problema:

Un lanț de pipe-uri nu produce rezultatul așteptat.

Metodă:

Nu analiza lanțul ca un întreg.

Desparte-l.

Exemplu:

A | B | C | D

Testează mai întâi:

A

Apoi:

A | B

După aceea:

A | B | C

Și abia la final:

A | B | C | D

Această metodă este folosită frecvent de administratorii experimentați deoarece izolează rapid locul în care apare problema.

Laborator 5.2

Obiectiv

Înțelegerea faptului că pipe-ul conectează procese care rulează concurent.

Cerințe

Execută:

`yes | head -5`

Observă că:

- `yes` ar putea produce un număr nelimitat de linii;
- `head` citește doar primele cinci.

Răspunde:

1. De ce se oprește comanda?
2. Cine decide că sunt suficiente cinci linii?
3. Ce se întâmplă cu procesul `yes` după ce `head` termină?

Notă : răspunsul complet despre semnalul SIGPIPE îl vom oferi într-un capitol dedicat semnalelor. Aici este suficient ca cititorul să observe comportamentul și să înțeleagă ideea generală.

Soluția laboratorului

- `head` citește doar numărul de linii solicitat.
- După închiderea capătului de citire al pipe-ului, procesul producător nu mai poate continua să scrie.
- Kernelul și mecanismele de comunicare dintre procese gestionează această situație automat.

Nu este necesar ca utilizatorul să intervină.

Ce trebuie să reții

- Pipe-ul conectează procese, nu simple comenzi.
- Procesele rulează concurrent și schimbă date printr-un canal creat de shell și gestionat de kernel.
- Datele circulă printr-un buffer, ceea ce permite cooperarea eficientă dintre procese.
- La depanare, un lanț de pipe-uri se analizează etapă cu etapă.

Capitolul 5

5.3 Filtrele Unix – Programele care transformă date

„Puterea Unix nu vine din existența unor programe foarte complexe, ci din existența multor programe mici care pot primi date, le pot transforma și le pot transmite mai departe.”

Introducere

În capitolul anterior am învățat că un pipe conectează două procese.

Acum apare o întrebare firească.

Ce fel de programe sunt cele care se află între două pipe-uri?

În Unix există o categorie specială de programe numite **filtre** (*filters*).

Un filtru are un comportament foarte simplu.

1. Citește date din **stdin**.
2. Le prelucrează.
3. Trimite rezultatul pe **stdout**.

Nu îl interesează de unde provin datele.

Nu îl interesează unde vor ajunge după prelucrare.

Acest lucru îl face extrem de reutilizabil.

Primul exemplu

Executăm:

```
sort
```

Programul așteaptă introducerea unor linii.

Scriem:

```
mere  
pere  
caise
```

Apăsăm **Ctrl+D** pentru a semnaliza sfârșitul intrării.

Rezultatul este:

```
caise  
mere  
pere
```

Observă ceva important.

`sort` nu a citit dintr-un fișier.

A citit din **stdin**.

Acesta este comportamentul tipic al unui filtru.

Același program, altă sursă

Executăm:

```
cat fructe.txt | sort
```

Acum `sort` citește tot din `stdin`.

Dar `stdin` nu mai este tastatura.

Este pipe-ul creat de shell.

Programul nici măcar nu observă diferența.

Aceasta este una dintre cele mai elegante idei din Unix.

Ce este, de fapt, un filtru?

Un filtru poate fi privit ca o stație de prelucrare.

Primește materie primă.

Produce un rezultat.

Nu știe cine produce materia primă.

Nu știe cine consumă rezultatul.

Exact ca într-o linie industrială.

Exemple de filtre

Pe parcursul manualului vom studia în detaliu multe dintre ele.

Printre cele mai utilizate se numără:

Program	Rol principal
<code>sort</code>	Sortează liniile de text
<code>uniq</code>	Elimină sau identifică liniile duplicate consecutive
<code>grep</code>	Selectează liniile care corespund unui model
<code>cut</code>	Extrage coloane sau câmpuri
<code>tr</code>	Înlocuiește sau elimină caractere
<code>wc</code>	Numără linii, cuvinte și caractere

Nu este nevoie să le cunoaștem acum în detaliu.

Important este să înțelegem că toate respectă același model.

O analogie

Imaginează-ți o fabrică de reciclare.

Prima stație sortează materialele.

A doua elimină impuritățile.

A treia le compactează.

Fiecare stație are o singură responsabilitate.

Împreună formează un proces complex.

Unix funcționează la fel.

Caseta „În spatele cortinei”

De ce există atât de multe programe mici?

La început poate părea mai comod să existe un singur program uriaș care să facă totul.

Creatorii Unix au ales altă cale.

Au preferat programe mici, fiecare foarte bun la o singură sarcină.

Avantajele sunt importante:

- programele sunt mai ușor de testat;
- pot fi reutilizate în combinații foarte diferite;
- sunt mai simple de întreținut;
- pot fi combinate prin pipe-uri fără modificări.

Această filozofie este cunoscută sub numele de:

„Do one thing, and do it well.”

Nu este doar un slogan.

Este unul dintre principiile care au influențat proiectarea întregului ecosistem Unix.

Greșeală din lumea reală

Un administrator scrie un script de sute de linii pentru a filtra și transforma un fișier text.

Mai târziu descoperă că aceeași problemă putea fi rezolvată prin combinarea a trei filtre standard.

Nu orice problemă necesită un script nou.

De multe ori soluția există deja și constă în alegerea și conectarea corectă a unor utilitare existente.

Administrator Notes

Înainte să începi să scrii un script lung, întreabă-te:

Există deja un filtru Unix care rezolvă o parte din problemă?

Foarte des răspunsul este „da”.

Un administrator experimentat încearcă mai întâi să combine utilitățile existente și abia apoi scrie cod nou.

Cum aş depana această problemă?

Problema:

Lanțul meu de filtre produce un rezultat greșit.

Metodă:

Nu analiza întregul lanț.

Verifică fiecare filtru separat.

De exemplu:

A | B | C | D

Testează:

A

Apoi:

A | B

Apoi:

A | B | C

Astfel poți identifica rapid filtrul care modifică datele într-un mod neașteptat.

Laborator 5.3

Obiectiv

Observarea comportamentului unui filtru.

Cerințe

1. Creează fișierul:

```
cat > fructe.txt
```

Introdu:

```
pere  
mere  
caise  
prune
```

Încheie introducerea cu **Ctrl+D**.

- Sortează conținutul:

```
sort < fructe.txt
```

- Sortează folosind un pipe:

```
cat fructe.txt | sort
```

- Compară rezultatele.
- Explică de ce `sort` funcționează în ambele cazuri.

Soluția laboratorului

- `sort` citește din `stdin`.
- În primul caz, `stdin` este redirecționat din fișier.
- În al doilea caz, `stdin` provine din pipe.
- Programul funcționează identic deoarece folosește aceeași interfață de intrare.

Ce trebuie să reții

- Un filtru citește din `stdin` și scrie pe `stdout`.
- Filtrele pot fi combinate cu ușurință folosind pipe-uri.
- Programele nu trebuie să cunoască sursa datelor sau destinația rezultatului.
- Filosofia Unix favorizează compunerea unor programe simple în locul construirii unor aplicații monolitice.

Întrebare de la un cursant

„Dacă `sort` poate primi și un nume de fișier ca argument, de ce îl mai folosim împreună cu pipe-uri?”

Răspuns

Este o întrebare foarte bună.

Multe utilitare Unix sunt flexibile: pot citi atât din fișiere specificate ca argumente, cât și din **stdin**. Acest lucru le face ușor de folosit atât individual, cât și în lanțuri de prelucrare.

În schimb, există și programe care citesc exclusiv din **stdin** sau care primesc date generate dinamic de alte comenzi. În aceste situații, pipe-ul devine soluția naturală.

Ca administrator, este util să cunoști ambele variante și să alegi forma care exprimă cel mai clar intenția ta. Uneori `sort fisier.txt` este cea mai simplă alegere. Alteori comanda `| sort` este singura posibilă.

Capitolul 5

5.4 Construirea unui pipeline – Cum proiectăm lanțuri de comenzi

„Un administrator experimentat nu caută comanda perfectă. El construiește un flux de prelucrare în care fiecare program rezolvă o etapă clară a problemei.”

Introducere

Până acum am învățat că:

- un program produce date;
- alt program le poate primi printr-un pipe;
- filtrele transformă aceste date.

Acum trebuie să învățăm ceva nou.

Cum proiectăm un pipeline?

Aceasta este una dintre competențele care diferențiază un utilizator obișnuit de un administrator experimentat.

Nu porni de la comandă

Foarte mulți începători gândesc astfel:

Ce comandă rezolvă problema mea?

Administratorii experimentați gândesc altfel.

Ei întreabă:

1. Care este sursa datelor?
2. Ce transformări trebuie aplicate?
3. Care este rezultatul dorit?

Abia după aceea aleg comenzile.

Un exemplu simplu

Presupunem că dorim:

Să aflăm câte fișiere există într-un director.

Putem gândi astfel:

Pasul 1

Obțin lista fișierelor.

↓

```
ls
```

Pasul 2

Număr liniile.

↓

```
wc -l
```

Pipeline-ul devine:

```
ls | wc -l
```

Observă că nu am început cu pipe-ul.

Am început cu problema.

Un exemplu mai complex

Dorim:

Să afișăm utilizatorii conectați și să îi sortăm alfabetic.

Gândirea este:

Programul A produce lista.

↓

Programul B sortează.

Pipeline-ul:

who | sort

Nu este important acum să învățăm who.

Important este modul de construcție.

Regula unei singure responsabilități

Fiecare program din pipeline trebuie să aibă o singură responsabilitate.

De exemplu:

produce

↓

selectează

↓

transformă

↓

ordonează

↓

numără

↓

afișează

Această succesiune este mult mai ușor de înțeles decât o comandă foarte complicată care încearcă să facă totul simultan.

Caseta „În spatele cortinei”

De ce pipeline-urile sunt ușor de întreținut?

Pentru că fiecare etapă poate fi verificată independent.

Dacă rezultatul final este greșit, putem testa fiecare componentă.

Aceasta este o idee fundamentală și în ingineria software:

Descompune problemele complexe în componente simple și verificabile.

Unix aplică acest principiu încă din anii '70.

Un model mental

Imaginează-ți o uzină.

Prima secție spală materia primă.

↓

A doua o sortează.

↓

A treia o ambalează.

↓

Ultima o trimite către client.

Dacă apare o problemă, verifici fiecare secție separat.

Exact la fel se depanează și un pipeline.

Greșeală din lumea reală

Un administrator scrie o comandă foarte lungă.

După câteva zile nu mai înțelege nici el ce face.

Problema nu este lungimea.

Problema este lipsa unei structuri logice.

Un pipeline bine construit poate avea multe etape și totuși să fie ușor de urmărit dacă fiecare etapă are un scop clar.

Administrator Notes

Atunci când construiești un pipeline, încearcă să poți răspunde la întrebarea:

„Ce face această etapă și de ce este necesară?”

Dacă nu poți răspunde, probabil că pipeline-ul poate fi simplificat.

Cum aş depana această problemă?

Problema:

Pipeline-ul produce un rezultat incorect.

Metodă:

Să presupunem:

A | B | C | D | E

Testează:

A

Apoi:

A | B

Apoi:

A | B | C

Şi continuă până la capăt.

Nu încerca să înţelegi toate cele cinci etape simultan.

Izolarea problemei este aproape întotdeauna mai eficientă.

Laborator 5.4

Obiectiv

Învăţarea metodei de construire a unui pipeline.

Cerinţe

Construieşte următoarele pipeline-uri şi explică rolul fiecărei etape:

ls | wc -l

who | sort

cat fructe.txt | sort

Pentru fiecare:

1. Cine produce datele?
2. Cine le transformă?
3. Cine consumă rezultatul final?

Nu urmări doar ieșirea comenzii.

Explică rolul fiecărui program.

Soluția laboratorului

În toate exemplele există aceeași structură:

- primul program produce informația;
- următorul o transformă;
- ultimul produce rezultatul dorit.

Aceasta este structura de bază a aproape tuturor pipeline-urilor.

Ce trebuie să reții

- Un pipeline se proiectează pornind de la problemă, nu de la sintaxă.
 - Fiecare program ar trebui să aibă o responsabilitate clară.
 - Pipeline-urile sunt ușor de testat și întreținut deoarece fiecare etapă poate fi verificată separat.
 - Gândirea modulară este unul dintre principiile fundamentale ale Unix.
-

Întrebare de la un cursant

„Cum știu ce comandă trebuie pusă prima și care trebuie pusă după pipe?”

Răspuns

Aceasta este una dintre cele mai importante întrebări pe care și le pune orice începător.

Nu începe cu comenzile.

Începe cu fluxul datelor.

Întreabă-te:

1. **De unde provin datele?**
2. **Ce transformare vreau să aplic?**
3. **Care este rezultatul final?**

Prima comandă trebuie să producă datele.

A doua trebuie să știe să le prelucreze.

Dacă urmează o a treia comandă, ea va primi rezultatul celei de-a doua și îl va transforma mai departe.

Cu alte cuvinte, nu alegi ordinea comenzilor.

Urmezi traseul natural al datelor.

Capitolul 5

5.5 Pipeline-uri cu mai multe etape – Construirea unui flux de prelucrare

Ce vei învăța

În acest capitol vei descoperi cum pot colabora mai multe programe într-un singur lanț de prelucrare și de ce fiecare etapă are o responsabilitate bine definită.

Introducere

Până acum am folosit exemple precum:

```
ls | wc -l
```

sau

```
cat fructe.txt | sort
```

Aceste exemple conțin două programe.

În practică însă, un administrator Linux construiește frecvent lanțuri formate din trei, patru sau chiar mai multe programe.

Important este să nu privești întregul lanț ca pe o comandă foarte lungă.

Privește-l ca pe o succesiune de pași simpli.

Gândirea în etape

Să presupunem că dorim să aflăm câte fișiere cu extensia `.conf` există într-un director.

În loc să căutăm o comandă "magică", descompunem problema.

1. Obținem lista fișierelor.
2. Selectăm doar fișierele `.conf`.

3. Numărăm rezultatele.

Acesta este modul de gândire pe care vrem să îl formăm.

Nu:

"Ce comandă rezolvă problema?"

Ci:

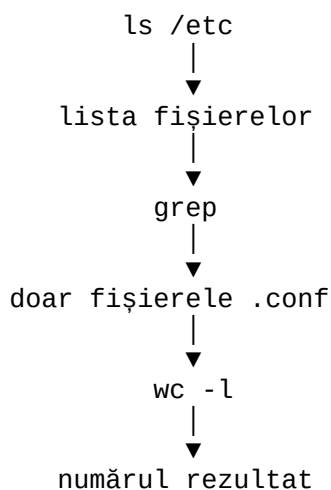
"Din ce pași este alcătuită problema?"

Primul exemplu

```
ls /etc | grep '\.conf$' | wc -l
```

Nu este important încă să înțelegem toate opțiunile lui `grep`.

Important este traseul datelor.



Observă că fiecare program primește exact ceea ce produce etapa anterioară.

Fiecare etapă are o singură responsabilitate

În exemplul anterior:

`ls`

produce informația.

↓

`grep`

selectează informația relevantă.

↓

WC

o transformă într-o valoare numerică.

Niciun program nu încearcă să facă tot procesul.

Aceasta este cheia simplității.

O analogie

Imaginează-ți o linie de asamblare.

Prima persoană despachetează produsele.

↓

A doua verifică dacă sunt complete.

↓

A treia le etichetează.

↓

A patra le împachetează.

Nimeni nu încearcă să facă toate operațiile.

Același principiu apare și în pipeline-urile Unix.

Caseta „În spatele cortinei”

De ce nu există o comandă care să facă totul?

Pentru că problemele reale sunt foarte diferite.

Dacă am construi un program gigantic care să știe toate combinațiile posibile de filtrare, sortare, numărare și transformare, acesta ar deveni foarte greu de întreținut.

Unix a ales o soluție mult mai flexibilă.

A creat programe mici și independente, pe care utilizatorul le poate combina după nevoie.

Această decizie explică de ce ecosistemul Unix a rămas atât de adaptabil timp de decenii.

Greșeală din lumea reală

Un administrator încearcă să construiască din prima un pipeline cu șase sau șapte etape.

Rezultatul nu este cel așteptat.

Încearcă să depaneze întreaga comandă simultan și pierde mult timp.

O metodă mult mai eficientă este să construiască lanțul progresiv.

Mai întâi:

```
ls /etc
```

Apoi:

```
ls /etc | grep '\.conf$'
```

Și abia după ce fiecare etapă produce rezultatul dorit, adaugă următoarea comandă.

Această metodă este folosită de foarte mulți administratori experimentați.

Administrator Notes

Construiește pipeline-urile exact cum ai construi un program.

Adaugă o singură etapă.

Verifică rezultatul.

Abia apoi continuă.

Este mult mai simplu să depanezi cinci pași mici decât o comandă foarte lungă scrisă dintr-o singură încercare.

Laborator 5.5

Obiectiv

Construirea progresivă a unui pipeline.

Cerințe

Execută succesiv:

```
ls /etc
```

```
ls /etc | grep '\.conf$'
```

```
ls /etc | grep '\.conf$' | wc -l
```

Pentru fiecare etapă răspunde:

1. Ce informație produce această comandă?
 2. Ce informație primește următoarea comandă?
 3. Ce rol are fiecare program în întregul lanț?
-

Soluția laboratorului

Prima comandă produce lista completă.

A doua selectează doar elementele care respectă criteriul de căutare.

Ultima transformă lista într-o valoare numerică.

Pipeline-ul nu trebuie privit ca o singură comandă.

El este alcătuit din mai multe transformări succesive ale aceleiași informații.

Ce trebuie să reții

- Un pipeline poate conține oricâte etape sunt necesare.
 - Fiecare program ar trebui să aibă o responsabilitate clară.
 - Construirea progresivă este mai eficientă decât scrierea unui lanț foarte lung dintr-o singură încercare.
 - Depanarea se face verificând fiecare etapă separat.
-

Întrebare de la un cursant

"Cum știu când să mă opresc și să mai adaug încă un pipe?"

Răspuns

Aceasta este o întrebare foarte bună și apare frecvent în practică.

Nu există un număr maxim de pipe-uri.

Există însă o regulă simplă:

Dacă o comandă începe să facă două lucruri diferite, întreabă-te dacă nu cumva este mai clar să adaugi încă o etapă.

Scopul nu este să ai cât mai puține pipe-uri.

Scopul este ca fiecare etapă să fie ușor de înțeles, verificat și modificat.

Capitolul 5

5.6 Metoda administratorului – Cum construim un pipeline fără să ghicim

Ce vei învăța

În acest capitol vei învăța o metodă practică prin care poți construi și verifica un pipeline pas cu pas, fără să încerci combinații la întâmplare.

Introducere

Una dintre cele mai frecvente greșeli ale începătorilor este aceasta:

Scriu dintr-o singură încercare un pipeline foarte lung.

Dacă rezultatul este greșit, modifică la întâmplare comenzile până când „pare să funcționeze”.

Această abordare consumă timp și produce frustrare.

Administratorii experimentați procedează diferit.

Ei construiesc pipeline-ul progresiv și verifică fiecare etapă înainte de a continua.

O metodă simplă în patru pași

Ori de câte ori ai de rezolvat o problemă folosind mai multe comenzi, urmează această ordine.

Pasul 1. Găsește sursa datelor

Întreabă-te:

De unde provin informațiile?

Poate fi:

- un fișier;
- rezultatul unei comenzi;
- un jurnal;
- o listă de procese;
- o ieșire din rețea.

Nu continua până nu ai identificat clar sursa.

Pasul 2. Verifică sursa

Execută numai prima comandă.

Exemplu:

```
ps aux
```

sau

```
ls /etc
```

sau

```
journalctl
```

Privește rezultatul.

Există informația pe care o cauți?

Dacă nu, problema este deja identificată.

Pasul 3. Adaugă o singură transformare

Acum introdu primul filtru.

De exemplu:

```
ls /etc | grep '\.conf$'
```

Verifică din nou.

Ai obținut exact ceea ce doreai?

Abia acum mergi mai departe.

Pasul 4. Repetă procesul

Mai adaugi un filtru.

Testezi.

Mai adaugi unul.

Testezi.

Pipeline-ul crește treptat.

Nu apare niciodată complet din prima încercare.

Caseta „În spatele cortinei”

De ce funcționează această metodă?

Pentru că fiecare comandă poate fi verificată independent.

Dacă apare o problemă, știi exact după ce etapă a apărut.

Acest mod de lucru seamănă foarte mult cu depanarea unui program scris într-un limbaj de programare.

În ambele cazuri, izolarea problemei este mai eficientă decât modificarea simultană a mai multor componente.

Greșeală din lumea reală

Un administrator scrie un pipeline de șapte etape și îl execută pentru prima dată.

Rezultatul nu este cel dorit.

Începe să modifice trei sau patru comenzi simultan.

După câteva minute nu mai știe care modificare a rezolvat problema și care a introdus una nouă.

Construirea progresivă evită această situație.

Administrator Notes

Nu încerca să impresionezi prin comenzi foarte lungi.

Un pipeline bun nu este cel mai scurt.

Este cel mai ușor de înțeles și de întreținut.

În administrarea sistemelor, claritatea este aproape întotdeauna mai valoroasă decât ingeniozitatea.

Cum aș depana această problemă?

Dacă un pipeline nu funcționează:

1. Rulez prima comandă.
2. Verific rezultatul.
3. Adaug a doua comandă.
4. Verific din nou.
5. Continui până la ultima etapă.

Nu modific niciodată mai multe componente simultan.

Aceasta este una dintre cele mai eficiente metode de depanare pe care le folosesc administratorii Linux.

Laborator 5.6

Obiectiv

Aplicarea metodei progresive de construire a unui pipeline.

Cerințe

Construiește următorul pipeline în etape:

```
ls /etc
```

↓

```
ls /etc | grep '\.conf$'
```

↓

```
ls /etc | grep '\.conf$' | sort
```

↓

```
ls /etc | grep '\.conf$' | sort | wc -l
```

După fiecare etapă:

- explică ce s-a schimbat;
 - verifică dacă rezultatul este cel așteptat;
 - continuă doar dacă etapa precedentă este corectă.
-

Ce trebuie să reții

- Nu construi pipeline-uri complexe dintr-o singură încercare.
- Verifică fiecare etapă înainte de a adăuga următoarea.
- Depanarea devine mult mai simplă atunci când problema este izolată.
- O metodă bună valorează mai mult decât memorarea unor comenzi lungi.

Capitolul 5

5.7 Fluxul datelor printr-un pipe – Ce se întâmplă între două procese?

Ce vei învăța

În acest capitol vei înțelege cum circulă datele printr-un pipe și de ce două programe pot lucra simultan fără să se cunoască unul pe celălalt.

Introducere

Până acum am privit pipe-ul ca pe un canal care leagă două programe.

Dar apare o întrebare firească.

Cum circulă, de fapt, informația prin acest canal?

Este trimis tot rezultatul odată?

Așteaptă al doilea program până când primul termină?

Sau cele două procese lucrează în același timp?

Pentru a răspunde, trebuie să privim puțin mai atent mecanismul intern.

Primul experiment

Executăm:

```
yes | head -5
```

Rezultatul este:

```
y  
y  
y  
y  
y
```

și comanda se oprește imediat.

Acest exemplu este foarte interesant.

Programul `yes` poate produce un număr practic nelimitat de linii.

Programul head citește doar primele cinci.

Cum este posibil ca întregul lanț să se oprească atât de repede?

Datele nu circulă toate odată

Mulți începători își imaginează următorul scenariu:

Programul A termină.

↓

Trimite toate datele.

↓

Programul B începe.

Nu așa funcționează.

În realitate:

Programul A produce puține date.

↓

Programul B începe deja să le consume.

↓

Programul A produce altele.

↓

Programul B continuă să citească.

Cele două procese colaborează continuu.

Un model mental

Imaginează-ți două persoane care mută cărămizi.

Prima persoană pune câte o cărămidă pe o bandă transportoare.

A doua le ia imediat de pe bandă.

Nu este nevoie ca prima să termine toate cărămizile înainte ca a doua să înceapă.

Pipe-ul funcționează exact la fel.

Caseta „În spatele cortinei”

Bufferul pipe-ului

Între cele două procese există o zonă de memorie administrată de kernel.

Aceasta se numește **buffer**.

Rolul ei este să permită cooperarea dintre procese chiar dacă ele nu lucrează exact cu aceeași viteză.

Dacă programul care produce date este puțin mai rapid, bufferul le păstrează temporar.

Dacă programul care le consumă este mai rapid, acesta va aștepta apariția unor date noi.

Acest mecanism este complet transparent pentru programe.

Ele nu trebuie să știe că bufferul există.

De ce este important acest lucru?

Pentru că explică multe situații întâlnite în practică.

De exemplu, de ce un pipeline începe să producă rezultate imediat, fără să aștepte terminarea primei comenzi.

Sau de ce un proces poate fi încetinit atunci când celălalt nu mai consumă date.

Nu este nevoie să memorăm aceste detalii acum.

Este suficient să înțelegem ideea generală.

Greșeală din lumea reală

Un cursant afirmă:

„Pipe-ul stochează tot rezultatul primei comenzi și apoi îl trimite celei de-a doua.”

Această idee este intuitivă, dar greșită.

În majoritatea situațiilor, datele circulă continuu între cele două procese.

Tocmai acest lucru face pipe-urile atât de eficiente.

Administrator Notes

Atunci când vezi un pipeline care începe să afișeze rezultate imediat, amintește-ți că procesele lucrează în paralel.

Nu aștepta ca prima comandă să termine pentru a presupune că a doua poate începe.

Cum aş depana această problemă?

Problema

Un pipeline pare blocat.

Paşii

1. Verific dacă prima comandă produce date.
2. Verific dacă a doua comandă citeşte din stdin.
3. Mă întreb dacă una dintre etape aşteaptă informaţii care nu mai sosesc.
4. Testez fiecare comandă separat.

În multe situaţii, blocajul apare deoarece unul dintre procese nu mai produce sau nu mai consumă date.

Laborator 5.7

Obiectiv

Observarea modului în care datele circulă printr-un pipe.

Cerinţe

Execută:

```
yes | head -5
```

Apoi:

```
seq 1 10 | cat
```

şi:

```
seq 1 10 | wc -l
```

Pentru fiecare exemplu răspunde:

1. Cine produce datele?
2. Cine le consumă?

3. Datele sunt procesate treptat sau toate odată?

Soluția laboratorului

- `yes` produce continuu date.
 - `head` consumă doar primele cinci linii și apoi se oprește.
 - `seq` produce un șir de numere.
 - `cat` și `wc` procesează informațiile pe măsură ce le primesc.
 - Pipe-ul permite colaborarea continuă dintre procese.
-

Ce trebuie să reții

- Datele circulă continuu prin pipe.
 - Procesele pot rula în paralel.
 - Kernelul coordonează schimbul de date prin intermediul unui buffer.
 - Programele nu trebuie să implementeze ele însele acest mecanism.
-

Întrebare de la un cursant

„Dacă primul program produce milioane de linii, trebuie să existe suficientă memorie pentru toate?”

Răspuns

Nu.

Aceasta este una dintre marile calități ale pipe-urilor.

În mod obișnuit, datele nu sunt păstrate integral în memorie înainte de a fi prelucrate. Ele circulă în flux, pe măsură ce sunt produse și consumate.

Din acest motiv, multe pipeline-uri pot prelucra volume foarte mari de date fără să fie nevoie ca întregul conținut să fie încărcat simultan în memorie.

Capitolul 5

5.8 Modelul fundamental al Unix – Totul este un flux de date

Ce vei învăța

În acest capitol vei reuni toate conceptele învățate până acum într-un singur model mental, care te va ajuta să înțelegi mai ușor atât Bash, cât și multe dintre utilitarele Unix pe care le vei studia în continuare.

Introducere

Până acum am învățat multe mecanisme:

- stdin;
- stdout;
- stderr;
- redirectionări;
- pipe-uri;
- filtre.

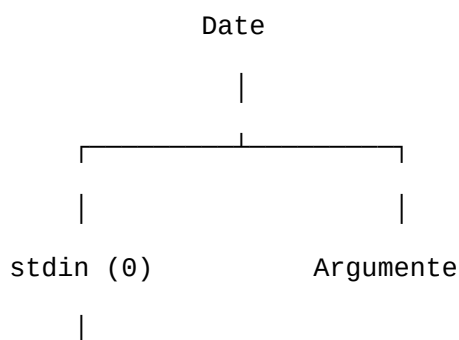
La prima vedere par subiecte diferite.

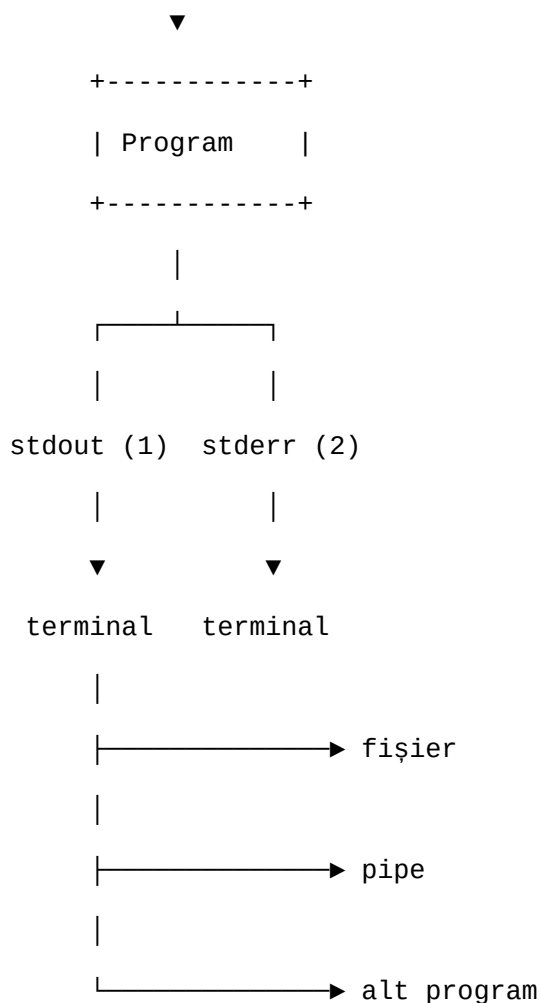
În realitate, toate descriu aceeași idee.

Unix tratează informația ca pe un **flux de date** care poate fi direcționat, filtrat și transformat.

Acesta este unul dintre cele mai importante modele mentale pe care le poate dobândi un administrator Linux.

Un singur desen explică aproape tot





Dacă cititorul înțelege această diagramă, înțelege o parte foarte importantă din arhitectura Bash.

Ce observăm?

Programul nu știe:

- dacă datele provin din tastatură;
- dintr-un fișier;
- dintr-un pipe.

Programul nu știe nici:

- dacă rezultatul ajunge pe ecran;
- într-un fișier;
- într-un alt proces.

El lucrează doar cu fluxurile standard.

Aceasta este cheia reutilizării programelor Unix.

De ce contează?

Să luăm exemplul comenzii:

```
sort
```

Ea poate sorta:

date introduse manual;

```
sort
```

un fișier:

```
sort < nume.txt
```

rezultatul unei alte comenzi:

```
cat nume.txt | sort
```

sau chiar ieșirea unui script.

Programul este același.

Doar conexiunile sunt diferite.

Caseta „În spatele cortinei”

Separarea responsabilităților

În aproape toate exemplele din acest manual apar trei componente.

Programul

Rezolvă problema pentru care a fost creat.

Shell-ul

Pregătește execuția și conectează fluxurile.

Kernelul

Pune la dispoziție mecanismele necesare: procese, descriptori de fișiere, pipe-uri și comunicarea dintre procese.

Această separare este una dintre marile calități ale arhitecturii Unix.

Greșeală din lumea reală

Un administrator încearcă să rezolve orice problemă scriind un script foarte mare.

În multe situații, soluția poate fi obținută prin combinarea unor utilitare deja existente.

Nu pentru că scripturile sunt rele.

Ci pentru că reutilizarea componentelor existente este, de regulă, mai simplă și mai ușor de întreținut.

Administrator Notes

Ori de câte ori întâlnești un utilitar nou, pune-ți trei întrebări:

1. Din ce citește?
2. Ce transformare face?
3. Unde trimite rezultatul?

Dacă poți răspunde la aceste întrebări, vei înțelege rapid rolul acelui utilitar într-un pipeline.

Cum aş depana această problemă?

Dacă un lanț de comenzi nu funcționează, nu începe prin a schimba comenzile.

Începe prin a urmări fluxul datelor.

Întreabă-te:

- intrarea este cea corectă?
- transformarea este cea dorită?
- ieșirea ajunge unde trebuie?

Foarte multe probleme se rezolvă urmărind traseul informației, nu sintaxa comenzilor.

Laborator 5.8

Obiectiv

Aplicarea modelului „fluxului de date”.

Cerințe

Pentru fiecare dintre comenzile următoare identifică:

- sursa datelor;
- programul;
- destinația rezultatului.

```
cat fisier.txt
```

```
sort < fisier.txt
```

```
cat fisier.txt | sort
```

```
grep root /etc/passwd
```

```
grep root /etc/passwd > utilizatori.txt
```

Nu urmări doar rezultatul.

Describe traseul complet al informației.

Soluția laboratorului

În toate exemplele apare aceeași structură:

- există o sursă de date;
- există un program care le procesează;
- există o destinație pentru rezultat.

Se schimbă doar conexiunile dintre aceste componente.

Modelul rămâne același.

Ce trebuie să reții

- Bash nu este doar un interpretor de comenzi, ci și un orchestrator al fluxurilor de date.
- Redirecționările și pipe-urile modifică traseul informației, nu logica programelor.
- Programele Unix sunt proiectate să fie reutilizabile datorită utilizării fluxurilor standard.
- Gândirea în termeni de fluxuri de date simplifică înțelegerea și depanarea comenzilor complexe.

Capitolul 6

6.1 Ce face Bash înainte să execute o comandă?

Ce vei învăța

În acest capitol vei descoperi că Bash nu execută imediat ceea ce scrii. Înainte de lansarea oricărui program, shell-ul analizează, transformă și pregătește comanda. Înțelegerea acestui proces explică multe comportamente care, la început, par surprinzătoare.

Introducere

Să presupunem că scriem:

```
echo $HOME/*.txt
```

Mulți utilizatori cred că:

Bash trimite această comandă către programul `echo`.

Dar nu este adevărat.

Programul `echo` nu primește textul exact pe care l-am tastat.

Între tastatură și execuția programului există o etapă foarte importantă.

Bash analizează și transformă comanda.

Abia apoi o execută.

Acesta este unul dintre cele mai importante lucruri pe care trebuie să le înțelegem despre shell.

Un experiment simplu

Executăm:

```
echo *
```

Dacă în director există:

```
document.txt  
script.sh  
imagini
```

rezultatul va fi:

document.txt script.sh imagini

Întrebarea este:

Cine a înlocuit caracterul * cu lista fișierelor?

Programul echo?

Nu.

Bash.

Iar echo nici măcar nu află că utilizatorul a scris un asterisc.

El primește deja lista completă a fișierelor.

Acesta este un moment de tip „**Aha!**” pentru foarte mulți cursanți.

Încă un exemplu

Executăm:

```
echo $HOME
```

Rezultatul poate fi:

```
/home/daniel
```

Întrebarea este aceeași.

A făcut echo această înlocuire?

Nu.

Și de această dată Bash a înlocuit variabila înainte de lansarea programului.

Modelul mental

De acum înainte trebuie să privim Bash în două etape.

Utilizator

|



Bash analizează

|



Bash transformă

|



Bash execută

|



Programul

Această diagramă va explica aproape toate capitolele următoare.

Caseta „În spatele cortinei”

De ce face Bash toate aceste transformări?

Pentru că shell-ul este mai mult decât un lansator de programe.

El este un **interpretor de limbaj**.

La fel cum un compilator analizează un program înainte de a genera cod executabil, Bash analizează linia de comandă înainte de a porni procesele.

Acesta este motivul pentru care două comenzi care par asemănătoare pot produce rezultate foarte diferite.

Greșeală din lumea reală

Un utilizator spune:

„echo știe să afișeze conținutul lui \$HOME.”

Nu.

echo nu știe nimic despre variabile.

Dacă am scrie un program simplu în C sau Python care doar afișează argumentele primite, am observa același comportament.

Pentru că expansiunea este făcută de Bash, nu de program.

Aceasta este una dintre cele mai importante distincții din întregul shell.

Administrator Notes

Ori de câte ori un rezultat pare „ciudat”, întreabă-te:

A făcut această transformare Bash sau programul pe care l-am executat?

Foarte multe probleme legate de quoting, variabile și wildcard-uri se rezolvă răspunzând corect la această întrebare.

Cum aş depana această problemă?

Dacă o comandă nu produce rezultatul aşteptat:

1. Verific mai întâi ce argumente primeşte programul.
2. Mă întreb dacă Bash a modificat comanda înainte de execuţie.
3. Analizez variabilele, wildcard-urile şi quoting-ul.
4. Abia apoi investighez programul propriu-zis.

De multe ori problema nu este în program, ci în modul în care Bash a interpretat linia de comandă.

Laborator 6.1

Obiectiv

Observarea faptului că Bash transformă comanda înainte de execuţie.

Cerinţe

Într-un director care conţine câteva fişiere, execută:

```
echo *
```

Apoi:

```
echo "$HOME"
```

Şi:

```
echo '$HOME'
```

Pentru fiecare comandă răspunde:

1. Ce ai tastat?
2. Ce argumente crezi că a primit programul `echo`?
3. Ce transformare a realizat Bash înainte de execuţie?

Notă pentru autor: Nu explicăm încă diferența dintre ghilimelele simple și cele duble. Este suficient ca cititorul să observe că rezultatele sunt diferite. Explicația completă va veni în capitolul despre *quoting*. În felul acesta stârnim curiozitatea fără să anticipăm concepte care încă nu au fost predate.

Soluția laboratorului

- În cazul `echo *`, Bash înlocuiește `*` cu lista fișierelor.
 - În cazul `echo "$HOME"`, Bash înlocuiește variabila cu valoarea ei.
 - În cazul `echo '$HOME'`, comportamentul este diferit și va fi explicat în capitolul dedicat ghilimelelor.
 - Programul `echo` afișează argumentele pe care le primește; el nu realizează aceste transformări.
-

Ce trebuie să reții

- Bash interpretează linia de comandă înainte de a executa programele.
- Variabilele și wildcard-urile sunt procesate de Bash.
- Programul executat primește rezultatul acestor transformări, nu textul introdus de utilizator.
- Înțelegerea acestei etape este esențială pentru toate capitolele care urmează.

Unde suntem acum

Dacă privim manualul ca pe o construcție, până acum avem:

Partea I — Fundamentele Bash și ale shell-ului

- Introducerea în Bash
- Pornirea shell-ului
- Comenzi și argumente
- `stdin`, `stdout`, `stderr`
- Redirecționări
- Pipe-uri
- Filtre
- Modele mentale privind fluxul datelor

Urmează Partea a II-a — Cum interpretează Bash o comandă

Aici vom intra în una dintre cele mai importante secțiuni ale întregului manual:

- cum citește Bash linia de comandă;

- tokenizarea;
- quoting;
- escape character (\);
- ghilimele simple și duble;
- variabile;
- parameter expansion;
- command substitution;
- arithmetic expansion;
- pathname expansion (globbing);
- word splitting;
- ordinea expansiunilor;
- alias-uri;
- funcții;
- și apoi, mai departe, structurile limbajului Bash.

Capitolul 6

6.2 Tokenizarea – Cum împarte Bash linia de comandă

„Înainte să înțeleagă o comandă, Bash trebuie mai întâi să știe unde începe și unde se termină fiecare element al ei.”

Ce vei învăța

În acest capitol vei descoperi cum transformă Bash o linie de text într-o structură pe care o poate interpreta. Vom vedea ce este un **token**, de ce spațiile sunt importante și cum modifică ghilimelele și caracterele speciale procesul de analiză.

Introducere

Pentru utilizator, o comandă este doar un șir de caractere.

De exemplu:

```
ls -l /etc
```

Pentru Bash însă, această linie nu este încă o comandă.

Este doar text.

Primul lucru pe care trebuie să îl facă shell-ul este să răspundă la o întrebare aparent banală:

Unde începe fiecare element și unde se termină?

Abia după ce răspunde la această întrebare poate continua cu expansiunile, verificarea aliasurilor, identificarea built-in-urilor și executarea programului.

Acest proces poartă numele de **tokenizare** (*tokenization*).

Ce este un token?

Un **token** este cea mai mică unitate logică pe care Bash o obține în urma analizei unei linii de comandă.

În exemplul:

```
ls -l /etc
```

Bash identifică trei token-uri:

Token	Rol
ls	numele comenzii
-l	opțiune
/etc	argument

Observă că Bash nu știe încă dacă `ls` este un alias, o funcție, un built-in sau un program extern.

Tokenizarea stabilește doar structura textului.

Interpretarea va urma ulterior.

Spațiul are semnificație

În multe limbaje de programare spațiile sunt, în mare parte, ignorate.

În Bash, ele delimitează token-urile.

Compară:

```
echo salut lume
```

cu:

```
echo "salut lume"
```

În primul caz, Bash identifică două argumente:

```
salut
```

lume

În al doilea caz, identifică un singur argument:

salut lume

Aceasta este prima situație în care observăm că **ghilimelele influențează procesul de tokenizare**.

Nu explicăm încă regulile lor complete; este suficient să înțelegem că ele modifică felul în care Bash vede linia de comandă.

Metacaracterele

Nu toate caracterele sunt tratate la fel.

Pentru Bash, anumite caractere au o semnificație specială.

De exemplu:

```
|
&
;
<
>
(
)
$
*
?
[
]
{
}
'
"
\
```

Aceste caractere nu sunt întotdeauna considerate text obișnuit.

În funcție de context, ele pot:

- separa comenzi;
- iniția redirectionări;
- construi pipeline-uri;
- declanșa expansiuni;
- modifica regulile de interpretare.

Vom studia fiecare dintre ele în capitolele următoare.

Un exemplu aparent banal

Compară cele două comenzi:

```
echo test
```

și

```
echo; test
```

În primul caz există o singură comandă.

În al doilea caz, caracterul `;` delimitează două comenzi distincte.

Acest lucru se întâmplă înainte ca Bash să execute orice program.

Prin urmare, `;` nu este un argument pentru `echo`.

Este un separator recunoscut în etapa de analiză sintactică.

Caseta „În spatele cortinei”

De ce nu folosește Bash doar funcția `split()`?

La prima vedere ar părea suficient să împărțim linia după spații.

Dar acest lucru ar produce rezultate greșite.

De exemplu:

```
echo "Linux Essentials"
```

Dacă Bash ar împărți pur și simplu după spații, ar obține trei token-uri:

```
echo  
"Linux  
Essentials"
```

Acest rezultat ar fi incorect.

Parserul Bash trebuie să țină cont de ghilimele, caractere escape, substituții de comandă și multe alte reguli înainte de a decide unde se termină un token.

Din acest motiv, analiza sintactică a unei linii Bash este considerabil mai complexă decât simpla împărțire a unui șir de caractere.

Greșeală din lumea reală

Un utilizator scrie:

```
cd My Documents
```

și primește mesajul:

```
bash: cd: too many arguments
```

Problema nu este comanda `cd`.

Problema este că Bash a identificat două token-uri:

```
My  
Documents
```

Utilizatorul intenționa un singur argument.

Vom vedea în capitolul următor cum quoting-ul rezolvă această situație.

Administrator Notes

Atunci când o comandă produce un rezultat neașteptat, întreabă-te mai întâi:

A interpretat Bash linia în același mod în care am intenționat eu?

Foarte multe erori apar înainte ca programul să înceapă efectiv execuția.

Legătura cu LPIC

Examenul LPIC rareori întreabă direct ce este un *token*.

În schimb, multe întrebări presupun că înțelegi modul în care Bash împarte și interpretează linia de comandă. Această înțelegere va deveni esențială în capitolele despre quoting, wildcard-uri și expansiuni.

Laborator 6.2

Obiectiv

Observarea modului în care Bash delimitează token-urile.

Cerințe

Execută următoarele comenzi:

```
echo Linux Essentials
```

```
echo "Linux Essentials"
```

```
mkdir Test Director
```

```
mkdir "Test Director"
```

Pentru fiecare comandă răspunde:

1. Câte token-uri identifică Bash?
2. Care dintre ele reprezintă comanda?
3. Care sunt argumentele?
4. Rezultatul corespunde intenției tale?

Soluția laboratorului

- `echo Linux Essentials` → trei token-uri (`echo`, `Linux`, `Essentials`).
- `echo "Linux Essentials"` → două token-uri (`echo`, `Linux Essentials`).
- `mkdir Test Director` → Bash încearcă să creeze două directoare: `Test` și `Director`.
- `mkdir "Test Director"` → Bash transmite un singur argument către `mkdir`, iar acesta creează un director cu spațiu în nume.

Acest laborator arată că **problema nu este în programe**, ci în modul în care Bash interpretează linia de comandă înainte de a le executa.

Ce trebuie să reții

- Bash începe prin a transforma textul introdus în token-uri.
- Spațiile delimitează, în mod obișnuit, token-urile.
- Ghilimelele pot modifica această regulă și vor fi studiate în detaliu în capitolul următor.
- Metacaracterele au semnificații speciale și influențează analiza sintactică.
- O interpretare corectă a token-urilor este baza tuturor etapelor ulterioare ale procesării unei comenzi.

Capitolul 6

6.3 Quoting – Cum îi spunem lui Bash să interpreteze textul exact așa cum dorim

„Nu toate caracterele pe care le tastăm trebuie interpretate. Uneori dorim ca Bash să le trateze ca text obișnuit. Quoting-ul este mecanismul prin care controlăm acest comportament.”

Ce vei învăța

În acest capitol vei înțelege de ce există quoting-ul și cum influențează modul în care Bash interpretează o comandă. Vom studia conceptul general, iar în capitolele următoare vom analiza separat fiecare tip de quoting.

Introducere

În capitolul anterior am văzut că Bash nu execută imediat ceea ce tastăm.

Mai întâi analizează linia de comandă.

Identifică token-urile.

Recunoaște metacaracterele.

Pregătește expansiunile.

Acest lucru ridică o întrebare foarte importantă.

Ce facem atunci când dorim ca un caracter special să fie tratat ca text obișnuit?

De exemplu:

*

Trebuie Bash să îl interpreteze ca wildcard?

Sau vrem pur și simplu caracterul *?

Același lucru este valabil pentru:

\$

Dorim expansiunea unei variabile?

Sau chiar caracterul \$?

Pentru astfel de situații există mecanismul numit **quoting**.

Ideea fundamentală

Quoting-ul nu modifică programul executat.

Nu modifică rezultatul unei comenzi.

El modifică **modul în care Bash interpretează linia de comandă înainte de execuție**.

Aceasta este ideea pe care trebuie să o reținem.

Un exemplu simplu

Executăm:

```
echo $HOME
```

Rezultatul poate fi:

```
/home/daniel
```

Acum executăm:

```
echo '$HOME'
```

Rezultatul este:

```
$HOME
```

Observă că programul executat este același.

echo.

Diferența nu este în program.

Diferența este în modul în care Bash a interpretat argumentul.

Încă un exemplu

Executăm:

```
echo *
```

Rezultatul poate fi:

```
document.txt script.sh manual.pdf
```

Acum:

```
echo "*"
```

Rezultatul este:

```
*
```

Din nou, `echo` nu s-a schimbat.

S-a schimbat doar interpretarea realizată de Bash.

Ce protejează quoting-ul?

În practică, quoting-ul poate împiedica interpretarea unor elemente precum:

- spațiile;
- wildcard-urile (`*`, `?`, `[]`);
- variabilele (`$HOME`);
- substituțiile de comandă (`$(...)`);
- caracterele speciale ale shell-ului.

Vom analiza fiecare dintre aceste situații în capitolele următoare.

Momentan este suficient să înțelegem scopul general.

Trei mecanisme de quoting

Bash oferă trei modalități principale prin care putem controla interpretarea caracterelor:

1. Caracterul escape

```
\
```

2. Ghilimelele simple

```
'...'
```

3. Ghilimelele duble

```
"..."
```

Fiecare dintre ele are reguli proprii.

Nu sunt echivalente.

Acesta este motivul pentru care le vom studia separat.

Caseta „În spatele cortinei”

De ce există trei mecanisme diferite?

La prima vedere pare că unul singur ar fi fost suficient.

Dar utilizatorii au nevoi diferite.

Uneori doresc:

„Nu interpreta absolut nimic.”

Alteori:

„Interpretează variabilele, dar nu wildcard-urile.”

Sau:

„Protejează doar caracterul următor.”

Un singur mecanism nu ar putea acoperi toate aceste situații într-un mod simplu.

De aceea Bash oferă mai multe niveluri de control.

Greșeală din lumea reală

Un utilizator execută:

```
rm *
```

crezând că șterge doar fișierul numit `*`.

În realitate, Bash înlocuiește `*` cu lista tuturor fișierelor din director.

Problema nu este comanda `rm`.

Problema este că utilizatorul nu a înțeles cum interpretează Bash caracterul `*`.

Acest exemplu arată de ce quoting-ul nu este doar un subiect teoretic, ci unul cu implicații directe asupra administrării unui sistem.

Administrator Notes

Ori de câte ori o comandă produce un rezultat neașteptat, întreabă-te:

A interpretat Bash unul dintre caracterele mele într-un mod pe care nu l-am anticipat?

Foarte multe probleme aparent misterioase se explică prin lipsa quoting-ului sau prin alegerea unui tip nepotrivit de quoting.

Legătura cu LPIC

Subiectul quoting-ului apare frecvent în întrebările LPIC.

De regulă, examenul nu cere definiții.

În schimb, prezintă două comenzi foarte asemănătoare și întreabă de ce rezultatele sunt diferite.

Dacă înțelegi mecanismul prezentat în acest capitol, vei putea explica aceste diferențe fără să memorezi exemple.

Laborator 6.3

Obiectiv

Observarea influenței quoting-ului asupra interpretării comenzilor.

Cerințe

Execută următoarele comenzi:

```
echo $HOME
```

```
echo "$HOME"
```

```
echo '$HOME'
```

Apoi:

```
echo *
```

```
echo "*"
```

```
echo '*'
```

Pentru fiecare comandă răspunde:

1. Ce ai tastat?
2. Ce a interpretat Bash?
3. Ce argument a primit programul `echo`?
4. Care este diferența față de comanda anterioară?

Nu încerca încă să explici regulile fiecărui tip de quoting.

Observă doar că rezultatele diferă.

Explicațiile complete vor urma imediat.

Soluția laboratorului

- Bash interpretează diferit aceeași secvență de caractere în funcție de quoting.
 - Programul `echo` primește argumente diferite, chiar dacă utilizatorul a scris aproape aceeași comandă.
 - Quoting-ul controlează interpretarea realizată de Bash înainte de execuția programului.
-

Ce trebuie să reții

- Quoting-ul controlează modul în care Bash interpretează caracterele speciale.
- El acționează înainte de executarea programului.
- Bash oferă trei mecanisme principale de quoting.
- În capitolele următoare vom studia separat fiecare mecanism și regulile lui.

Capitolul 6

6.4 Caracterul Escape (\) – Cum protejăm un singur caracter

„Nu întotdeauna dorim să dezactivăm interpretarea unei întregi secvențe de caractere. Uneori vrem să protejăm doar unul singur. Pentru aceasta există caracterul escape.”

Ce vei învăța

În acest capitol vei învăța rolul caracterului `\`, când este util și de ce influențează doar caracterul care îl urmează.

Introducere

În capitolul anterior am aflat că Bash interpretează anumite caractere într-un mod special.

De exemplu:

- \$
- *
- ?
- "
- '
- spațiul
- \

Uneori însă nu dorim această interpretare.

Dorim ca Bash să trateze un caracter exact așa cum este scris.

Pentru astfel de situații există **caracterul escape**.

Acesta este reprezentat prin:

\

Rolul lui este foarte simplu:

Îi spune lui Bash să trateze caracterul următor ca text obișnuit, chiar dacă în mod normal ar avea o semnificație specială.

Primul exemplu

Executăm:

```
echo \ $HOME
```

Rezultatul este:

```
$HOME
```

Observă ce s-a întâmplat.

Fără backslash:

```
echo $HOME
```

Bash interpretează variabila și afișează valoarea ei.

Cu backslash:

```
echo \ $HOME
```

Bash nu mai interpretează caracterul \$.

Programul echo primește textul literal:

```
$HOME
```

Încă un exemplu

Executăm:

```
echo \*
```

Rezultatul este:

```
*
```

Fără backslash:

```
echo *
```

Bash înlocuiește caracterul `*` cu lista fișierelor din director.

Cu backslash:

```
echo \*
```

Caracterul este protejat.

Wildcard-ul nu mai este interpretat.

Spațiile pot fi și ele protejate

Presupunem că dorim să creăm un director numit:

```
Linux Essentials
```

Putem scrie:

```
mkdir Linux\ Essentials
```

Bash nu mai tratează spațiul ca separator de token-uri.

Programul `mkdir` primește un singur argument:

```
Linux Essentials
```

Acesta este unul dintre cele mai întâlnite exemple din administrarea Linux.

Un model mental

Imaginează-ți că Bash citește caracterele unul câte unul.

În momentul în care întâlnește caracterul:

```
\
```

el spune:

„Nu interpreta următorul caracter. Trimite-l exact așa mai departe.”

După aceea își continuă analiza normal.

Aceasta explică de ce backslash-ul influențează doar caracterul imediat următor.

Caseta „În spatele cortinei”

De ce doar caracterul următor?

Backslash-ul nu deschide și nu închide un context, așa cum fac ghilimelele.

El este un modificador local.

Acționează asupra unui singur caracter.

Acest lucru îl face foarte util atunci când trebuie protejat un singur simbol special fără a afecta restul textului.

Greșeală din lumea reală

Un utilizator dorește să afișeze textul:

```
$USER
```

și execută:

```
echo $USER
```

Rezultatul nu este textul dorit, ci numele utilizatorului conectat.

Problema nu este comanda `echo`.

Problema este că Bash interpretează caracterul `$`.

Una dintre soluții este:

```
echo \ $USER
```

sau, după cum vom vedea în capitolele următoare:

```
echo '$USER'
```

Ambele variante produc textul literal, însă prin mecanisme diferite.

Administrator Notes

Caracterul \ este foarte util atunci când trebuie protejat un singur caracter.

Dacă observi că începi să îl folosești de foarte multe ori în aceeași comandă, întreabă-te dacă nu ar fi mai clar să folosești ghilimele.

În general:

- pentru unul sau două caractere speciale, \ este o alegere bună;
 - pentru un text mai lung, quoting-ul cu ghilimele este, de regulă, mai ușor de citit.
-

Legătura cu LPIC

Examenul LPIC include frecvent întrebări în care un singur caracter \ schimbă complet comportamentul unei comenzi.

Nu este important să memorezi exemple.

Este important să înțelegi regula:

Backslash-ul protejează caracterul imediat următor.

Cum aş depana această problemă?

Problema

Comanda interpretează un caracter pe care îl doream afișat ca text.

Pași

1. Identific caracterul special (\$, *, ?, spațiu etc.).
 2. Verific dacă Bash îl interpretează.
 3. Testez protejarea lui cu \.
 4. Dacă trebuie protejat un text mai lung, analizez dacă ghilimelele reprezintă o soluție mai potrivită.
-

Laborator 6.4

Obiectiv

Înțelegerea efectului caracterului \.

Cerințe

Execută:

```
echo $HOME
```

```
echo \ $HOME
```

```
echo *
```

```
echo \*
```

Creează apoi două directoare:

```
mkdir Test\ Directory
```

și

```
mkdir "Test Directory 2"
```

Răspunde:

1. Ce caracter a fost protejat în fiecare exemplu?
 2. Ce argument a primit programul executat?
 3. În ce situații ai prefera \ și în ce situații ai prefera ghilimelele?
-

Soluția laboratorului

- \ \$HOME împiedică expansiunea variabilei.
 - \ * împiedică interpretarea wildcard-ului.
 - Test\ Directory transformă spațiul într-un caracter obișnuit.
 - Ghilimelele și caracterul \ pot rezolva uneori aceeași problemă, însă în moduri diferite.
-

Ce trebuie să reții

- Caracterul \ influențează numai caracterul imediat următor.
- El împiedică Bash să interpreteze acel caracter într-un mod special.
- Este foarte util pentru protejarea unui singur caracter.
- Pentru texte mai lungi, ghilimelele oferă de regulă o soluție mai clară.

Capitolul 6

6.5 Ghilimelele simple (') – Când dorești ca Bash să nu interpreteze nimic

„Ghilimelele simple îi spun lui Bash un lucru foarte clar: tratează textul exact așa cum este scris. Nu interpreta variabile, wildcard-uri sau alte caractere speciale.”

Ce vei învăța

În acest capitol vei înțelege rolul ghilimelelor simple și vei vedea de ce reprezintă cea mai strictă formă de quoting oferită de Bash.

Introducere

În capitolul anterior am învățat că backslash-ul protejează un singur caracter.

Aceasta este o soluție foarte bună atunci când avem nevoie să tratăm literal doar unul sau două caractere speciale.

Dar apare rapid o situație practică.

Dorim să afișăm textul:

```
$HOME/*.txt
```

Am putea scrie:

```
echo \"$HOME/*.txt
```

Comanda funcționează.

Dar este greu de citit.

Există o soluție mai simplă.

Ghilimelele simple

Executăm:

```
echo '$HOME/*.txt'
```

Rezultatul este:

```
$HOME/* .txt
```

Observă ce s-a întâmplat.

Bash nu a interpretat:

- \$
- *
- /

Textul a fost transmis exact așa cum a fost scris.

Acesta este rolul ghilimelelor simple.

Regula fundamentală

Putem formula regula astfel:

Tot ceea ce se află între ghilimele simple este tratat ca text literal.

Nu există expansiune de variabile.

Nu există globbing.

Nu există command substitution.

Nu există interpretarea caracterelor speciale.

Această regulă este foarte ușor de reținut deoarece are foarte puține excepții.

Exemple

Variabilă

```
echo '$USER'
```

Rezultatul:

```
$USER
```

Wildcard

```
echo '*'
```


Rezultatul:

*

Command substitution

```
echo 'Today is $(date)'
```

Rezultatul:

```
Today is $(date)
```

Comanda `date` nu este executată.

Textul este afișat exact așa cum a fost introdus.

O comparație

Compară:

```
echo \ $USER
```

și

```
echo ' $USER '
```

Ambele produc același rezultat.

Diferența este că:

- backslash-ul protejează doar caracterul \$;
- ghilimelele simple protejează întreaga secvență.

Din acest motiv, pentru texte mai lungi, ghilimelele simple sunt de regulă mai ușor de citit și de întreținut.

Limitarea ghilimelelor simple

Există însă și un dezavantaj.

Fiind cea mai strictă formă de quoting, ele nu permit niciun fel de expansiune.

Dacă dorim ca Bash să înlocuiască valoarea unei variabile, ghilimelele simple nu sunt potrivite.

Pentru această situație vom studia, în capitolul următor, ghilimelele duble.

Caseta „În spatele cortinei”

De ce există ghilimelele simple?

Pentru că există situații în care utilizatorul dorește control total asupra textului transmis către program.

De exemplu:

- expresii regulate;
- comenzi SQL;
- fragmente de cod;
- exemple din documentație;
- texte care conțin multe caractere speciale.

În toate aceste cazuri este mai sigur ca Bash să nu interpreteze nimic.

Greșeală din lumea reală

Un administrator dorește să afișeze un exemplu într-o documentație:

```
$HOME
```

Scrie:

```
echo "$HOME"
```

În locul textului dorit apare calea către directorul personal.

Problema nu este `echo`.

Problema este alegerea tipului de quoting.

În acest caz soluția corectă este:

```
echo '$HOME'
```

Administrator Notes

Ori de câte ori scopul tău este să afișezi sau să transmiți **text literal**, întreabă-te dacă ghilimelele simple nu reprezintă cea mai clară alegere.

Ele reduc foarte mult riscul ca Bash să interpreteze accidental un caracter special.

Legătura cu LPIC

În examenul LPIC apar frecvent întrebări în care două comenzi diferă doar prin tipul ghilimelelor utilizate.

Nu încerca să memorezi toate exemplele.

Memorează regula:

Între ghilimele simple, Bash nu interpretează conținutul.

Din această regulă derivă toate exemplele.

Cum aş depana această problemă?

Problema

O variabilă nu este expandată.

Paşii

1. Verific dacă textul este între ghilimele simple.
 2. Dacă da, îmi amintesc că Bash tratează întregul conţinut ca text literal.
 3. Analizez dacă intenţia mea era să afişez textul sau valoarea variabilei.
 4. Dacă doream expansiune, aleg un alt mecanism de quoting.
-

Laborator 6.5

Obiectiv

Observarea efectului ghilimelelor simple.

Cerinţe

Execută:

```
echo '$HOME'
```

```
echo '$USER'
```

```
echo '*.txt'
```

```
echo 'Today is $(date)'
```

Răspunde:

1. Ce a tastat utilizatorul?

2. Ce a interpretat Bash?
 3. Ce argument a primit programul `echo`?
 4. Există vreo expansiune?
-

Soluția laboratorului

În toate exemplele:

- Bash transmite textul exact așa cum apare între ghilimele.
- Nu are loc expansiunea variabilelor.
- Nu sunt interpretate wildcard-urile.
- Nu este executată substituția de comandă.

Programul `echo` primește textul literal.

Ce trebuie să reții

- Ghilimelele simple reprezintă cea mai strictă formă de quoting din Bash.
- Tot textul dintre ele este transmis literal.
- Nu au loc expansiuni sau interpretări ale caracterelor speciale.
- Sunt recomandate atunci când dorești să protejezi un text întreg, nu doar un caracter.

Capitolul 6

6.6 Ghilimelele duble (") – Protecție și expansiune în același timp

„Există situații în care dorim să protejăm textul împotriva unor interpretări ale shell-ului, dar vrem în același timp ca variabilele și alte expansiuni să funcționeze. Pentru aceste situații Bash oferă ghilimelele duble.”

Ce vei învăța

În acest capitol vei înțelege de ce ghilimelele duble sunt diferite de ghilimelele simple și în ce situații reprezintă alegerea corectă.

Introducere

În capitolul anterior am văzut că ghilimelele simple tratează întregul conținut ca text literal.

Acest comportament este foarte util.

Dar apare rapid o situație practică.

Presupunem că avem variabila:

```
NUME="Daniel"
```

Dorim să afișăm:

```
Salut, Daniel!
```

Dacă scriem:

```
echo 'Salut, $NUME!'
```

rezultatul este:

```
Salut, $NUME!
```

Nu este ceea ce doream.

Am vrea ca textul să rămână protejat, dar variabila să fie totuși expandată.

Acesta este exact rolul ghilimelelor duble.

Primul exemplu

Executăm:

```
NUME="Daniel"  
echo "$NUME"
```

Rezultatul este:

```
Daniel
```

Compară cu:

```
echo '$NUME'
```

care produce:

```
$NUME
```

Diferența nu este în `echo`.

Diferența este în modul în care Bash interpretează conținutul dintre ghilimele.

Ce permit ghilimelele duble?

În interiorul ghilimelelor duble, Bash continuă să realizeze anumite expansiuni importante.

De exemplu:

- **Parameter Expansion**

```
echo "$HOME"
```

- **Command Substitution**

```
echo "Astăzi este $(date)"
```

- **Arithmetic Expansion**

```
echo "2 + 3 = $((2+3))"
```

Aceste mecanisme funcționează în continuare.

Ce nu permit?

În schimb, ghilimelele duble împiedică interpretarea unor mecanisme precum:

- **Pathname Expansion (globbing)**

```
echo "*.txt"
```

Rezultatul este:

```
*.txt
```

și nu lista fișierelor.

Acest comportament este extrem de util atunci când dorim să transmitem un model către un program fără ca Bash să îl extindă mai întâi.

Spațiile sunt păstrate

Un alt efect important este că textul dintre ghilimele duble este tratat ca un singur argument.

Presupunem:

```
NUME="Linux Essentials"
```

Executăm:

```
echo "$NUME"
```

Programul `echo` primește un singur argument:

`Linux Essentials`

Dacă variabila nu ar fi fost între ghilimele, Bash ar putea împărți rezultatul în mai multe argumente în etapa de **Word Splitting**, pe care o vom studia mai târziu.

Notă pentru autor: nu dezvoltăm acum mecanismul de *Word Splitting*; doar pregătim terenul și facem trimitere către capitolul dedicat.

Un model mental

Ghilimelele duble îi spun lui Bash:

„Protejează structura textului, dar continuă să interpretezi expansiunile care sunt sigure și utile.”

Aceasta este diferența esențială față de ghilimelele simple.

Caseta „În spatele cortinei”

De ce există două tipuri de ghilimele?

Pentru că utilizatorii au două nevoi diferite.

Uneori doresc:

„Nu interpreta absolut nimic.”

Atunci folosesc ghilimelele simple.

Alteori doresc:

„Interpretează variabilele și substituțiile de comandă, dar păstrează textul ca un singur argument.”

Atunci folosesc ghilimelele duble.

Nu există un tip „mai bun”.

Există tipul potrivit pentru situația respectivă.

Greșeală din lumea reală

Un administrator scrie:

```
cp $SURSA $DESTINATIE
```

Funcționează cât timp directoarele nu conțin spații.

Mai târziu definește:

```
SURSA="/home/daniel/Documente importante"
```

și comanda începe să dea erori.

Forma recomandată este:

```
cp "$SURSA" "$DESTINATIE"
```

Nu pentru că variabilele nu ar fi fost expandate și înainte, ci pentru că ghilimelele păstrează rezultatul expansiunii ca un singur argument.

Aceasta este una dintre cele mai importante bune practici în scripting Bash.

Administrator Notes

În scripturile Bash există o regulă simplă care previne foarte multe erori:

Când expandezi o variabilă care reprezintă un nume de fișier sau o cale, pune-o între ghilimele duble.

Există excepții, pe care le vom studia mai târziu, dar pentru început această regulă este una dintre cele mai utile pe care le poate învăța un administrator.

Legătura cu LPIC

În exercițiile LPIC, multe întrebări despre scripting implică utilizarea corectă a ghilimelelor duble în jurul variabilelor.

Nu este doar o convenție de stil.

Este un mecanism care influențează direct modul în care Bash construiește lista argumentelor transmise programului.

Cum aş depana această problemă?

Problema

O comandă funcționează pentru unele directoare, dar eșuează pentru altele.

Pașii

1. Verific dacă variabilele conțin spații sau caractere speciale.
2. Verific dacă expansiunile sunt încadrate între ghilimele duble.
3. Analizez argumentele pe care le primește programul după interpretarea realizată de Bash.
4. Testez din nou comanda după adăugarea ghilimelelor.

Foarte multe erori aparent misterioase dispar în acest moment.

Laborator 6.6

Obiectiv

Înțelegerea comportamentului ghilimelelor duble.

Cerințe

Execută:

```
NUME="Linux Essentials"
```

Apoi:

```
echo $NUME
```

```
echo "$NUME"
```

Execută și:

```
echo "*.txt"
```

```
echo "Data: $(date)"
```

```
echo "Rezultat: $((10+5))"
```

Pentru fiecare comandă explică:

1. Ce expansiuni au avut loc?
2. Ce mecanisme au fost împiedicate?
3. Ce argumente a primit programul echo?

Soluția laboratorului

- Variabilele sunt expandate în interiorul ghilimelelor duble.
 - Substituția de comandă și expansiunea aritmetică funcționează în continuare.
 - Globbing-ul este dezactivat.
 - Rezultatul expansiunii este păstrat ca un singur argument.
-

Ce trebuie să reții

- Ghilimelele duble reprezintă un compromis între protecție și flexibilitate.
- Variabilele și alte expansiuni importante continuă să funcționeze.
- Wildcard-urile nu sunt expandate.
- În scripting, utilizarea ghilimelelor duble în jurul variabilelor care reprezintă căi sau nume de fișiere este o bună practică.

Capitolul 6

6.7 Cum aleg între \, ' și "?

„Bash oferă mai multe mecanisme de quoting deoarece utilizatorii au nevoi diferite. Alegerea corectă depinde de ceea ce dorim să protejăm și de ceea ce dorim să permitem shell-ului să interpreteze.”

Ce vei învăța

În acest capitol vei compara cele trei mecanisme de quoting și vei învăța cum să alegi soluția potrivită în funcție de situație.

Introducere

În capitolele anterioare am studiat:

- caracterul escape (\);
- ghilimelele simple (');
- ghilimelele duble (").

Fiecare dintre ele rezolvă aceeași problemă generală:

Controlul modului în care Bash interpretează caracterele speciale.

Dar nu o fac în același mod.

Acum este momentul să le privim împreună.

Caracterul Escape (\)

Se utilizează atunci când dorim să protejăm **un singur caracter**.

Exemplu:

```
echo \$HOME
```

Avantaje:

- simplu;
- precis;
- ideal pentru unul sau două caractere speciale.

Limitări:

- devine greu de citit atunci când trebuie protejate multe caractere.
-

Ghilimelele simple (')

Se utilizează atunci când dorim să transmitem **text literal**.

Exemplu:

```
echo '$HOME/* .txt'
```

Avantaje:

- foarte ușor de înțeles;
- Bash nu interpretează conținutul.

Limitări:

- nu permit expansiunea variabilelor;
 - nu permit substituția comenzilor;
 - nu permit expansiunea aritmetică.
-

Ghilimelele duble (")

Se utilizează atunci când dorim:

- să păstrăm textul ca un singur argument;
- dar să permitem expansiunile utile.

Exemplu:

```
echo "$HOME"
```

Avantaje:

- variabilele sunt expandate;
- command substitution funcționează;
- arithmetic expansion funcționează;
- spațiile sunt păstrate.

Limitări:

- nu reprezintă text complet literal.

Tabel comparativ

Caracteristică	\	'	"
Protejează un singur caracter	✓	✗	✗
Protejează un text întreg	✗	✓	✓
Permite Parameter Expansion	✓ (prin alegerea caracterelor neprotejate)	✗	✓
Permite Command Substitution	✓ (dacă nu este protejat \$)	✗	✓
Permite Arithmetic Expansion	✓ (dacă nu este protejat \$)	✗	✓
Păstrează spațiile într-un singur argument	doar dacă fiecare spațiu este protejat	✓	✓

Notă pentru autor: În cazul caracterului \, expansiunile nu sunt „activate” sau „dezactivate” global. Totul depinde de caracterul pe care îl protejăm. De aceea formularea din tabel trebuie explicată în text și nu lăsată să fie interpretată absolut.

Cum aleg în practică?

Poți folosi următorul algoritm.

Vrei să protejezi un singur caracter?

Folosește:

\

Vrei text complet literal?

Folosește:

'...'

Vrei variabile expandate, dar și protecție împotriva împărțirii în argumente?

Folosește:

"..."

Studii de caz

Cazul 1

Vrei să afișezi:

\$HOME

Soluția:

echo '\$HOME'

Cazul 2

Vrei să afișezi valoarea variabilei.

Soluția:

echo "\$HOME"

Cazul 3

Vrei doar caracterul \$.

Soluția:

echo \ \$HOME

Cazul 4

Ai o variabilă care conține:

/home/daniel/My Documents

Soluția recomandată:

cp "\$SURSA" "\$DESTINATIE"

Caseta „În spatele cortinei”

De ce nu există un singur tip de quoting?

Pentru că Bash încearcă să ofere utilizatorului control cât mai precis.

Uneori este suficient să protejezi un singur caracter.

Alteori trebuie protejat un text întreg.

Alteori dorești un echilibru între protecție și expansiune.

Cele trei mecanisme răspund acestor trei nevoi diferite.

Greșeală din lumea reală

Un administrator folosește ghilimele simple în jurul unei variabile:

```
echo '$HOME'
```

și se întreabă de ce nu apare directorul personal.

Problema nu este Bash.

Problema este alegerea mecanismului de quoting.

Administrator Notes

Nu încerca să memorezi toate exemplele.

Încearcă să răspunzi la întrebarea:

Ce vreau să interpreteze Bash și ce vreau să lase nemodificat?

Dacă răspunsul este clar, alegerea mecanismului de quoting devine aproape evidentă.

Legătura cu LPIC

În multe întrebări LPIC, diferența dintre răspunsul corect și unul greșit este reprezentată doar de tipul de quoting utilizat.

Dacă înțelegi principiile prezentate în acest capitol, nu vei mai avea nevoie să memorezi cazuri particulare.

Laborator 6.7

Obiectiv

Alegerea mecanismului potrivit de quoting.

Cerințe

Rezolvă următoarele situații:

1. Afișează textul:

```
$USER
```

2. Afișează numele utilizatorului conectat.

3. Creează directorul:

```
Linux Essentials
```

folosind:

- backslash;
- ghilimele duble.

4. Explică de ce următoarele două comenzi produc rezultate diferite:

```
echo '$HOME'
```

```
echo "$HOME"
```

Soluția laboratorului

- 1.

```
echo '$USER'
```

- 2.

```
echo "$USER"
```

- 3.

```
mkdir Linux\ Essentials
```

sau

```
mkdir "Linux Essentials"
```

- 4.

În primul caz Bash transmite textul literal.

În al doilea caz Bash realizează expansiunea variabilei înainte de lansarea programului `echo`.

Ce trebuie să reții

- `\` protejează un singur caracter.
- `' . . . '` transmite text literal.
- `" . . . "` permite expansiunile utile și păstrează textul ca un singur argument.
- Alegerea mecanismului potrivit depinde de intenția utilizatorului, nu de preferințe personale.

Capitolul 6

6.8 Parameter Expansion – Cum înlocuiește Bash variabilele înainte de execuția unei comenzi

„Atunci când scriem `$HOME` sau `${USER}`, programul executat nu vede niciodată aceste expresii. Bash le înlocuiește cu valorile lor înainte de lansarea procesului.”

Ce vei învăța

În acest capitol vei înțelege ce este **Parameter Expansion**, când are loc și de ce reprezintă unul dintre cele mai importante mecanisme ale limbajului Bash.

Introducere

Până acum am studiat quoting-ul și am văzut că Bash interpretează linia de comandă înainte de a executa programele.

Acum analizăm unul dintre mecanismele cele mai utilizate în fiecare zi:

Parameter Expansion.

De fiecare dată când scriem:

```
echo $HOME
```

sau

```
echo ${USER}
```


Bash efectuează o transformare înainte de execuția comenzii.

Programul `echo` nu primește textul:

```
$HOME
```

ci valoarea variabilei.

Acest mecanism poartă numele de **Parameter Expansion**.

Ce este un parameter?

În terminologia Bash, un **parameter** este un nume asociat unei valori.

Poate fi:

- o variabilă definită de utilizator;

```
NUME="Daniel"
```

- o variabilă de mediu;

```
HOME
```

```
PATH
```

```
USER
```

```
LANG
```

- un parametru special;

```
$?
```

```
$$
```

```
$#
```

```
$*
```

```
$@
```

```
$0
```

Vom studia parametrii speciali într-un capitol separat.

În acest capitol ne concentrăm asupra mecanismului de expansiune.

Primul exemplu

Presupunem:

```
NUME="Daniel"
```

Executăm:

```
echo $NUME
```

Rezultatul este:

Daniel

Întrebarea este:

Cine a înlocuit \$NUME cu Daniel?

Nu echo.

Bash.

Cum putem vizualiza procesul?

Utilizator

|

echo \$NUME

|

▼

Parameter Expansion

|

▼

echo Daniel

|

▼

execve()

|

▼

Programul echo

Această diagramă explică aproape toate exemplele din capitol.

De ce există forma \${NUME}?

Mulți începători întreabă:

„Dacă \$NUME funcționează, de ce mai există \${NUME}?”

Este o întrebare foarte bună.

În exemple simple:

```
echo $HOME
```

și

```
echo ${HOME}
```

produc același rezultat.

Dar există situații în care Bash nu poate determina unde se termină numele variabilei.

De exemplu:

```
NUME="Daniel"  
echo "${NUME}123"
```

Rezultatul este:

```
Daniel123
```

Dacă am scrie:

```
echo $NUME123
```

Bash ar căuta variabila:

```
NUME123
```

care este diferită de NUME.

Prin urmare, acoladele nu sunt doar o chestiune de stil.

Ele delimitează explicit numele parametrului.

Caseta „În spatele cortinei”

Expansiunea produce text

Parameter Expansion nu transmite obiecte și nici referințe.

Rezultatul este întotdeauna un șir de caractere.

Acest șir poate participa apoi la celelalte etape ale procesării Bash:

- Word Splitting;
- Pathname Expansion;
- Quote Removal.

Vom vedea aceste etape în capitolele următoare.

Greșeală din lumea reală

Un utilizator scrie:

```
FISIER=raport  
echo $FISIER.txt
```

și se întreabă de ce rezultatul este gol.

Bash încearcă să găsească variabila:

```
FISIER.txt
```

sau, mai exact, interpretează numele variabilei până unde regulile îi permit, ceea ce nu exprimă intenția utilizatorului.

Forma recomandată este:

```
echo "${FISIER}.txt"
```

Acoladele elimină orice ambiguitate.

Administrator Notes

O recomandare simplă, dar foarte utilă:

În scripturi, folosește forma `${VAR}` ori de câte ori variabila este lipită de alt text.

Codul devine mai clar și evită interpretările greșite.

De exemplu:

```
LOG="${HOSTNAME}.log"  
BACKUP="${DIRECTOR}_backup"
```

Acest stil este întâlnit frecvent în proiectele Bash bine scrise.

Legătura cu LPIC

Forma `${VAR}` apare frecvent în exemplele și exercițiile LPIC.

Nu pentru că este obligatorie în toate situațiile, ci pentru că exprimă fără ambiguitate limitele numelui variabilei și permite utilizarea formelor avansate de Parameter Expansion pe care le vom studia imediat.

Cum aş depăna această problemă?

Problema

O variabilă pare să nu fie expandată corect.

Paşii

1. Verific dacă variabila există:

```
echo "$NUME"
```

2. Dacă este urmată imediat de alte caractere, folosesc forma:

```
"${NUME}"
```

3. Verific dacă quoting-ul influenţează rezultatul.
 4. Abia apoi investighez logica scriptului.
-

Laborator 6.8

Obiectiv

Înţelegerea mecanismului de Parameter Expansion.

Cerinţe

Execută:

```
NUME="Daniel"
```

Apoi:

```
echo $NUME
```

```
echo ${NUME}
```

```
echo "${NUME}123"
```

```
echo $NUME123
```

Răspunde:

1. Cine realizează expansiunea?
 2. Ce argument primeşte programul echo?
 3. De ce ultimele două comenzi produc rezultate diferite?
-

Soluția laboratorului

- Bash înlocuiește variabila înainte de execuția lui `echo`.
 - `echo` primește doar textul rezultat în urma expansiunii.
 - Acoladele delimitează explicit numele variabilei și împiedică interpretările ambigue.
-

Ce trebuie să reții

- Parameter Expansion este realizată de Bash înainte de executarea programului.
- Programele executate nu văd expresiile `$VAR` sau `${VAR}`, ci doar rezultatul expansiunii.
- Forma `${VAR}` este recomandată atunci când variabila este urmată imediat de alte caractere sau când folosim forme avansate de expansiune.
- Parameter Expansion reprezintă baza pentru multe dintre mecanismele avansate ale limbajului Bash.

Capitolul 6

6.9 Parameter Expansion – Stabilirea valorilor implicite

„În multe scripturi, o variabilă poate să nu fie definită. Bash oferă mecanisme elegante prin care putem utiliza o valoare alternativă sau putem inițializa variabila în mod automat.”

Ce vei învăța

În acest capitol vei învăța cele mai utilizate forme de **Parameter Expansion** pentru gestionarea variabilelor nedefinite sau vide și vei vedea de ce acestea sunt foarte folosite în scripturile Bash.

Introducere

În capitolul anterior am văzut că Bash înlocuiește variabilele cu valorile lor.

Dar apare imediat o întrebare practică.

Ce se întâmplă dacă variabila nu există?

De exemplu:

```
echo "$NUME"
```

Dacă variabila NUME nu este definită, Bash nu produce o eroare.

Rezultatul este pur și simplu un șir gol.

În multe situații acest lucru este acceptabil.

În altele însă dorim un control mai bun.

Operatorul : -

Acesta este cel mai utilizat mecanism.

Sintaxa este:

```
${VAR: -va loare}
```

Semnificația este:

Dacă VAR este definită și nu este vidă, folosește valoarea ei.

În caz contrar:

Folosește va loare.

Exemplu

```
NUME="Daniel"
```

```
echo "${NUME: -Anonim}"
```

Rezultatul:

```
Daniel
```

Dacă executăm:

```
unset NUME
```

```
echo "${NUME: -Anonim}"
```

rezultatul devine:

```
Anonim
```

Observă că variabila **nu este modificată**.

Este folosită doar o valoare alternativă.

Operatorul :=

Acesta seamăna foarte mult cu precedentul.

Sintaxa:

```
${VAR:=valoare}
```

Diferența este esențială.

Dacă variabila nu există, Bash:

- utilizează valoarea specificată;
- **și o atribuie variabilei.**

Exemplu

```
unset NUME
```

```
echo "${NUME:=Anonim}"
```

Rezultatul:

```
Anonim
```

Dacă verificăm ulterior:

```
echo "$NUME"
```

obținem:

```
Anonim
```

Variabila a fost creată.

Operatorul :+

Acesta funcționează invers.

Sintaxa:

```
${VAR:+valoare}
```

Semnificația este:

Dacă variabila există și nu este vidă, afișează `valoare`.

Nu valoarea variabilei.

Ci textul specificat după operator.

Exemplu

```
NUME="Daniel"
```

```
echo "${NUME:+Variabila exista}"
```

Rezultatul:

```
Variabila exista
```

Dacă variabila nu există:

```
unset NUME
```

```
echo "${NUME:+Variabila exista}"
```

nu se afișează nimic.

Operatorul : ?

Acesta este foarte util în scripturi.

Sintaxa:

```
${VAR:?mesaj}
```

Dacă variabila nu există sau este vidă, Bash oprește execuția și afișează mesajul specificat.

Exemplu

```
echo "${HOME:?Variabila HOME lipseste}"
```

În mod normal nu apare nicio eroare.

Dar:

```
unset NUME
```

```
echo "${NUME:?Variabila NUME este obligatorie}"
```

produce un mesaj asemănător cu:

```
bash: NUME: Variabila NUME este obligatorie
```

Caseta „În spatele cortinei”

Aceste mecanisme sunt evaluate în etapa de **Parameter Expansion**.

Programul executat nu știe că ele au fost utilizate.

El primește doar rezultatul final.

Greșeală din lumea reală

Un script presupune că variabila:

```
BACKUP_DIR
```

există întotdeauna.

Într-o zi este executat dintr-un mediu diferit.

Variabila lipsește.

Scriptul începe să scrie fișiere într-o locație neașteptată.

O verificare precum:

```
"${BACKUP_DIR:?Variabila BACKUP_DIR trebuie definita}"
```

ar fi oprit imediat execuția și ar fi semnalat problema.

Administrator Notes

În scripturile de administrare există o regulă foarte bună:

Nu presupune că o variabilă există. Verifică acest lucru.

Operatorii `-`, `:`, `+=` și `?:` sunt printre cele mai utile instrumente pentru a face scripturile mai robuste și mai ușor de diagnosticat.

Legătura cu LPIC

Operatorii `-`, `:`, `+=` și `?:` apar frecvent în obiectivele LPIC-1, în special în întrebările despre scripting Bash. Nu este suficient să le memorezi sintaxa; este important să înțelegi diferența de comportament dintre ei.

Laborator 6.9

Obiectiv

Utilizarea operatorilor pentru valori implicite.

Cerințe

Execută:

```
unset NUME  
  
echo "${NUME:-Anonim}"
```

Apoi:

```
unset NUME  
  
echo "${NUME:=Anonim}"  
  
echo "$NUME"
```

Execută:

```
NUME="Daniel"  
  
echo "${NUME:+Variabila exista}"
```

Și:

```
unset NUME  
  
echo "${NUME:?Variabila este obligatorie}"
```

Explică diferențele dintre cele patru mecanisme.

Soluția laboratorului

- `:` - utilizează o valoare alternativă fără a modifica variabila.
 - `:=` utilizează valoarea alternativă și o atribuie variabilei.
 - `:+` produce textul specificat doar dacă variabila există.
 - `:?` oprește execuția și afișează un mesaj atunci când variabila lipsește.
-

Ce trebuie să reții

- `:` - → valoare implicită.
- `:=` → valoare implicită + atribuire.

- `:+ →` text alternativ dacă variabila există.
- `:? →` eroare dacă variabila lipsește.
- Aceste mecanisme sunt evaluate de Bash înainte de executarea programului.

Capitolul 6

6.10 Parameter Expansion – Lungimea variabilelor și manipularea șirurilor de caractere

„În Bash, o variabilă nu este doar o valoare care poate fi afișată. Ea poate fi analizată, scurtată, modificată și prelucrată fără a apela la programe externe precum `cut`, `sed` sau `awk`.”

Ce vei învăța

În acest capitol vei descoperi că Bash poate manipula șiruri de caractere folosind doar mecanismele proprii de **Parameter Expansion**, fără a lansa procese suplimentare.

Introducere

Până acum am folosit variabile astfel:

```
NUME="Daniel"
```

```
echo "$NUME"
```

Dar Parameter Expansion poate face mult mai mult.

Poate răspunde la întrebări precum:

- Câte caractere are această variabilă?
- Cum elimin extensia unui fișier?
- Cum elimin un prefix?
- Cum înlocuiesc un text cu altul?
- Cum obțin doar o parte din conținutul unei variabile?

Toate aceste operații sunt realizate de Bash înainte de executarea oricărui program.

Acesta este un avantaj important.

Nu este nevoie să lansăm procese externe pentru operații simple asupra șirurilor de caractere.

Lungimea unei variabile

Cea mai simplă operație este determinarea lungimii unui șir.

Sintaxa este:

```
${#VAR}
```

Exemplu

```
NUME="Linux"
```

```
echo "${#NUME}"
```

Rezultatul este:

5

Bash numără caracterele din valoarea variabilei.

Nu este lansat niciun program extern.

Încă un exemplu

```
FISIER="/home/daniel/document.txt"
```

```
echo "${#FISIER}"
```

Rezultatul va fi numărul total de caractere din șir.

Acest mecanism este foarte util în scripturi atunci când trebuie validate lungimi minime sau maxime.

Eliminarea unui prefix

Presupunem:

```
CALE="/home/daniel/document.txt"
```

Dorim să eliminăm partea:

```
/home/
```

Bash oferă operatorul:

```
${VAR#model}
```

Exemplu:

```
echo "${CALE#/home/}"
```

Rezultatul:

```
daniel/document.txt
```

Operatorul elimină cea mai scurtă potrivire a modelului de la începutul șirului.

Eliminarea celui mai lung prefix

Există și varianta:

```
${VAR##model}
```

Exemplu:

```
CALE="/home/daniel/document.txt"
```

```
echo "${CALE##*/}"
```

Rezultatul:

```
document.txt
```

Observă că modelul:

```
*/
```

elimină tot ceea ce precede ultimul caracter /.

Acesta este unul dintre cele mai utilizate exemple din scripting Bash.

Eliminarea unui sufix

Pentru sfârșitul șirului există operatorii:

```
${VAR%model}
```

și

```
${VAR%%model}
```

Exemplu:

```
FISIER="raport.pdf"
```

```
echo "${FISIER%.pdf}"
```

Rezultatul:

```
raport
```

Nu este nevoie de cut, basename sau sed.

Bash poate face această operație direct.

Înlocuirea unui text

Parameter Expansion permite și înlocuirea unor secvențe.

Sintaxa:

```
${VAR/text/alt_text}
```

Exemplu:

```
MESAJ="Linux este extraordinar"
```

```
echo "${MESAJ/Linux/Bash}"
```

Rezultatul:

```
Bash este extraordinar
```

Variabila originală nu este modificată.

Este produs doar rezultatul expansiunii.

Caseta „În spatele cortinei”

De ce există aceste mecanisme?

În primele versiuni ale shell-urilor Unix, multe operații asupra textului necesitau lansarea unor programe externe.

De exemplu:

- cut
- expr
- sed
- awk

Aceste programe rămân esențiale și astăzi, mai ales pentru prelucrări complexe.

Totuși, pentru operații simple asupra conținutului unei variabile, Bash oferă mecanisme integrate care evită lansarea unor procese suplimentare. În scripturile care execută aceste operații de foarte multe ori, soluția bazată pe *Parameter Expansion* este, de regulă, mai eficientă și mai ușor de citit.

Greșeală din lumea reală

Un script conține:

```
NUME=$(basename "$SCALE")
```

Funcționează.

Dar dacă singurul scop este eliminarea căii dintr-o variabilă deja existentă, aceeași operație poate fi realizată direct în Bash:

```
NUME="${CALE##*/}"
```

Aceasta evită lansarea unui proces extern și exprimă foarte clar intenția autorului.

Notă: Nu afirmăm că `basename` este greșit. El rămâne alegerea potrivită în multe situații, în special când lucrăm cu argumente provenite din exterior sau când claritatea codului este prioritară. Scopul acestui exemplu este să arătăm că Bash oferă și o alternativă integrată.

Administrator Notes

Parameter Expansion este foarte puternic, dar nu trebuie folosit doar pentru că este disponibil.

Dacă o expresie devine greu de citit, întreabă-te dacă un utilitar precum `sed`, `awk` sau `basename` nu exprimă mai clar intenția codului.

Un script bun este cel care poate fi înțeles ușor de colegii care îl vor întreține peste câteva luni.

Legătura cu LPIC

Operatorii `${#VAR}`, `${VAR#...}`, `${VAR##...}`, `${VAR%...}` și `${VAR%%...}` apar în obiectivele LPIC deoarece reprezintă funcționalități native ale Bash. Examenul testează frecvent capacitatea de a recunoaște rezultatul unei astfel de expansiuni.

Cum aş depana această problemă?

Problema

Scriptul produce un nume de fișier sau o cale incorectă.

Pași

1. Afișez mai întâi valoarea variabilei originale.
 2. Testez separat expresia de *Parameter Expansion*.
 3. Verific dacă modelul folosit (*, sufixul sau prefixul) exprimă exact ceea ce doresc să elimin.
 4. Confirm că rezultatul este cel așteptat înainte de a continua execuția scriptului.
-

Laborator 6.10

Obiectiv

Utilizarea operatorilor pentru manipularea șirurilor de caractere.

Cerințe

Definește:

```
CALE="/home/student/documente/raport.pdf"
```

Execută:

```
echo "${#CALE}"
```

```
echo "${CALE##*/}"
```

```
echo "${CALE%. *}"
```

```
echo "${CALE/documente/Documents}"
```

Răspunde:

1. Ce operație realizează fiecare expresie?
 2. Este modificată variabila CALE?
 3. În ce situații ai prefera aceste mecanisme în locul unui utilitar extern?
-

Soluția laboratorului

- `${#CALE}` întoarce lungimea șirului.
 - `${CALE##*/}` extrage numele fișierului.
 - `${CALE%. *}` elimină extensia.
 - `${CALE/documente/Documents}` produce un șir nou în care primul text găsit este înlocuit.
 - Variabila CALE rămâne neschimbată în toate exemplele.
-

Ce trebuie să reții

- Bash poate manipula șiruri de caractere fără a lansa programe externe.
- Operatorii de *Parameter Expansion* permit măsurarea lungimii, eliminarea prefixelor și sufixelor și înlocuirea textului.
- Aceste mecanisme produc un rezultat nou; ele nu modifică automat variabila originală.
- Alege întotdeauna soluția care oferă cel mai bun echilibru între eficiență și lizibilitate.

Capitolul 6

6.11 Parameter Expansion – Extragerea unei porțiuni dintr-un șir (Substring Expansion)

„Uneori nu avem nevoie de întreaga valoare a unei variabile. Bash ne permite să extragem doar o parte din conținutul ei, fără a utiliza utilitare externe.”

Ce vei învăța

În acest capitol vei învăța cum poate Bash extrage o porțiune din conținutul unei variabile și de ce această tehnică este utilă în scripturile de administrare.

Introducere

Până acum am învățat să:

- afișăm variabile;
- determinăm lungimea lor;
- eliminăm prefixe;
- eliminăm sufixe;
- înlocuim text.

Mai există însă o operație foarte utilă.

Cum extragem doar o parte dintr-un șir?

De exemplu, dacă variabila conține:

Linux Essentials

cum obținem doar:

Linux

sau doar:

Essentials

Bash poate face acest lucru direct.

Sintaxa

Forma generală este:

```
${VAR:offset}
```

sau

```
${VAR:offset:length}
```

unde:

- **offset** reprezintă poziția de la care începe extragerea;
 - **length** reprezintă numărul de caractere care vor fi extrase.
-

Primul exemplu

```
TEXT="Linux Essentials"
```

```
echo "${TEXT:0:5}"
```

Rezultatul:

Linux

Explicație:

Pornim de la poziția **0** și extragem **5** caractere.

Al doilea exemplu

```
TEXT="Linux Essentials"
```

```
echo "${TEXT:6}"
```

Rezultatul:

Essentials

Observă că nu am specificat lungimea.

În această situație Bash extrage toate caracterele până la sfârșitul șirului.

Un model mental

Imaginează-ți că fiecare caracter primește un indice.

```
L i n u x _ E s s e n t i a l s  
0 1 2 3 4 5 6 7 8 9 ...
```

Parameter Expansion nu caută cuvinte.

Nu caută delimitatori.

Lucrează exclusiv cu pozițiile caracterelor.

Acesta este un aspect important.

Exemple practice

Primele patru caractere

```
echo "${TEXT:0:4}"
```

Rezultat:

```
Linu
```

Ultimele caractere

Presupunem:

```
FISIER="raport_final.pdf"
```

Putem extrage:

```
echo "${FISIER:7}"
```

Rezultatul depinde exclusiv de poziția caracterelor.

Nu contează dacă textul reprezintă un nume de fișier, o adresă URL sau orice alt șir.

Atenție

Substring Expansion **nu înțelege semnificația datelor**.

Dacă mutăm sau schimbăm conținutul variabilei, pozițiile se modifică.

De aceea această tehnică este recomandată atunci când structura șirului este cunoscută și stabilă.

Dacă trebuie să analizăm un format variabil, există adesea soluții mai potrivite (`cut`, `awk`, expresii regulate sau alte forme de Parameter Expansion).

Caseta „În spatele cortinei”

Substring Expansion este evaluată complet în interiorul Bash.

Nu este lansat niciun proces nou.

Acesta este unul dintre motivele pentru care este foarte eficientă în scripturile care execută aceeași operație de foarte multe ori.

Totuși, eficiența nu trebuie să fie singurul criteriu. Un script ușor de înțeles este, de regulă, mai valoros decât unul ușor mai rapid, dar greu de întreținut.

Greșeală din lumea reală

Un administrator încearcă să elimine extensia unui fișier folosind poziții fixe:

```
echo "${FISIER:0:10}"
```

Scriptul funcționează doar pentru anumite nume de fișiere.

Mai târziu apar fișiere cu lungimi diferite, iar rezultatul devine incorect.

În această situație, operatorul:

```
${FISIER%.*}
```

este mai robust, deoarece se bazează pe structură, nu pe poziții fixe.

Acesta este un exemplu bun în care trebuie ales mecanismul potrivit, nu doar primul care funcționează.

Administrator Notes

Folosește Substring Expansion atunci când pozițiile sunt cunoscute și constante.

Dacă informația este delimitată de caractere precum /, ., : sau -, întreabă-te dacă unul dintre operatorii `${VAR#...}` sau `${VAR%...}` exprimă mai bine intenția codului.

În Bash există adesea mai multe soluții corecte; alege-o pe cea mai clară.

Legătura cu LPIC

Substring Expansion apare în obiectivele LPIC deoarece este un mecanism nativ al Bash. Examenul urmărește, de regulă, să verifice dacă poți interpreta rezultatul unei expresii precum `${VAR:3:5}`, nu dacă memorezi toate variantele posibile.

Laborator 6.11

Obiectiv

Extragerea unor porțiuni dintr-un șir.

Cerințe

Definește:

```
TEXT="Linux Essentials"
```

Execută:

```
echo "${TEXT:0:5}"
```

```
echo "${TEXT:6}"
```

```
echo "${TEXT:6:4}"
```

Apoi definește:

```
DATA="2026-08-04"
```

și extrage:

- anul;
- luna;
- ziua.

Întrebare de reflecție

Poți rezolva aceeași problemă folosind `cut`?

Care dintre soluții ți se pare mai ușor de înțeles și de întreținut?

Soluția laboratorului

- `${TEXT:0:5}` → Linux
- `${TEXT:6}` → Essentials
- `${TEXT:6:4}` → Esse
- `${DATA:0:4}` → 2026
- `${DATA:5:2}` → 08
- `${DATA:8:2}` → 04

Observă că toate operațiile sunt realizate de Bash, fără lansarea unor programe externe.

Ce trebuie să reții

- `${VAR:offset}` extrage șirul începând de la poziția indicată.
- `${VAR:offset:length}` extrage un număr precis de caractere.
- Operația se bazează pe poziții, nu pe semnificația textului.
- Pentru structuri variabile, alte mecanisme de Parameter Expansion sau utilitare dedicate pot exprima mai bine intenția.

Capitolul 6

6.12 Command Substitution – Cum folosim rezultatul unei comenzi în altă comandă

„În Bash, rezultatul unei comenzi poate deveni argument pentru o altă comandă. Acest mecanism poartă numele de Command Substitution și reprezintă unul dintre cele mai elegante instrumente ale shell-ului.”

Ce vei învăța

În acest capitol vei înțelege cum execută Bash o comandă, capturează rezultatul acesteia și îl utilizează ca parte a unei alte comenzi.

Introducere

Până acum am studiat Parameter Expansion.

Aceasta înlocuiește:

\$NUME

cu valoarea unei variabile.

Dar apare imediat o întrebare.

Dacă în locul unei variabile vrem să folosim rezultatul unei comenzi?

De exemplu.

Dorim să afișăm data curentă.

Putem executa:

date

Dar dacă dorim să construim propoziția:

Astăzi este: Mon Aug 4 ...

Cum procedăm?

Pentru astfel de situații Bash oferă **Command Substitution**.

Ideea fundamentală

Command Substitution înseamnă:

Execută această comandă și folosește rezultatul ei ca text.

Nu ca proces.

Nu ca obiect.

Ci ca text.

Aceasta este ideea esențială.

Primul exemplu

Executăm:

```
echo "Astăzi este $(date)"
```

Rezultatul poate fi:

Astăzi este Mon Aug 4 15:20:18 EEST 2026

Ce s-a întâmplat?

Bash a executat:

```
date
```

A preluat rezultatul.

A construit noua comandă:

```
echo "Astăzi este Mon Aug 4 15:20:18 EEST 2026"
```

Abia apoi a lansat programul echo.

Modelul mental

Utilizator

|

```
echo $(date)
```

|

▼

Bash execută

```
date
```

|

▼

Primește rezultatul

|

▼

Construiește noua comandă

|

▼

```
echo "Mon Aug ..."
```

|

▼

```
execve()
```

|

▼

Programul echo

Observă că **echo** nu execută niciodată comanda **date**.

Bash face acest lucru înainte.

Încă un exemplu

```
mkdir "$(date +%Y-%m-%d)"
```

Rezultatul poate fi:

```
2026-08-04
```

ca nume al directorului.

Programul `mkdir` nu știe că ai folosit `date`.

Primește doar numele rezultat după expansiune.

Command Substitution poate apărea aproape oriunde

De exemplu:

```
cp raport.txt "backup-$(date +%F).txt"
```

sau

```
echo "Kernel: $(uname -r)"
```

sau

```
echo "Utilizator: $(whoami)"
```

Toate urmează același mecanism.

Forma veche

În documentația mai veche vei întâlni și sintaxa:

```
`date`
```

(backticks)

Aceasta funcționează și astăzi.

Totuși, în scripturile moderne este recomandată forma:

`$(date)`

De ce?

Pentru că este:

- mai ușor de citit;
 - mai ușor de imbricat;
 - mai puțin predispusă la erori.
-

Un exemplu

Forma modernă:

```
echo "$(date)"
```

este mult mai clară decât:

```
echo "`date`"
```

Diferența devine și mai evidentă atunci când comenzile sunt complexe sau conțin la rândul lor ghilimele.

Caseta „În spatele cortinei”

Command Substitution este evaluată înainte de execuția comenzii principale.

Rezultatul comenzii este capturat de Bash printr-un mecanism intern și introdus în linia de comandă sub forma unui șir de caractere.

Programul executat nu știe că acel text provine din executarea unei alte comenzi.

El primește doar argumentele finale.

Greșeală din lumea reală

Un utilizator scrie:

```
echo "$(ls)"
```

și se miră că rezultatul apare pe o singură linie.

Acest lucru se întâmplă deoarece Bash capturează ieșirea standard a comenzii și o introduce în expresia finală. Modul în care apar caracterele de sfârșit de linie depinde apoi de context și de programul care primește rezultatul.

Acest exemplu arată că *Command Substitution* nu este doar „copierea” ieșirii unei comenzi, ci face parte din procesul de construire a noii linii de comandă.

Notă: Nu intrăm încă în detalii despre eliminarea caracterelor finale de newline. `man bash` discută acest comportament, dar îl putem trata într-un subcapitol separat despre subtilitățile *Command Substitution*, pentru a nu încălca primul contact cu mecanismul.

Administrator Notes

Folosește întotdeauna forma:

`$( ... )`

în scripturile noi.

Pe lângă faptul că este recomandată de documentația modernă, ea face codul mai ușor de citit și simplifică expresiile imbricate.

Legătura cu LPIC

Command Substitution apare frecvent în exercițiile LPIC și în scripturile de administrare.

Examenul urmărește în special să verifice dacă înțelegi **momentul în care este executată comanda** și faptul că rezultatul devine parte din argumentele comenzii principale.

Cum aş depana această problemă?

Problema

Rezultatul unei comenzi nu apare acolo unde mă aștept.

Pașii

1. Execut separat comanda din interiorul `$( ... )`.
 2. Verific ieșirea produsă.
 3. Analizez cum este utilizată această ieșire în comanda principală.
 4. Dacă rezultatul conține spații, verific și quoting-ul (`"$( ... )"` este adesea alegerea corectă).
-

Laborator 6.12

Obiectiv

Utilizarea *Command Substitution*.

Cerințe

Execută:

```
echo "$(date)"
```

```
echo "Kernel: $(uname -r)"
```

```
echo "Utilizator: $(whoami)"
```

Creează apoi un director folosind data curentă:

```
mkdir "backup-$(date +%F)"
```

Răspunde:

1. Ce comandă este executată prima?
 2. Ce argument primește comanda principală?
 3. Ce s-ar întâmpla dacă ai elimina ghilimelele?
-

Soluția laboratorului

- Bash execută mai întâi comanda aflată în `$( . . . )`.
 - Rezultatul acesteia este introdus în linia de comandă.
 - Abia apoi este lansată comanda principală.
 - În multe situații, ghilimelele sunt recomandate pentru a păstra rezultatul ca un singur argument.
-

Ce trebuie să reții

- `$( . . . )` execută o comandă și folosește rezultatul ei ca text.
- Command Substitution este realizată de Bash înainte de execuția comenzii principale.
- Forma `$( . . . )` este recomandată în locul backticks.
- În practică, este recomandat să folosești `"$( )"` atunci când rezultatul poate conține spații sau alte caractere care ar putea influența împărțirea în argumente.

Capitolul 6

6.13 Command Substitution – Imbricarea comenzilor și bune practici

„O comandă poate utiliza rezultatul altei comenzi, iar aceasta, la rândul ei, poate depinde de rezultatul unei a treia comenzi. Bash permite construirea acestor relații într-un mod natural prin Command Substitution.”

Ce vei învăța

În acest capitol vei înțelege cum poate Bash executa comenzi imbricate și vei învăța câteva reguli simple care fac scripturile mai ușor de citit și de întreținut.

Introducere

În capitolul anterior am văzut că Bash poate executa o comandă și poate utiliza rezultatul ei.

De exemplu:

```
echo "$(date)"
```

Acum apare o întrebare firească.

Dacă rezultatul unei comenzi depinde de rezultatul altei comenzi?

Răspunsul este simplu.

Bash permite **imbricarea** (*nesting*) expresiilor `$( . . . )`.

Primul exemplu

Executăm:

```
echo "$(basename "$(pwd)")"
```

La prima vedere pare complicat.

Dar dacă îl analizăm pas cu pas, devine foarte simplu.

Pasul 1

Bash identifică expresia cea mai interioară.

```
$(pwd)
```

și o execută.

Rezultatul poate fi:

```
/home/daniel/proiecte
```

Pasul 2

Noua comandă devine:

```
basename "/home/daniel/proiecte"
```

Bash execută această comandă.

Rezultatul este:

```
proiecte
```

Pasul 3

Comanda finală devine:

```
echo "proiecte"
```

Acum este lansat programul echo.

Modelul mental

```
echo $( basename $( pwd ) )
```

|



execută pwd

|



primește calea curentă

```
|
▼
execută basename

|
▼
primește ultimul director

|
▼
execută echo
```

Observă că Bash rezolvă expresiile din interior către exterior.

Acesta este un principiu pe care îl vom regăsi și în alte mecanisme ale shell-ului.

Încă un exemplu

```
mkdir "$(date +%Y)/$(date +%m)"
```

Dacă directorul nu există încă, putem crea rapid o structură organizată pe ani și luni.

Mai întâi sunt executate cele două comenzi `date`.

Rezultatele lor sunt apoi folosite pentru construirea argumentului transmis către `mkdir`.

Imbricarea nu este un scop în sine

Faptul că Bash permite expresii foarte complexe nu înseamnă că trebuie să le folosim întotdeauna.

Compară:

```
echo "$(basename "$(dirname "$(readlink -f "$FISIER")")")"
```

cu:

```
CALE_REALA="$(readlink -f "$FISIER")"
DIRECTOR="$(dirname "$CALE_REALA")"
NUME="$(basename "$DIRECTOR")"
```

```
echo "$NUME"
```

Ambele variante pot produce același rezultat.

A doua este însă mult mai ușor de citit și de depanat.

Caseta „În spatele cortinei”

Bash execută de la interior către exterior

Acest comportament seamănă cu evaluarea expresiilor matematice.

În expresia:

$(2 + (3 \times 4))$

calculăm mai întâi:

3×4

și apoi:

$2 + \text{rezultat}$

La fel procedează și Bash.

Rezolvă întâi expresiile cele mai interioare, apoi continuă către exterior.

Greșeală din lumea reală

Un administrator scrie o comandă cu patru niveluri de $\$(\dots)$.

Funcționează.

Dar peste șase luni nici el nu mai înțelege ce face.

Problema nu este Bash.

Problema este lizibilitatea codului.

Uneori câteva variabile intermediare fac scriptul mult mai ușor de întreținut.

Administrator Notes

Un principiu folosit în multe echipe DevOps este:

Dacă trebuie să urmărești cu degetul parantezele pentru a înțelege expresia, probabil este timpul să o împarți în pași mai mici.

Bash permite expresii foarte sofisticate.

Un administrator bun știe și când să nu le folosească.

Legătura cu LPIC

LPIC verifică înțelegerea mecanismului \$(. . .), însă în practică este la fel de important să poți citi și înțelege scripturile scrise de alți administratori.

Capacitatea de a urmări execuția din interior către exterior este esențială.

Cum aş depana această problemă?

Problema

O expresie cu mai multe niveluri de \$(. . .) nu produce rezultatul dorit.

Paşii

1. Execut separat expresia cea mai interioară.
2. Verific rezultatul.
3. Înlocuiesc expresia cu rezultatul obținut și continui analiza.
4. Repet procesul până ajung la comanda finală.

Acest mod de lucru simplifică foarte mult depanarea.

Laborator 6.13

Obiectiv

Înțelegerea execuției expresiilor imbricate.

Cerințe

Execută:

```
echo "$(basename "$(pwd))"
```

Apoi:

```
echo "$(dirname "$(pwd))"
```

și:

```
echo "$(basename "$(dirname "$(pwd))")"
```

Pentru fiecare comandă răspunde:

1. Care este prima comandă executată?

2. Ce rezultat produce?
 3. Cum se transformă comanda înainte de execuția lui `echo`?
-

Soluția laboratorului

- Bash execută întotdeauna expresia cea mai interioară.
 - Rezultatul acesteia este introdus în expresia următoare.
 - Procesul continuă până când comanda principală primește toate argumentele finale.
 - Programul `echo` nu execută comenzile din `$( . . . )`; el primește doar rezultatul lor.
-

Ce trebuie să reții

- Expresiile `$( . . . )` pot fi imbricate.
- Bash le evaluează din interior către exterior.
- Imbricarea este un instrument puternic, dar trebuie folosită cu măsură.
- Pentru expresii complexe, variabilele intermediare cresc lizibilitatea și ușurează depanarea.

Capitolul 6

6.14 Arithmetic Expansion – Cum efectuează Bash calcule matematice

„Bash nu este doar un interpretor de comenzi. El poate evalua și expresii aritmetice, iar rezultatul acestora poate fi utilizat oriunde este permisă o expansiune.”

Ce vei învăța

În acest capitol vei înțelege cum efectuează Bash calcule matematice folosind mecanismul **Arithmetic Expansion** și când este recomandată utilizarea lui în scripturi.

Introducere

În capitolele anterioare am văzut că Bash poate înlocui:

- valoarea unei variabile;

- rezultatul unei comenzi.

Mai există însă un tip important de expansiune.

Presupunem că dorim să afișăm rezultatul expresiei:

`25 + 17`

O soluție ar fi să folosim un program extern precum `expr`.

Dar Bash poate face acest lucru singur.

Pentru astfel de situații există **Arithmetic Expansion**.

Sintaxa

Forma generală este:

`$((expresie))`

Observă cele două perechi de paranteze.

Acestea indică faptul că Bash trebuie să evalueze expresia ca pe o expresie aritmetică.

Primul exemplu

Executăm:

```
echo $((2 + 3))
```

Rezultatul este:

5

Nu este lansat niciun program extern.

Bash evaluează expresia și transmite rezultatul către `echo`.

Expresiile pot conține variabile

Presupunem:

```
A=15  
B=7
```

Executăm:

```
echo $((A + B))
```

Rezultatul este:

22

Observă un detaliu interesant.

În interiorul expresiei aritmetice nu este obligatoriu să scriem:

\$A

Putem scrie pur și simplu:

A

Bash recunoaște automat numele variabilelor.

Acesta este un comportament specific **Arithmetic Expansion**.

Operatorii disponibili

Arithmetic Expansion suportă majoritatea operatorilor utilizați în limbajele de programare.

Cele mai folosite sunt:

Operator	Semnificație
+	adunare
-	scădere
*	înmulțire
/	împărțire între numere întregi
%	restul împărțirii (modulo)

Exemple:

```
echo $((20 - 5))
```

```
echo $((6 * 8))
```

```
echo $((20 / 4))
```

```
echo $((17 % 5))
```

Ordinea evaluării

La fel ca în matematică, Bash respectă prioritatea operatorilor.

Exemplu:

```
echo $((2 + 3 * 4))
```

Rezultatul este:

14

deoarece înmulțirea este evaluată înaintea adunării.

Dacă dorim o altă ordine, folosim paranteze:

```
echo $(( (2 + 3) * 4 ))
```

Rezultatul devine:

20

Caseta „În spatele cortinei”

Arithmetic Expansion este evaluată în întregime de Bash.

Programul care primește rezultatul nu știe că acesta provine dintr-un calcul.

De exemplu:

```
sleep $(5 * 60)
```

Programul `sleep` primește pur și simplu argumentul:

300

Nu știe că utilizatorul a scris o expresie matematică.

Greșeală din lumea reală

Un utilizator încearcă:

```
echo $((10 / 3))
```

și se așteaptă la:

3.333333

Rezultatul este însă:

3

Arithmetic Expansion lucrează cu **numere întregi**.

Dacă sunt necesare calcule cu virgulă mobilă, se folosesc de regulă utilitare precum `bc` sau limbaje de scripting dedicate.

Administrator Notes

Arithmetic Expansion este excelentă pentru:

- contoare;
- calcule de dimensiuni;
- conversii simple;
- operații asupra datelor calendaristice;
- scripturi de administrare.

Nu încerca însă să o folosești pentru calcule științifice sau financiare complexe.

Legătura cu LPIC

Arithmetic Expansion este inclusă în obiectivele LPIC deoarece apare frecvent în scripturile Bash.

Examenul urmărește în special:

- sintaxa `$ ( . . . )`;
 - utilizarea variabilelor în expresii;
 - comportamentul împărțirii între numere întregi.
-

Cum aş depana această problemă?

Problema

Rezultatul unui calcul este diferit de cel așteptat.

Pași

1. Verific dacă toate variabilele conțin valori numerice.
 2. Analizez ordinea operatorilor.
 3. Adaug paranteze dacă expresia poate fi interpretată în mai multe moduri.
 4. Dacă am nevoie de valori zecimale, verific dacă Bash este instrumentul potrivit pentru acel calcul.
-

Laborator 6.14

Obiectiv

Utilizarea Arithmetic Expansion.

Cerințe

Definește:

A=12

B=5

Execută:

```
echo $((A + B))
```

```
echo $((A - B))
```

```
echo $((A * B))
```

```
echo $((A / B))
```

```
echo $((A % B))
```

Apoi compară rezultatele:

```
echo $((2 + 3 * 4))
```

și

```
echo $(((2 + 3) * 4))
```

Explică diferența.

Soluția laboratorului

- Bash efectuează calculele înainte de execuția comenzii **echo**.
 - Variabilele pot fi utilizate direct în expresiile aritmetice.
 - Împărțirea produce un rezultat întreg.
 - Parantezele modifică ordinea evaluării operatorilor.
-

Ce trebuie să reții

- Arithmetic Expansion utilizează sintaxa `$((...))`.
- Expresiile sunt evaluate de Bash înainte de execuția programului.
- Variabilele pot fi folosite fără prefixul `$` în interiorul expresiei.
- Bash efectuează calcule cu numere întregi.
- Parantezele permit controlul ordinii evaluării.

Capitolul 6

6.15 Ordinea expansiunilor – Cum transformă Bash o linie de comandă

„Atunci când apăsăm Enter, Bash nu execută imediat comanda. Înainte de lansarea programului, shell-ul aplică o serie de transformări într-o ordine bine definită. Înțelegerea acestei ordini explică aproape toate comportamentele aparent misterioase ale Bash.”

Ce vei învăța

În acest capitol vei înțelege că Bash nu aplică expansiunile la întâmplare. Există o ordine precisă, iar această ordine influențează rezultatul final al unei comenzi.

Introducere

Să analizăm următoarea comandă:

```
echo "$HOME/$(date +%Y)/$((2+3))"
```

La prima vedere pare simplă.

Dar Bash trebuie să răspundă la mai multe întrebări.

- Care este valoarea lui `$HOME`?
- Care este rezultatul comenzii `date`?
- Care este rezultatul expresiei matematice?
- Unde începe și unde se termină fiecare argument?

Toate aceste transformări trebuie făcute înainte ca programul `echo` să fie executat.

Ele nu sunt aplicate aleator.

Bash urmează o ordine foarte precisă.

Un model mental

Imaginează-ți că Bash lucrează asemenea unui atelier.

Linia de comandă trece prin mai multe stații de lucru.

Fiecare stație efectuează o singură operație.

Abia la final comanda este gata pentru execuție.

Utilizator

|



Analiza sintactică

|



Expansiuni

|



Word Splitting

|



Pathname Expansion

|



Quote Removal

|



execve()

|



Program

Acesta este modelul mental pe care îl vom folosi în toate capitolele următoare.

Ce am studiat deja?

Până acum am analizat trei tipuri importante de expansiuni.

- Parameter Expansion

\$HOME

- Command Substitution

`$(date)`

- Arithmetic Expansion

`$((2+3))`

Mai există însă și alte transformări pe care încă nu le-am studiat.

De exemplu:

- Word Splitting
- Pathname Expansion (Globbing)
- Quote Removal

Acestea urmează imediat.

Un exemplu complet

Presupunem:

```
DIR="Documente"
```

```
echo "$HOME/$DIR/$(date +%Y)"
```

Bash nu execută imediat `echo`.

Mai întâi:

1. expandează `$HOME`;
2. expandează `$DIR`;
3. execută `date`;
4. construiește argumentul final;
5. elimină ghilimelele utilizate pentru quoting;
6. lansează programul `echo`.

Programul `echo` nu vede niciodată variabilele și nici expresia `$(date)`.

Primește doar textul rezultat.

Un lucru foarte important

Foarte mulți utilizatori își imaginează că Bash citește linia de la stânga la dreapta și execută imediat fiecare element.

Nu așa funcționează.

Bash:

- analizează;
- transformă;
- verifică;
- abia apoi execută.

Această idee explică foarte multe dintre capitolele pe care le-am studiat deja.

Caseta „În spatele cortinei”

De ce există această ordine?

Pentru că unele transformări depind de rezultatul altora.

De exemplu:

Command Substitution poate produce un text care conține spații.

Aceste spații pot influența etapa de Word Splitting.

La rândul ei, Word Splitting poate produce cuvinte care vor participa ulterior la Pathname Expansion.

Dacă Bash ar modifica această ordine, multe scripturi existente nu ar mai funcționa corect.

Compatibilitatea este unul dintre motivele pentru care această succesiune este foarte stabilă și este documentată în `man bash`.

Greșeală din lumea reală

Un administrator spune:

„Nu înțeleg de ce această comandă funcționează diferit atunci când pun ghilimele.”

Problema nu este quoting-ul în sine.

Problema este că ghilimelele modifică una dintre etapele prin care trece linia de comandă.

Înțelegerea ordinii expansiunilor explică astfel de diferențe fără a fi nevoie să memorăm cazuri particulare.

Administrator Notes

Când depanezi o comandă complicată, nu încerca să înțelegi totul dintr-o dată.

Întreabă-te:

1. Ce expansiuni sunt prezente?
2. În ce ordine sunt evaluate?
3. Ce text rezultă după fiecare etapă?

Această abordare este folosită frecvent de administratorii experimentați și simplifică mult analiza comenzilor complexe.

Legătura cu LPIC

Examenul LPIC nu cere, de regulă, să enumeri toate etapele procesării unei comenzi.

În schimb, multe întrebări presupun că înțelegi această ordine.

Un răspuns corect depinde adesea de identificarea mecanismului care este aplicat primul.

Cum aş depana această problemă?

Problema

O comandă produce un rezultat diferit de cel așteptat.

Pașii

1. Identific toate tipurile de expansiuni din comandă.
2. Analizez fiecare transformare separat.
3. Verific rezultatul obținut după fiecare etapă.
4. Confirm argumentele finale care vor fi transmise programului.

Foarte multe probleme devin evidente după această analiză.

Laborator 6.15

Obiectiv

Observarea faptului că Bash transformă comanda în mai multe etape.

Cerințe

Definește:

DIR="Documente"

Execută:

```
echo "$HOME/$DIR"
```

Apoi:

```
echo "$(date +%Y)"
```

și:

```
echo "$((20+5))"
```

În final:

```
echo "$HOME/$(date +%Y)/$((20+5))"
```

Pentru fiecare exemplu răspunde:

1. Ce tipuri de expansiuni sunt utilizate?
 2. În ce ordine sunt evaluate?
 3. Ce argument primește programul `echo`?
-

Soluția laboratorului

- Bash identifică mecanismele de expansiune prezente în comandă.
 - Fiecare expansiune este evaluată înainte de execuția programului.
 - Rezultatul final este transmis către `echo`.
 - Programul executat nu vede expresiile originale, ci doar textul rezultat după toate transformările.
-

Ce trebuie să reții

- Bash transformă linia de comandă în mai multe etape.
- Expansiunile sunt evaluate într-o ordine bine definită.
- Programele executate primesc doar rezultatul final al acestor transformări.
- Înțelegerea ordinii expansiunilor este una dintre cheile stăpânirii Bash.

Capitolul 6

6.16 Word Splitting – Cum transformă Bash un șir în mai multe argumente

„După ce Bash termină Parameter Expansion, Command Substitution și Arithmetic Expansion, poate urma o etapă foarte importantă: împărțirea textului în cuvinte. Acest mecanism poartă numele de Word Splitting.”

Ce vei învăța

În acest capitol vei înțelege când Bash împarte un șir de caractere în mai multe argumente și de ce utilizarea ghilimelelor duble previne adesea erori în scripturi.

Introducere

Să presupunem că avem variabila:

```
NUME="Linux Essentials"
```

și executăm:

```
echo $NUME
```

Mulți utilizatori cred că Bash transmite către `echo` un singur argument.

În realitate, lucrurile sunt puțin mai complicate.

După ce realizează **Parameter Expansion**, Bash obține textul:

```
Linux Essentials
```

Acum apare întrebarea:

Este acesta un singur argument sau două argumente?

Răspunsul depinde de etapa numită **Word Splitting**.

Ideea fundamentală

Word Splitting înseamnă:

Bash poate împărți rezultatul unei expansiuni în mai multe cuvinte, folosind anumite caractere separatoare.

Aceste cuvinte devin argumentele transmise programului.

Programul executat nu știe că această împărțire a avut loc.

Primul exemplu

Definim:

```
NUME="Linux Essentials"
```

Executăm:

```
echo $NUME
```

După Parameter Expansion:

```
Linux Essentials
```

După Word Splitting:

```
Linux  
Essentials
```

Programul `echo` primește două argumente.

Același exemplu cu ghilimele

Executăm:

```
echo "$NUME"
```

Parameter Expansion produce același rezultat:

```
Linux Essentials
```

Dar ghilimelele împiedică Word Splitting.

Programul primește:

```
Linux Essentials
```

ca **un singur argument**.

Aceasta este una dintre cele mai importante funcții ale ghilimelelor duble.

Cine decide unde se face împărțirea?

Implicit, Bash utilizează conținutul variabilei speciale:

IFS

(Internal Field Separator)

În mod normal aceasta conține:

- spațiul;
- TAB-ul;
- caracterul newline.

Prin urmare:

Ana Maria Popescu

poate deveni:

Ana

Maria

Popescu

dacă nu este protejat prin quoting.

Un model mental

Imaginează-ți următoarele etape:

Parameter Expansion

↓

Linux Essentials

↓

Word Splitting

↓

Linux

Essentials

↓

execve()

Observă că Word Splitting nu lucrează asupra textului introdus de utilizator.

Lucrează asupra rezultatului expansiunilor.

Aceasta este o diferență foarte importantă.

Un exemplu practic

Presupunem:

```
FISIER="/home/daniel/My Documents/raport.txt"
```

Executăm:

```
cat $FISIER
```

Bash vede:

```
/home/daniel/My Documents/raport.txt
```

Word Splitting produce:

```
/home/daniel/My
```

```
Documents/raport.txt
```

`cat` primește două argumente și comanda eșuează.

Forma recomandată este:

```
cat "$FISIER"
```

Acum rezultatul expansiunii este păstrat ca un singur argument.

Caseta „În spatele cortinei”

De ce există Word Splitting?

Pentru că shell-ul a fost proiectat să permită construirea dinamică a listei de argumente.

De exemplu:

```
LISTA="fisier1 fisier2 fisier3"
```

```
rm $LISTA
```

În această situație este util ca Bash să transforme conținutul variabilei în trei argumente distincte.

Același mecanism însă poate produce erori dacă este aplicat asupra unor căi care conțin spații.

Greșeală din lumea reală

Un script funcționează luni de zile.

Într-o zi apare un director numit:

Backup August

Scriptul începe să dea erori.

Problema nu este `cp`.

Problema nu este `mv`.

Problema este Word Splitting.

Variabila nu era încadrată între ghilimele duble.

Aceasta este una dintre cele mai frecvente cauze ale erorilor în scripturile Bash.

Administrator Notes

Există o recomandare întâlnită în aproape toate ghidurile moderne de Bash:

Pune aproape întotdeauna variabilele între ghilimele duble atunci când reprezintă nume de fișiere, directoare sau texte introduse de utilizator.

Există excepții, pe care le vom studia mai târziu.

Dar aceasta este una dintre cele mai valoroase reguli pe care le poate învăța un administrator Linux.

Legătura cu LPIC

Word Splitting apare frecvent indirect în întrebările LPIC.

Examenul prezintă adesea un script aparent corect și cere identificarea motivului pentru care acesta eșuează atunci când apar spații în numele fișierelor.

În foarte multe cazuri, explicația este Word Splitting.

Cum aş depana această problemă?

Problema

O comandă funcționează pentru unele fișiere, dar nu pentru cele care conțin spații.

Pașii

1. Verific dacă argumentele provin din variabile.
2. Verific dacă variabilele sunt încadrate între ghilimele duble.
3. Analizez dacă Word Splitting este aplicat după expansiune.
4. Testez din nou comanda folosind:

"\$VAR"

În foarte multe situații problema dispare imediat.

Laborator 6.16

Obiectiv

Observarea efectului Word Splitting.

Cerințe

Execută:

```
NUME="Linux Essentials"
```

Apoi:

```
echo $NUME
```

și:

```
echo "$NUME"
```

Definește apoi:

```
FISIER="/tmp/My File.txt"
```

și execută:

```
printf '<%s>\n' $FISIER
```

apoi:

```
printf '<%s>\n' "$FISIER"
```

Răspunde:

1. Câte argumente primește `printf` în fiecare caz?
 2. În ce etapă apare diferența de comportament?
 3. Ce rol joacă ghilimelele duble?
-

Soluția laboratorului

- Fără ghilimele, Bash poate împărți rezultatul expansiunii în mai multe argumente folosind conținutul variabilei IFS.
 - Cu ghilimele duble, rezultatul este păstrat ca un singur argument.
 - Programul executat primește liste diferite de argumente, deși variabila are aceeași valoare.
-

Ce trebuie să reții

- Word Splitting este aplicat după expansiunile studiate anterior.
- El acționează asupra rezultatului expansiunilor, nu asupra textului introdus de utilizator.
- Variabila IFS stabilește separatorii utilizați în mod implicit.
- Ghilimelele duble împiedică Word Splitting în majoritatea situațiilor.
- În scripturile de administrare, încadrarea variabilelor care conțin căi sau texte între ghilimele duble este una dintre cele mai importante bune practici.

Capitolul 6

6.17 Pathname Expansion (Globbing) – Cum găsește Bash fișiere folosind modele

„După ce Bash termină expansiunile și Word Splitting, urmează o etapă în care poate transforma anumite modele în liste de fișiere existente. Acest mecanism poartă numele de Pathname Expansion și este cunoscut în practică sub denumirea de globbing.”

Ce vei învăța

În acest capitol vei înțelege cum utilizează Bash caractere speciale pentru a identifica fișiere și directoare și vei vedea de ce globbing-ul este diferit de expresiile regulate.

Introducere

Presupunem că într-un director există fișierele:

```
raport1.txt  
raport2.txt  
manual.pdf  
script.sh
```

Executăm:

```
ls *.txt
```

Mulți utilizatori cred că programul `ls` caută fișierele.

Nu este adevărat.

Bash execută mai întâi **Pathname Expansion**.

Comanda transmisă către `ls` devine:

```
ls raport1.txt raport2.txt
```

Programul `ls` nu primește modelul `*.txt`.

Primește lista fișierelor deja identificată de Bash.

Aceasta este una dintre cele mai importante idei ale capitolului.

Ideea fundamentală

Globbering-ul înseamnă:

Bash înlocuiește un model cu lista fișierelor care îl respectă.

Acest proces are loc înainte de executarea programului.

Primul exemplu

Directorul conține:

```
a.txt  
b.txt  
c.pdf
```

Executăm:

```
echo *.txt
```

Bash transformă comanda în:

```
echo a.txt b.txt
```

Programul `echo` nu știe că utilizatorul a scris `*.txt`.

Caracterul *

Cel mai utilizat wildcard este:

*

Acesta înseamnă:

Orice secvență de caractere (inclusiv secvența vidă).

Exemple:

```
ls *.txt
```

```
rm backup*
```

```
cp *.pdf arhiva/
```

Caracterul ?

Wildcard-ul:

?

reprezintă:

Exact un caracter.

Exemple:

```
ls file?.txt
```

va identifica:

```
file1.txt  
fileA.txt  
fileX.txt
```

dar nu:

```
file10.txt
```

deoarece acolo există două caractere după file.

Clase de caractere

Bash permite și utilizarea parantezelor pătrate.

Exemplu:

```
ls raport[123].txt
```

vor fi selectate:

```
raport1.txt  
raport2.txt  
raport3.txt
```

Nu vor fi selectate:

```
raport4.txt
```

sau

```
raport10.txt
```

Intervale

Exemplu:

```
ls imagine[0-9].png
```

sau:

```
ls capitol[A-Z].pdf
```

Aceste modele permit selectarea unor intervale de caractere.

Un lucru foarte important

Globbering-ul caută **fişiere existente**.

Nu caută text.

Nu caută conţinutul fişierelor.

Nu este o expresie regulată.

Aceasta este probabil cea mai frecventă confuzie întâlnită la începători.

Caseta „În spatele cortinei”

Cine execută globbing-ul?

Nu programul.

Bash.

De exemplu:

```
rm *.log
```

Bash identifică toate fişierele `.log`.

Abia apoi execută:

```
rm log1.log log2.log log3.log
```

Dacă programul primeşte deja lista fişierelor, el nu mai ştie că utilizatorul a folosit wildcard-uri.

Globbering şi quoting

Compară:

```
echo *.txt
```

cu:

```
echo "*.txt"
```

În primul caz, Bash încearcă să găsească fişierele care corespund modelului.

În al doilea caz, ghilimelele împiedică Pathname Expansion.

Rezultatul va fi:

```
*.txt
```

Acesta este încă un motiv pentru care quoting-ul trebuie înţeles înainte de globbing.

Greşeală din lumea reală

Un administrator doreşte să transmită modelul:

```
*.log
```

către un script care trebuie să îl proceseze ulterior.

Scrie:

```
script.sh *.log
```

Dar Bash extinde modelul înainte ca scriptul să pornească.

Dacă intenţia era transmiterea modelului ca text, forma corectă este:

```
script.sh "*.log"
```

sau

```
script.sh '*.log'
```

în funcție de context.

Administrator Notes

Înainte de a executa o comandă distructivă precum:

```
rm *.log
```

obișnuiește-te să verifici mai întâi ce fișiere vor fi selectate:

```
printf '%s\n' *.log
```

sau

```
ls *.log
```

Această verificare simplă poate preveni ștergerea accidentală a unor fișiere importante.

Legătura cu LPIC

Globbering-ul este unul dintre subiectele clasice ale examenului LPIC.

Trebuie să cunoști:

- semnificația lui *;
 - semnificația lui ?;
 - utilizarea claselor de caractere;
 - relația dintre globbing și quoting.
-

Cum aş depana această problemă?

Problema

O comandă primește alte argumente decât cele așteptate.

Pașii

1. Verific dacă există wildcard-uri (*, ?, []).
2. Analizez ce fișiere există în directorul curent.
3. Testez mai întâi modelul cu:

```
printf '%s\n' *.txt
```

4. Dacă intenția era transmiterea modelului ca text, folosesc quoting.
-

Laborator 6.17

Obiectiv

Înțelegerea mecanismului de Pathname Expansion.

Cerințe

Creează următoarele fișiere:

```
touch raport1.txt
touch raport2.txt
touch raport3.log
touch manual.pdf
touch script.sh
```

Execută apoi:

```
echo *.txt

echo *.log

echo raport?.txt

echo raport[12].txt
```

Compară cu:

```
echo "*.txt"
```

și:

```
echo '*.txt'
```

Răspunde:

1. În ce caz Bash caută fișiere?
 2. În ce caz transmite textul literal?
 3. Ce argumente primește programul `echo` în fiecare situație?
-

Soluția laboratorului

- Fără quoting, Bash încearcă să extindă modelele în funcție de fișierele existente.
- Cu ghilimele, modelul este transmis ca text.

- Programul `echo` primește fie lista fișierelor, fie textul literal `*.txt`, în funcție de existența quoting-ului.
-

Ce trebuie să reții

- Pathname Expansion este realizată de Bash înainte de executarea programului.
- `*` reprezintă orice secvență de caractere.
- `?` reprezintă exact un caracter.
- `[]` permit definirea claselor și intervalelor de caractere.
- Globbing-ul operează asupra **numelor de fișiere**, nu asupra conținutului acestora.
- Quoting-ul împiedică Pathname Expansion.

Nu confunda!

Globbing Expresii regulate (Regex)

Caută nume de fișiere Caută modele în text

Este interpretat de Bash Este interpretat de programe precum `grep`, `sed`, `awk`

`*` înseamnă orice secvență de caractere `*` repetă elementul precedent

`?` înseamnă exact un caracter `?` are altă semnificație în regex moderne și poate lipsi în unele implementări POSIX

Capitolul 6

6.18 Quote Removal – Ultima etapă înainte de execuția comenzii

„Ghilimelele și caracterul escape controlează modul în care Bash interpretează linia de comandă. După ce și-au îndeplinit rolul, ele nu mai sunt necesare și sunt eliminate. Acest proces poartă numele de Quote Removal.”

Ce vei învăța

În acest capitol vei înțelege de ce ghilimelele nu ajung niciodată la programul executat și care este rolul ultimei etape din procesarea unei comenzi.

Introducere

Până acum am văzut că ghilimelele:

- împiedică Word Splitting;
- împiedică Pathname Expansion;
- controlează Parameter Expansion;
- controlează interpretarea caracterelor speciale.

Acum apare o întrebare foarte interesantă.

Dacă ghilimelele sunt atât de importante, de ce nu apar în argumentele primite de program?

Răspunsul este simplu.

Pentru că ele sunt eliminate înainte de execuția comenzii.

Primul exemplu

Executăm:

```
echo "Linux Essentials"
```

Mulți utilizatori cred că `echo` primește argumentul:

```
"Linux Essentials"
```

Nu.

Programul primește:

```
Linux Essentials
```

Ghilimelele au fost folosite de Bash pentru interpretare.

După aceea au fost eliminate.

Un model mental

Să urmărim întregul proces.

Utilizatorul tastează:

```
echo "$HOME"
```

Bash execută:

1. identifică ghilimelele;

2. realizează Parameter Expansion;
3. împiedică Word Splitting;
4. împiedică Globbing;
5. elimină ghilimelele;
6. lansează programul.

Programul `echo` primește doar:

```
/home/daniel
```

Nu vede niciodată caracterul:

```
"
```

Un exemplu cu caracterul escape

Executăm:

```
echo $HOME
```

Bash observă:

```
\
```

Înțelege că trebuie protejat caracterul:

```
$
```

După ce această regulă a fost aplicată, caracterul:

```
\
```

nu mai are niciun rol.

Este eliminat.

Programul primește:

```
$HOME
```

Quote Removal nu modifică textul

Este important să înțelegem ce face această etapă.

Quote Removal **nu schimbă valoarea argumentelor**.

Ea elimină doar caracterele de quoting care au fost utilizate pentru controlul interpretării.

Acesta este un detaliu foarte important.

Exemplu complet

Executăm:

```
NUME="Linux Essentials"
```

```
echo "$NUME"
```

Procesarea are loc astfel:

Linia introdusă

↓

Parameter Expansion

↓

Linux Essentials

↓

Word Splitting

(este blocat)

↓

Pathname Expansion

(este blocată)

↓

Quote Removal

↓

Linux Essentials

↓

execve()

↓

echo

Observă că ghilimelele dispar chiar înainte de lansarea programului.

Caseta „În spatele cortinei”

De ce este necesară Quote Removal?

Imaginează-ți că ghilimelele nu ar fi eliminate.

Comanda:

```
echo "Salut"
```

ar transmite către program argumentul:

```
"Salut"
```

Practic, toate programele ar trebui să știe ce înseamnă quoting-ul Bash.

Acest lucru ar fi imposibil.

De aceea quoting-ul este responsabilitatea exclusivă a shell-ului.

După interpretare, aceste caractere sunt eliminate.

Programul executat primește doar argumentele finale.

Greșeală din lumea reală

Un cursant spune:

„Comanda mea conține ghilimele. De ce programul nu le vede?”

Pentru că ghilimelele nu reprezintă date.

Ele reprezintă **instrucțiuni pentru Bash**.

După ce Bash le folosește, ele dispar.

Acest lucru explică foarte multe situații aparent ciudate din scripting.

Administrator Notes

Ori de câte ori analizezi o comandă, încearcă să faci diferența între:

- **ce tastează utilizatorul;**
- **ce vede Bash;**
- **ce primește programul executat.**

Aceste trei momente nu sunt întotdeauna identice.

Înțelegerea acestei diferențe simplifică foarte mult depanarea scripturilor.

Legătura cu LPIC

Quote Removal nu este întotdeauna menționată explicit în întrebările LPIC, însă explică multe dintre diferențele de comportament dintre comenzi aparent identice.

Înțelegerea acestei etape completează imaginea procesării unei linii de comandă în Bash.

Cum aş depana această problemă?

Problema

Programul pare să ignore ghilimelele introduse de utilizator.

Pași

1. Verific dacă ghilimelele sunt folosite doar pentru quoting.
 2. Îmi amintesc că ele sunt eliminate înainte de execuția programului.
 3. Analizez argumentele finale pe care le primește comanda.
 4. Dacă am nevoie ca un caracter de ghilime să facă parte din textul transmis, îl introduc explicit în șir, nu mă bazez pe caracterele de quoting.
-

Laborator 6.18

Obiectiv

Observarea efectului Quote Removal.

Cerințe

Execută:

```
echo "Linux"
```

Apoi:

```
echo 'Linux'
```

și:

```
echo \${HOME}
```

În final:

```
VAR="Linux Essentials"
```

```
printf '<%s>\n' "$VAR"
```

Răspunde:

1. Primește programul ghilimelele?
 2. Primește programul caracterul \?
 3. În ce moment sunt eliminate aceste caractere?
-

Soluția laboratorului

- Ghilimelele și caracterul \ sunt folosite de Bash pentru controlul interpretării.
 - După ce și-au îndeplinit rolul, ele sunt eliminate.
 - Programul executat primește doar argumentele finale, fără caracterele de quoting.
-

Ce trebuie să reții

- Quote Removal este ultima etapă înainte de execuția programului.
- Ghilimelele și caracterul \ nu sunt transmise programului.
- Quoting-ul este interpretat exclusiv de Bash.
- Programele executate primesc doar argumentele rezultate după toate transformările.

Capitolul 6.19

Sinteză – Viața unei linii de comandă în Bash

„Un administrator Linux nu trebuie doar să știe ce comandă să execute, ci și ce face Bash cu acea comandă înainte ca programul să înceapă să ruleze.”

Introducere

În ultimele capitole am studiat numeroase mecanisme interne ale Bash.

La început ele pot părea independente.

În realitate, toate fac parte din același proces.

Scopul acestui capitol este să privim imaginea de ansamblu.

Din momentul în care apăsăm Enter...

Să presupunem că utilizatorul execută:

```
echo "$HOME/$(date +%Y)/$((2+3))" *.txt
```

Această comandă **nu este executată imediat**.

Mai întâi Bash o procesează.

Etapă 1 – Analiza sintactică

Bash citește întreaga linie.

Identifică:

- comenzile;
- argumentele;
- operatorii;
- caracterele speciale;
- quoting-ul.

În această etapă încă nu există execuție.

Etapă 2 – Parameter Expansion

Bash înlocuiește:

`$HOME`

cu:

`/home/daniel`

Etapă 3 – Command Substitution

Execută:

`$(date +%Y)`

și obține:

2026

Etapa 4 – Arithmetic Expansion

Calculează:

```
$( (2+3) )
```

Rezultatul devine:

5

Etapa 5 – Word Splitting

Dacă este permis (adică nu există quoting care să îl împiedice), Bash împarte rezultatele expansiunilor în argumente folosind separatorii definiți de variabila `IFS`.

Etapa 6 – Pathname Expansion

Dacă există modele precum:

```
*.txt
```

Bash caută fișierele existente.

Exemplu:

```
manual.txt
```

```
script.txt
```

```
notes.txt
```

Comanda devine:

```
echo "/home/daniel/2026/5" manual.txt script.txt notes.txt
```

Etapa 7 – Quote Removal

Acum ghilimelele și caracterele escape și-au îndeplinit rolul.

Ele sunt eliminate.

Programul executat nu le va vedea niciodată.

Etapa 8 – execve()

Abia acum Bash solicită nucleului Linux lansarea programului.

Acesta este momentul în care începe execuția efectivă.

Până aici am discutat exclusiv despre activitatea shell-ului.

Imaginea completă

Utilizator

↓

Tastează comanda

↓

Analiza sintactică

↓

Parameter Expansion

↓

Command Substitution

↓

Arithmetic Expansion

↓

Word Splitting

↓

Pathname Expansion

↓

Quote Removal

↓

execve()

↓

Kernel Linux

↓

Programul executat

Aceasta este una dintre cele mai importante diagrame din întregul manual.

Un exercițiu mental

Analizează următoarea comandă:

```
FISIER="Manual Linux.pdf"
```

```
echo "$HOME/$FISIER/$(date +%Y)/$((10+5))"
```

Nu încerca să afli doar rezultatul final.

Întreabă-te:

1. Ce etapă rulează prima?
2. Ce modificare produce?
3. Care este rezultatul după fiecare etapă?
4. Ce argument primește efectiv programul `echo`?

Dacă poți răspunde la aceste întrebări fără să execuți comanda, înseamnă că ai înțeles modul în care Bash procesează o linie de comandă.

Cele mai frecvente erori

1. „Programul execută variabilele.”

Nu.

Bash expandează variabilele.

2. „echo execută \$(date).”

Nu.

Bash execută `date`.

3. „rm interpretează *.txt.”

Nu.

Bash realizează globbing-ul.

4. „Programul vede ghilimelele.”

Nu.

Quote Removal le elimină înainte de execuție.

5. „Spațiile sunt interpretate de program.”

Nu.

Word Splitting este realizat de Bash.

Administrator Notes

Când întâlnești un comportament neașteptat în Bash, evită să privești comanda ca pe un singur bloc.

Împarte analiza în etape.

Întreabă-te:

- Ce a schimbat Bash?
- Ce argumente au rezultat?
- Ce primește efectiv programul?

Această metodă de lucru este folosită de administratorii experimentați și transformă depanarea dintr-un proces bazat pe încercări într-un proces logic și predictibil.

Laborator de sinteză

Obiectiv

Analizarea completă a unei linii de comandă.

Execută:

```
NUME="Linux Essentials"
```

```
echo "$HOME/$NUME/$(date +%Y)/$( (25+5) )" *.txt
```

Completează următorul tabel.

Etapă	Rezultat
Linia introdusă	
Parameter Expansion	
Command Substitution	

Etapă	Rezultat
Arithmetic Expansion	
Word Splitting	
Pathname Expansion	
Quote Removal	
Argumentele finale ale lui <code>echo</code>	
Notă pentru cursant: Nu te grăbi să completezi tabelul. Scopul exercițiului nu este viteza, ci înțelegerea mecanismului prin care Bash transformă o linie de comandă.	

Soluția laboratorului

Cursantul trebuie să identifice, pentru fiecare etapă, transformările produse de Bash și să observe că programul `echo` primește doar argumentele rezultate după toate expansiunile și după eliminarea caracterelor de quoting.

Ce trebuie să reții

- Bash procesează fiecare linie de comandă într-o succesiune bine definită de etape.
- Fiecare etapă transformă rezultatul celei precedente.
- Programele executate nu văd linia de comandă originală, ci doar forma finală construită de Bash.
- Înțelegerea acestui flux explică majoritatea comportamentelor aparent surprinzătoare ale shell-ului.

Capitolul 7

Bash Redirections – Cum controlează Bash fluxul datelor

„Majoritatea comenzilor Linux nu comunică doar prin argumente. Ele comunică și prin fluxuri de date. Redirecționările permit controlul acestor fluxuri și reprezintă una dintre cele mai puternice caracteristici ale shell-ului.”

Ce vei învăța

În acest capitol vei înțelege ce sunt fluxurile standard (Standard Streams), de ce există și cum poate Bash să le redirecționeze către fișiere, dispozitive sau alte programe.

Introducere

Până acum am studiat modul în care Bash pregătește și lansează o comandă.

Acum apare o întrebare la fel de importantă.

Odată ce programul rulează, cum comunică acesta cu exteriorul?

De unde citește informații?

Unde afișează rezultatele?

Unde trimite mesajele de eroare?

Răspunsul îl reprezintă **fluxurile standard** (*Standard Streams*).

Totul este un flux de date

Majoritatea programelor Linux nu „știu” dacă citesc de la tastatură.

Nu „știu” dacă afișează pe ecran.

Ele citesc și scriu folosind fluxuri de date.

Acest lucru oferă sistemului Linux o flexibilitate extraordinară.

Cele trei fluxuri standard

Orice proces pornește, în mod normal, cu trei fluxuri deja deschise.

Descriptor	Nume	Direcție	Valoare numerică
Standard Input	stdin	Intrare	0
Standard Output	stdout	Ieșire normală	1
Standard Error	stderr	Mesaje de eroare	2

Aceste trei fluxuri sunt create înainte ca programul să înceapă execuția.

Primul exemplu

Executăm:

```
ls
```

Rezultatul apare pe ecran.

De fapt:

```
stdout → terminal
```

Executăm:

```
cat
```

Programul așteaptă să introducem text.

De fapt:

```
stdin ← tastatură
```

Executăm:

```
ls fisier_inexistent
```

Mesajul:

```
No such file or directory
```

nu este trimis pe stdout.

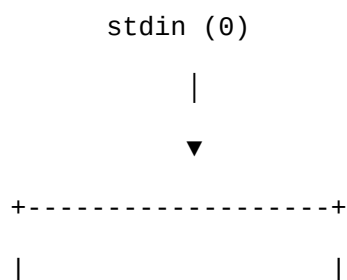
Este trimis pe:

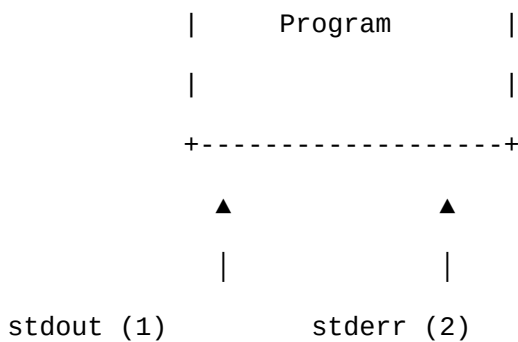
```
stderr
```

Această diferență va deveni foarte importantă în capitolele următoare.

Un model mental

Imaginează-ți fiecare proces ca pe o cutie.





Programul nu comunică direct cu tastatura sau monitorul.

Comunică prin aceste trei fluxuri.

De ce există două ieșiri?

Mulți cursanți întreabă:

De ce nu există un singur flux de ieșire?

Răspunsul este elegant.

Pentru că rezultatele și erorile au scopuri diferite.

Exemplu:

```
find / -name passwd
```

Produce:

- lista fișierelor găsite;
- mesaje de tip „Permission denied”.

Dacă toate ar fi trimise prin același flux, ar fi dificil să folosim rezultatul în alte comenzi.

Separarea dintre `stdout` și `stderr` permite redirecționarea lor independentă.

Caseta „În spatele cortinei”

File Descriptors

În Linux, fiecare flux este identificat printr-un **descriptor de fișier** (*File Descriptor*).

La pornirea procesului:

```
0 → stdin
```

1 → stdout

2 → stderr

Mai târziu vom vedea că un program poate deschide și alți descriptori:

3

4

5

...

Acesta este motivul pentru care în Bash putem scrie:

```
2> erori.log
```

Numărul 2 nu este ales întâmplător.

El reprezintă descriptorul asociat fluxului `stderr`.

Greșeală din lumea reală

Un administrator execută:

```
find / -name passwd > rezultate.txt
```

și se miră că pe ecran apar în continuare mesaje de eroare.

Explicația este simplă.

Operatorul:

```
>
```

redirecționează doar:

```
stdout
```

Mesajele de eroare continuă să fie trimise către:

```
stderr
```

Vom vedea imediat cum putem controla și acest flux.

Administrator Notes

Înainte să înveți operatorii `>`, `<`, `>>` sau `2>`, înțelege foarte bine cele trei fluxuri standard.

Odată ce această idee este clară, toate redirecționările din Bash devin logice.

Legătura cu LPIC

Fluxurile standard și descriptorii de fișiere sunt un subiect clasic în LPIC-1.

Examenul presupune că înțelegeți:

- diferența dintre `stdin`, `stdout` și `stderr`;
 - rolul descriptorilor 0, 1 și 2;
 - faptul că fiecare flux poate fi redirecționat independent.
-

Cum aș depana această problemă?

Problema

O comandă produce atât rezultate utile, cât și mesaje de eroare.

Pașii

1. Identific dacă informațiile provin din `stdout` sau `stderr`.
 2. Redirecționez separat cele două fluxuri.
 3. Verific dacă programul funcționează corect chiar dacă afișează avertismente.
 4. Aleg destinația potrivită pentru fiecare flux (terminal, fișier sau alt program).
-

Laborator 7.1

Obiectiv

Observarea celor trei fluxuri standard.

Cerințe

Execută:

```
cat
```

Introdu câteva linii și încheie introducerea cu **Ctrl+D**.

Observă că programul citește din `stdin`.

Execută apoi:

```
ls
```

Observă că rezultatul este afișat pe `stdout`.

În final execută:

```
ls fisier_inexistent
```

Observă că mesajul de eroare este trimis pe `stderr`.

Întrebări

1. Care flux este utilizat de `cat` pentru citirea datelor?
 2. Care flux este utilizat de `ls` pentru afișarea listei fișierelor?
 3. De ce mesajul de eroare este separat de rezultatul normal?
-

Soluția laboratorului

- `cat` citește din **`stdin`** (descriptorul 0).
 - `ls` afișează lista fișierelor pe **`stdout`** (descriptorul 1).
 - Mesajele de eroare sunt trimise pe **`stderr`** (descriptorul 2), pentru a putea fi tratate separat de rezultatul normal.
-

Ce trebuie să reții

- Orice proces Linux pornește cu trei fluxuri standard.
- `stdin` (0) este utilizat pentru intrare.
- `stdout` (1) este utilizat pentru rezultatele normale.
- `stderr` (2) este utilizat pentru mesaje de eroare.
- Înțelegerea acestor trei fluxuri este fundamentul tuturor redirecționărilor Bash.

Capitolul 7

7.2 Redirecționarea Standard Output (`>` și `>>`)

„Dacă `stdout` este doar un flux de date, atunci acesta nu trebuie să ajungă întotdeauna pe ecran. Bash ne permite să schimbăm destinația acestui flux fără ca programul executat să știe acest lucru.”

Ce vei învăța

În acest capitol vei înțelege cum poate Bash redirecționa ieșirea standard către un fișier și care este diferența dintre operatorii `>` și `>>`.

Introducere

În capitolul anterior am aflat că orice proces are un flux numit **Standard Output (stdout)**.

În mod implicit acesta este conectat la terminal.

De aceea executăm:

```
ls
```

și rezultatul apare pe ecran.

Dar Bash poate schimba destinația acestui flux.

Programul continuă să scrie pe `stdout`.

Diferența este că `stdout` nu mai este conectat la terminal.

Este conectat la un fișier.

Programul nu observă această schimbare.

Primul exemplu

Executăm:

```
ls > fisiere.txt
```

La prima vedere pare că `ls` scrie în fișier.

Nu este chiar așa.

Ce se întâmplă în realitate?

1. Bash creează (sau deschide) fișierul `fisiere.txt`.
2. Bash conectează descriptorul **1 (stdout)** la acel fișier.
3. Abia apoi lansează programul `ls`.

Programul `ls` continuă să scrie pe `stdout`.

Doar că `stdout` nu mai este terminalul.

Modelul mental

În mod normal:

```
stdout
```

```
ls -----> Terminal
```

Cu redirecționare:

```
stdout
```

```
ls -----> fisiere.txt
```

Programul este identic.

Destinația fluxului este diferită.

Operatorul >

Operatorul:

>

înseamnă:

scrie în fișier, iar dacă există deja, suprascrie conținutul.

Exemplu:

```
date > data.txt
```

Conținutul anterior al fișierului este înlocuit.

Demonstrație

Executăm:

```
echo "Prima linie" > test.txt
```

Apoi:

```
echo "A doua linie" > test.txt
```

Dacă afișăm fișierul:

```
cat test.txt
```

rezultatul este:

```
A doua linie
```

Prima linie a fost pierdută.

Operatorul >>

Uneori nu dorim suprascrierea.

Dorim adăugarea informației la sfârșitul fișierului.

Pentru aceasta folosim:

>>

Exemplu:

```
echo "Prima linie" >> jurnal.log
```

Apoi:

```
echo "A doua linie" >> jurnal.log
```

Rezultatul:

```
Prima linie  
A doua linie
```

Fișierul este extins.

Când folosim fiecare operator?

Folosește:

>

când dorești un fișier nou sau când conținutul anterior nu mai este necesar.

Folosește:

>>

când dorești să păstrezi informațiile existente și să adaugi altele noi.

Caseta „În spatele cortinei”

Cine deschide fișierul?

Mulți utilizatori cred că programul executat creează fișierul.

În realitate, Bash face acest lucru.

Mai întâi deschide fișierul.

Apoi modifică descriptorul 1 (stdout).

Abia după aceea programul începe execuția.

Aceasta explică de ce aproape orice comandă poate utiliza operatorii de redirecționare, chiar dacă programul nu știe nimic despre ei.

Greșeală din lumea reală

Un administrator execută:

```
cat /var/log/syslog > arhiva.log
```

A doua zi execută din nou aceeași comandă.

Se miră că informațiile din ziua precedentă au dispărut.

Motivul este simplu.

Operatorul:

```
>
```

suprascrie fișierul.

Dacă intenția era păstrarea istoricului, trebuia utilizat:

```
cat /var/log/syslog >> arhiva.log
```

Administrator Notes

Înainte să apeși Enter, uită-te încă o dată la operator.

O singură diferență de caracter:

```
>
```

sau

```
>>
```

poate însemna:

- păstrarea unui jurnal de câțiva ani;

sau

- pierderea definitivă a acestuia.

Administratorii experimentați dezvoltă reflexul de a verifica întotdeauna operatorul de redirecționare înainte de executarea comenzilor importante.

Legătura cu LPIC

LPIC verifică frecvent diferența dintre:

>

și

>>

Nu este suficient să memorezi că unul „scrie” și celălalt „adaugă”.

Trebuie să înțelegi că ambele redirectionează **stdout**, diferența fiind modul în care este deschis fișierul.

Cum aş depana această problemă?

Problema

Fișierul rezultat nu conține informațiile așteptate.

Pașii

1. Verific dacă am folosit > sau >>.
 2. Verific dacă fișierul exista înainte de execuția comenzii.
 3. Confirm că programul scrie pe **stdout**, nu pe **stderr**.
 4. Verific drepturile de scriere asupra fișierului și directorului.
-

Laborator 7.2

Obiectiv

Înțelegerea diferenței dintre > și >>.

Cerințe

Execută:

```
echo "Linia 1" > test.txt
```

Apoi:

```
echo "Linia 2" > test.txt
```

Afișează conținutul:

```
cat test.txt
```

Observă rezultatul.

Șterge fișierul și repetă exercițiul folosind:

```
echo "Linia 1" >> test.txt
```

și:

```
echo "Linia 2" >> test.txt
```

Compară rezultatele.

Soluția laboratorului

- Operatorul `>` creează fișierul dacă nu există și îi înlocuiește conținutul dacă există deja.
 - Operatorul `>>` creează fișierul dacă nu există și adaugă noile informații la sfârșitul lui dacă există.
 - În ambele cazuri este redirecționat fluxul `stdout` (descriptorul 1).
-

Ce trebuie să reții

- `stdout` este descriptorul de fișier 1.
- `>` redirecționează `stdout` și suprascrie fișierul.
- `>>` redirecționează `stdout` și adaugă informațiile la sfârșitul fișierului.
- Programul executat continuă să scrie pe `stdout`; Bash este cel care schimbă destinația fluxului.
- Înțelegerea mecanismului este mai importantă decât memorarea operatorilor.

Capitolul 7

7.3 Redirecționarea Standard Input (<)

„După cum putem schimba destinația ieșirii unui program, putem schimba și sursa din care acesta citește date. Redirecționarea intrării permite unui program să citească dintr-un fișier în locul tastaturii, fără ca programul să știe acest lucru.”

Ce vei învăța

În acest capitol vei înțelege cum funcționează redirectionarea fluxului **Standard Input (stdin)** și în ce situații este preferabilă folosirea operatorului `<`.

Introducere

În capitolul anterior am văzut că orice proces are trei fluxuri standard.

Am redirectionat:

```
stdout (1)
```

Acum vom lucra cu:

```
stdin (0)
```

În mod normal, `stdin` este conectat la tastatură.

De aceea comenzi precum:

```
cat
```

așteaptă să introducem text.

Dar Bash poate modifica această conexiune.

Programul continuă să citească din `stdin`, însă `stdin` nu mai este tastatura.

Este un fișier.

Primul exemplu

Presupunem că avem fișierul:

```
nume.txt
```

cu următorul conținut:

```
Ana  
Maria  
Ion
```

Executăm:

```
cat < nume.txt
```

Rezultatul este:

```
Ana
```

Maria
Ion

La prima vedere pare că `cat` citește fișierul.

În realitate, `cat` citește din **`stdin`**.

Doar că Bash a conectat `stdin` la fișierul `nume.txt`.

Modelul mental

Fără redirectionare:

Tastatură

|



`stdin (0)`

|



Program

Cu redirectionare:

`nume.txt`

|



`stdin (0)`

|



Program

Programul nu observă diferența.

El citește întotdeauna din descriptorul 0.

Cine deschide fișierul?

Exact ca la operatorul `>`.

Mai întâi Bash:

- deschide fișierul;
- conectează descriptorul 0;
- lansează programul.

Programul nu deschide fișierul.

Primește doar un `std` în deja pregătit.

Un exemplu practic

Executăm:

```
wc -l < nume.txt
```

Rezultatul:

3

Observă că `wc` citește datele din `std`.

Nu știe dacă acestea provin:

- de la tastatură;
- dintr-un fișier;
- dintr-un pipe.

Această flexibilitate este una dintre caracteristicile fundamentale ale filozofiei Unix.

< versus argument

Compară:

```
wc -l nume.txt
```

cu

```
wc -l < nume.txt
```

Rezultatul este asemănător, dar mecanismul este diferit.

În primul caz:

programul primește argumentul:

```
nume.txt
```

și îl deschide singur.

În al doilea caz:

Bash deschide fișierul și îl conectează la `stdin`.

Programul citește pur și simplu din fluxul de intrare.

Această diferență este foarte importantă.

Caseta „În spatele cortinei”

De ce există două modalități?

Pentru că nu toate programele acceptă nume de fișiere ca argumente.

Unele programe citesc exclusiv din `stdin`.

Redirecționarea permite alimentarea acestor programe cu date fără modificarea lor.

Acesta este unul dintre principiile fundamentale ale ecosistemului Unix:

Programele mici comunică prin fluxuri standard.

Greșeală din lumea reală

Un administrator execută:

```
sort
```

și așteaptă ca programul să proceseze fișierul:

```
date.txt
```

Programul pare blocat.

În realitate, `sort` așteaptă date pe `stdin`.

Soluțiile sunt:

```
sort < date.txt
```

sau

```
sort date.txt
```

în funcție de modul în care programul este proiectat să primească date.

Administrator Notes

Înainte de a folosi `<`, verifică documentația programului.

Unele utilitare acceptă atât fișiere ca argumente, cât și date prin `stdin`.

Altele funcționează exclusiv cu unul dintre aceste mecanisme.

Înțelegerea diferenței te ajută să alegi soluția cea mai potrivită.

Legătura cu LPIC

LPIC testează frecvent diferența dintre:

```
wc -l fisier.txt
```

și

```
wc -l < fisier.txt
```

Nu întotdeauna rezultatul vizibil diferă, dar mecanismul intern este diferit.

Examenul urmărește înțelegerea acestui mecanism.

Cum aş depana această problemă?

Problema

Programul pare că „așteaptă” fără să afișeze nimic.

Pașii

1. Verific dacă programul citește din `stdin`.
 2. Verific dacă trebuia utilizată redirectionarea `<`.
 3. Confirm că fișierul există și poate fi citit.
 4. Dacă folosesc un pipe, verific dacă procesul anterior produce date.
-

Laborator 7.3

Obiectiv

Înțelegerea redirectionării intrării standard.

Cerințe

Creează fișierul:

```
cat > persoane.txt
```

Introdu:

```
Ana  
Maria  
Ion
```

Încheie introducerea cu:

```
Ctrl+D
```

Execută apoi:

```
cat < persoane.txt
```

și:

```
wc -l < persoane.txt
```

Compară rezultatul cu:

```
wc -l persoane.txt
```

Răspunde:

1. Cine deschide fișierul în fiecare situație?
 2. Ce primește efectiv programul?
 3. Care este rolul operatorului <?
-

Soluția laboratorului

- Operatorul < redirecționează descriptorul `stdin` (0) către un fișier.
 - Bash deschide fișierul înainte de lansarea programului.
 - Programul citește din `stdin`, fără să știe dacă datele provin dintr-un fișier sau de la tastatură.
 - Utilizarea unui argument și utilizarea redirecționării pot produce același rezultat vizibil, dar implică mecanisme interne diferite.
-

Ce trebuie să reții

- `stdin` este descriptorul de fișier **0**.
- Operatorul < redirecționează intrarea standard către un fișier.

- Bash este cel care deschide fișierul și conectează fluxul de intrare.
- Programul executat citește întotdeauna din `stdin`, indiferent de sursa datelor.
- Înțelegerea diferenței dintre argumente și redirecționarea intrării este importantă atât pentru scripting, cât și pentru examenul LPIC.

Capitolul 7

7.4 Redirecționarea Standard Error (2> și 2>>)

„Un program poate produce atât rezultate normale, cât și mesaje de eroare. Bash permite tratarea acestor două fluxuri în mod independent, oferind un control foarte precis asupra modului în care sunt gestionate informațiile.”

Ce vei învăța

În acest capitol vei înțelege cum poate fi redirecționat fluxul **Standard Error (stderr)** și care este diferența dintre operatorii `2>` și `2>>`.

Introducere

În capitolele anterioare am învățat că orice proces pornește cu trei fluxuri standard:

Descriptor	Flux

0	stdin
1	stdout
2	stderr

Am învățat deja cum redirecționăm:

- `stdout` folosind `>` și `>>`;
- `stdin` folosind `<`.

Acum vom vedea cum redirecționăm:

`stderr`

De ce există stderr?

Să analizăm comanda:

```
ls
```

Rezultatul este lista fișierelor.

Acesta este un rezultat normal.

Dar dacă executăm:

```
ls fisier_inexistent
```

obținem:

```
ls: cannot access 'fisier_inexistent': No such file or directory
```

Acesta nu este rezultatul cerut.

Este un mesaj de eroare.

Programul îl trimite pe:

```
stderr
```

nu pe:

```
stdout
```

Primul exemplu

Executăm:

```
ls fisier_inexistent 2> erori.txt
```

Observă operatorul:

```
2>
```

Numărul:

```
2
```

reprezintă descriptorul asociat fluxului:

```
stderr
```

Rezultatul?

Pe ecran nu mai apare mesajul de eroare.

În schimb, fișierul:

erori.txt

conține:

```
ls: cannot access 'fisier_inexistent': No such file or directory
```

Modelul mental

Fără redirectionare:

stderr

|



Terminal

Cu redirectionare:

stderr

|



erori.txt

Programul continuă să scrie pe descriptorul 2.

Bash schimbă doar destinația.

Operatorul 2>

Operatorul:

2>

înseamnă:

Redirecționează fluxul stderr și suprascrie fișierul dacă acesta există.

Exemplu:

```
grep Linux fisier_inexistent 2> jurnal_erori.log
```

Rezultatele normale (dacă există) continuă să fie trimise pe stdout.

Numai erorile ajung în fișier.

Operatorul 2>>

La fel ca în cazul lui >>, există și varianta pentru adăugare.

Exemplu:

```
grep Linux fisier_inexistent 2>> jurnal_erori.log
```

Acum mesajele noi sunt adăugate la sfârșitul fișierului.

Istoricul este păstrat.

Un exemplu foarte important

Executăm:

```
find / -name passwd > rezultate.txt
```

Mulți cursanți sunt surprinși.

Fișierul:

```
rezultate.txt
```

conține rezultatele căutării.

Dar pe ecran continuă să apară:

```
Permission denied
```

De ce?

Pentru că:

```
>
```

redirecționează doar:

```
stdout
```

Mesajele de eroare continuă să fie trimise către:

```
stderr
```

Separarea fluxurilor

Putem scrie:

```
find / -name passwd > rezultate.txt 2> erori.txt
```

Acum:

`stdout`

merge în:

`rezultate.txt`

iar:

`stderr`

merge în:

`erori.txt`

Aceasta este una dintre cele mai utile tehnici în administrarea sistemelor Linux.

Caseta „În spatele cortinei”

De ce este utilă separarea?

Imaginează-ți că ai un script care caută fișiere de configurare pe întregul sistem.

Rezultatele sunt importante.

Mesajele:

`Permission denied`

nu sunt neapărat erori critice.

Ele indică doar că anumite directoare nu pot fi citite.

Separarea celor două fluxuri permite:

- analizarea rezultatelor;
- investigarea separată a erorilor;
- automatizarea procesării.

Aceasta este una dintre ideile fundamentale ale filozofiei Unix.

Greșeală din lumea reală

Un administrator execută:

```
grep root /etc/* > raport.txt
```

și observă că raportul este incomplet.

În realitate, mesajele:

Is a directory

au rămas pe terminal.

Mai grav, el nu le observă și presupune că toate fișierele au fost procesate.

O soluție mai bună este:

```
grep root /etc/* > raport.txt 2> erori.log
```

Astfel, atât rezultatele, cât și problemele pot fi analizate ulterior.

Administrator Notes

În scripturile de administrare, obișnuiește-te să tratezi separat:

- rezultatele;
- avertismentele;
- erorile.

Această disciplină face depanarea mult mai simplă și permite construirea unor jurnale clare și ușor de analizat.

Legătura cu LPIC

LPIC verifică frecvent:

- semnificația descriptorului 2;
 - diferența dintre > și 2>;
 - diferența dintre 2> și 2>>;
 - capacitatea de a separa stdout și stderr.
-

Cum aş depana această problemă?

Problema

Comanda produce atât rezultate utile, cât și mesaje de eroare.

Pașii

1. Verific dacă rezultatele sunt trimise pe stdout.
2. Verific dacă mesajele provin din stderr.

3. Redirecționez cele două fluxuri în fișiere diferite.
 4. Analizez separat conținutul fiecărui fișier.
-

Laborator 7.4

Obiectiv

Separarea rezultatelor normale de mesajele de eroare.

Cerințe

Execută:

```
ls /etc /director_inexistent
```

Observă rezultatul.

Apoi execută:

```
ls /etc /director_inexistent > rezultate.txt
```

Observă că mesajul de eroare apare în continuare pe ecran.

Execută apoi:

```
ls /etc /director_inexistent 2> erori.txt
```

Observă că lista fișierelor apare pe terminal, iar mesajele de eroare sunt salvate în fișier.

În final execută:

```
ls /etc /director_inexistent > rezultate.txt 2> erori.txt
```

Analizează separat cele două fișiere.

Soluția laboratorului

- > redirecționează doar stdout.
 - 2> redirecționează doar stderr.
 - Cele două fluxuri sunt independente și pot avea destinații diferite.
 - Bash modifică descriptorii înainte ca programul să înceapă execuția.
-

Ce trebuie să reții

- `stderr` este descriptorul 2.
- `2>` redirecționează mesajele de eroare și suprascrie fișierul.
- `2>>` adaugă mesajele de eroare la sfârșitul fișierului.
- `stdout` și `stderr` sunt fluxuri independente.
- Separarea lor este una dintre cele mai importante tehnici în administrarea sistemelor Linux și în scrierea scripturilor robuste.

Capitolul 7

7.5 Redirecționarea simultană a `stdout` și `stderr` (`2>&1` și `&>`)

„În multe situații dorim să păstrăm împreună rezultatele normale și mesajele de eroare. Bash permite acest lucru prin duplicarea descriptorilor de fișiere și prin redirecționări combinate.”

Ce vei învăța

În acest capitol vei înțelege cum poate Bash face ca `stdout` și `stderr` să ajungă în aceeași destinație și de ce ordinea redirecționărilor este esențială.

Introducere

Până acum am văzut că putem redirecționa separat:

`stdout` (1)

și

`stderr` (2)

Dar apare o situație foarte frecventă.

Executăm un script și dorim un singur jurnal care să conțină:

- rezultatele normale;
- mesajele de eroare;
- avertismentele.

Cum procedăm?

Primul exemplu

Executăm:

```
find / -name passwd > raport.log 2>&1
```

La final, atât rezultatele normale, cât și mesajele de eroare sunt salvate în:

```
raport.log
```

Ce înseamnă 2>&1?

Aceasta este una dintre cele mai neînțelese expresii din Bash.

Să o analizăm.

Numărul:

2

înseamnă:

`stderr`

Simbolul:

`>&`

înseamnă:

duplică descriptorul.

Numărul:

1

înseamnă:

`stdout`

Prin urmare:

`2>&1`

înseamnă:

Fă ca descriptorul 2 să indice aceeași destinație către care indică descriptorul 1 în acest moment.

Această formulare este mai exactă decât simplificarea „trimite stderr în stdout”.

Modelul mental

La început:

stdout —————> Terminal

stderr —————> Terminal

După:

```
> raport.log
```

situația devine:

stdout —————> raport.log

stderr —————> Terminal

După:

```
2>&1
```

devine:

stdout —————> raport.log

stderr —————> raport.log

Observă că `stderr` nu este conectat la `stdout`.

Ambele indică aceeași destinație.

Aceasta este diferența conceptuală importantă.

De ce ordinea contează?

Compară:

```
find / -name passwd > raport.log 2>&1
```

cu:

```
find / -name passwd 2>&1 > raport.log
```

Mulți utilizatori cred că sunt identice.

Nu sunt.

Primul caz

```
> raport.log
```

mută `stdout` în fișier.

Apoi:

```
2>&1
```

face ca `stderr` să urmeze aceeași destinație.

Rezultatul:

```
stdout → raport.log
```

```
stderr → raport.log
```

Al doilea caz

Mai întâi:

```
2>&1
```

face ca `stderr` să indice aceeași destinație ca `stdout`.

În acel moment:

```
stdout → Terminal
```

```
stderr → Terminal
```

Abia apoi:

```
> raport.log
```

mută doar `stdout`.

Rezultatul final:

```
stdout → raport.log
```

```
stderr → Terminal
```

Acesta este unul dintre cele mai importante exemple din întregul capitol despre redirecționări.

Operatorul `&>`

Bash oferă și o formă prescurtată:

```
comanda &> raport.log
```

Aceasta redirecționează atât `stdout`, cât și `stderr` către același fișier.

Pentru Bash, efectul este echivalent cu:

comanda `> raport.log 2>&1`

Notă pentru autor: Aș menționa explicit că `&>` este o extensie Bash și nu este definită de POSIX. În schimb, `> fișier 2>&1` este portabil și funcționează și în alte shell-uri compatibile POSIX. Este o nuanță importantă pentru cursanții LPIC.

Caseta „În spatele cortinei”

Descriptorii sunt copiați, nu legați între ei

Este tentant să ne imaginăm că `stderr` este „trimis prin `stdout`”.

Nu asta se întâmplă.

Bash copiază destinația descriptorului 1 în descriptorul 2.

După aceea, fiecare descriptor continuă să existe separat.

Această explicație este mai fidelă modului în care funcționează sistemul de operare și ajută la înțelegerea importanței ordinii redirecționărilor.

Greșeală din lumea reală

Un administrator execută:

```
script.sh 2>&1 > jurnal.log
```

și observă că mesajele de eroare apar în continuare pe terminal.

Scriptul nu este defect.

Ordinea redirecționărilor este.

Forma corectă este:

```
script.sh > jurnal.log 2>&1
```

sau, dacă este acceptabilă utilizarea unei extensii Bash:

```
script.sh &> jurnal.log
```

Administrator Notes

Când vezi:

`2>&1`

nu încerca să îl memorezi.

Întreabă-te:

Unde indică descriptorul 1 în acel moment?

Acesta este răspunsul care determină unde vor ajunge mesajele de eroare.

Legătura cu LPIC

Acesta este unul dintre subiectele clasice ale examenului LPIC.

Trebuie să înțelegi:

- semnificația lui `2>&1`;
 - importanța ordinii redirectionărilor;
 - diferența dintre `2>&1` și `&>`.
-

Cum aş depana această problemă?

Problema

Rezultatele sunt salvate în fișier, dar mesajele de eroare apar încă pe terminal.

Pașii

1. Verific ordinea redirectionărilor.
 2. Confirm destinația descriptorului 1 înainte de utilizarea lui `2>&1`.
 3. Testez separat `stdout` și `stderr`.
 4. Dacă folosesc Bash și portabilitatea nu este o cerință, verific dacă `&>` exprimă mai clar intenția.
-

Laborator 7.5

Obiectiv

Înțelegerea redirectionării simultane a `stdout` și `stderr`.

Cerințe

Execută:

```
ls /etc /director_inexistent > rezultate.txt 2> erori.txt
```

Analizează conținutul celor două fișiere.

Execută apoi:

```
ls /etc /director_inexistent > jurnal.log 2>&1
```

Observă că ambele tipuri de mesaje sunt în același fișier.

Compară cu:

```
ls /etc /director_inexistent 2>&1 > jurnal.log
```

Răspunde:

1. Unde ajunge `stdout`?
 2. Unde ajunge `stderr`?
 3. De ce rezultatul diferă între cele două variante?
-

Soluția laboratorului

- `2>&1` face ca descriptorul 2 să utilizeze aceeași destinație ca descriptorul 1 în momentul executării redirectionării.
 - Ordinea redirectionărilor influențează rezultatul final.
 - `&>` este o formă prescurtată specifică Bash pentru redirectionarea simultană a `stdout` și `stderr`.
-

Ce trebuie să reții

- `2>&1` copiază destinația descriptorului 1 către descriptorul 2.
- Ordinea redirectionărilor este esențială.
- `> fișier 2>&1` și `2>&1 > fișier` nu sunt echivalente.
- `&>` este o sintaxă specifică Bash, convenabilă, dar neportabilă către toate shell-urile POSIX.
- Înțelegerea descriptorilor de fișiere este cheia stăpânirii redirectionărilor în Bash.

Capitolul 7

7.6 /dev/null – Destinația în care datele dispar

„Uneori nu dorim să salvăm rezultatul unei comenzi și nici să îl afișăm pe ecran. În Linux există un dispozitiv special care primește orice informație și o elimină. Acest dispozitiv este /dev/null.”

Ce vei învăța

În acest capitol vei înțelege ce este fișierul special /dev/null, de ce este atât de utilizat în administrarea sistemelor Linux și cum poate fi folosit împreună cu redirectionările.

Introducere

Până acum am învățat că putem redirectiona un flux către:

- terminal;
- un fișier;
- alt descriptor.

Mai există însă o destinație specială.

O destinație în care informația este pur și simplu ignorată.

Aceasta este:

/dev/null

Este unul dintre cele mai cunoscute fișiere speciale din Linux.

Ce este /dev/null?

/dev/null este un **fișier special de tip caracter (character device)**.

El nu stochează date.

Tot ceea ce este scris în acest dispozitiv este eliminat imediat.

Din acest motiv este cunoscut și sub denumirea informală de:

bit bucket sau **black hole**.

Primul exemplu

Executăm:

```
ls > /dev/null
```

Comanda rulează normal.

Dar lista fișierelor nu mai apare pe ecran.

Ea este trimisă către:

```
/dev/null
```

unde este eliminată.

Modelul mental

În mod normal:

```
stdout
```

|



Terminal

Cu redirectionare:

```
stdout
```

|



```
/dev/null
```

|



(date eliminate)

Programul nu observă nicio diferență.

El continuă să scrie pe `stdout`.

De ce este util?

Există numeroase situații în administrarea sistemelor.

De exemplu:

Dorim doar să verificăm codul de ieșire al unei comenzi.

Nu ne interesează rezultatul afișat.

Executăm:

```
grep "root" /etc/passwd > /dev/null
```

Acum putem verifica:

```
echo $?
```

Rezultatul comenzii a fost ignorat.

Dar codul de ieșire rămâne disponibil.

Ignorarea mesajelor de eroare

Executăm:

```
ls fisier_inexistent 2> /dev/null
```

Mesajul:

```
No such file or directory
```

nu mai apare.

El este eliminat.

Rezultatul normal (dacă există) continuă să fie trimis pe `stdout`.

Ignorarea ambelor fluxuri

Uneori nu dorim nici rezultate normale, nici mesaje de eroare.

Executăm:

```
comanda > /dev/null 2>&1
```

sau, în Bash:

```
comanda &> /dev/null
```

Acum programul rulează complet "în tăcere".

Caseta „În spatele cortinei”

Cine citește din /dev/null?

Dacă scriem:

```
echo "Salut" > /dev/null
```

datele sunt eliminate.

Dar dacă citim:

```
cat < /dev/null
```

programul primește imediat sfârșitul fișierului (**EOF**).

/dev/null se comportă astfel ca o sursă de intrare goală.

Acesta este un comportament folosit frecvent în scripting.

Un exemplu practic

Dorim să verificăm dacă un serviciu răspunde la un ping:

```
ping -c 1 server.example.com > /dev/null 2>&1
```

Nu ne interesează afișarea pachetelor.

Ne interesează doar dacă comanda s-a încheiat cu succes.

Într-un script putem scrie:

```
if ping -c 1 server.example.com > /dev/null 2>&1
then
    echo "Serverul răspunde."
else
    echo "Serverul nu răspunde."
fi
```

Observă că informația utilă este dată de codul de ieșire, nu de textul afișat.

Greșeală din lumea reală

Un administrator redirecționează toate mesajele către:

```
> /dev/null 2>&1
```

în timpul depanării unui script.

Scriptul nu funcționează.

Dar nici nu mai afișează erorile.

În loc să simplifice depanarea, aceasta o face aproape imposibilă.

Regula este simplă:

Nu trimite erorile în `/dev/null` până când nu ești sigur că scriptul funcționează corect.

Administrator Notes

`/dev/null` este un instrument foarte util, dar trebuie utilizat cu discernământ.

În producție poate elimina mesaje care sunt așteptate și irelevante.

În timpul dezvoltării sau depanării, este de regulă mai bine să vezi mesajele de eroare decât să le ascunzi.

Legătura cu LPIC

`/dev/null` este un subiect clasic în LPIC-1.

Trebuie să cunoști:

- rolul său ca dispozitiv special;
 - utilizarea împreună cu `>`, `2>` și `2>&1`;
 - diferența dintre eliminarea rezultatului și eliminarea erorilor.
-

Cum aş depana această problemă?

Problema

O comandă nu afișează nimic.

Pașii

1. Verific dacă rezultatul este redirecționat către `/dev/null`.
 2. Elimin temporar redirecționarea și execut din nou comanda.
 3. Verific separat `stdout` și `stderr`.
 4. După identificarea problemei, decid dacă este justificată eliminarea unuia dintre fluxuri.
-

Laborator 7.6

Obiectiv

Utilizarea dispozitivului `/dev/null`.

Cerințe

Execută:

```
echo "Salut" > /dev/null
```

Observă că nu apare niciun rezultat.

Execută:

```
ls fisier_inexistent 2> /dev/null
```

Observă că mesajul de eroare nu mai este afișat.

Execută:

```
ls /etc /director_inexistent > /dev/null
```

Observă că mesajele de eroare sunt încă afișate.

Execută apoi:

```
ls /etc /director_inexistent > /dev/null 2>&1
```

Observă că nici rezultatele normale, nici mesajele de eroare nu mai sunt afișate.

În final execută:

```
cat < /dev/null
```

Observă că programul se încheie imediat.

Soluția laboratorului

- `/dev/null` elimină orice date scrise în el.
 - Redirecționarea `> /dev/null` afectează doar `stdout`.
 - Redirecționarea `2> /dev/null` afectează doar `stderr`.
 - Combinarea `> /dev/null 2>&1` elimină ambele fluxuri.
 - Citirea din `/dev/null` produce imediat sfârșitul fișierului (EOF).
-

Ce trebuie să reții

- `/dev/null` este un dispozitiv special care elimină datele scrise în el.
- Poate fi folosit atât pentru `stdout`, cât și pentru `stderr`.
- Citirea din `/dev/null` produce imediat EOF.
- Este foarte util în scripturi atunci când ne interesează doar codul de ieșire al unei comenzi.
- Evită să ascunzi mesajele de eroare în timpul dezvoltării și depanării.

Capitolul 7

7.7 Pipe (|) – Cum comunică procesele între ele

„Una dintre cele mai elegante idei din Unix este posibilitatea de a conecta două programe astfel încât rezultatul unuia să devină automat intrarea celuilalt. Acest mecanism poartă numele de pipe.”

Ce vei învăța

În acest capitol vei înțelege ce este un **pipe**, cum creează Bash o conexiune între două procese și de ce această tehnică reprezintă unul dintre fundamentele filozofiei Unix.

Introducere

Până acum am învățat că putem redirecționa:

- `stdout` către un fișier;
- `stdin` dintr-un fișier.

Dar apare imediat o întrebare.

Dacă rezultatul unei comenzi trebuie prelucrat imediat de o altă comandă?

O soluție ar fi:

```
ls > fisiere.txt
```

```
sort < fisiere.txt
```

Funcționează.

Dar este nevoie de un fișier temporar.

Există o soluție mai elegantă.

Ideea fundamentală

Un **pipe** conectează:

`stdout` al primului proces

la

`stdin` al celui de-al doilea proces.

Datele circulă direct între procese.

Nu este necesar un fișier intermediar.

Primul exemplu

Executăm:

```
ls | sort
```

Ce se întâmplă?

Programul:

```
ls
```

scrie pe:

```
stdout
```

Bash conectează acel flux la:

```
stdin
```

al programului:

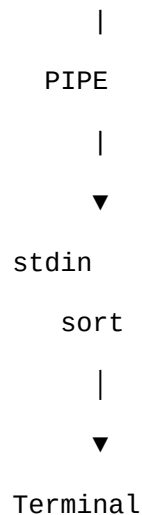
```
sort
```

Rezultatul afișat pe ecran este deja ordonat.

Modelul mental

```
stdout
```

```
ls -----+
```

Observă că cele două procese rulează în același timp.

Nu unul după altul.

Acesta este un aspect foarte important.

Pipe-ul nu înseamnă "execută apoi"

Foarte mulți începători cred că:

```
ls | sort
```

înseamnă:

1. rulează `ls`;
2. termină;
3. rulează `sort`.

Nu.

În majoritatea cazurilor cele două procese rulează simultan.

Primul produce date.

Al doilea le consumă pe măsură ce sosesc.

Această idee este fundamentală.

Exemplu practic

Executăm:

```
ps aux | grep ssh
```

Primul proces:

`ps`

produce lista proceselor.

Al doilea:

`grep`

filtrează doar liniile care conțin:

`ssh`

Niciun fișier temporar nu este creat.

Un alt exemplu

Executăm:

`dmesg | less`

`dmesg` produce foarte multe informații.

`less` le afișează pagină cu pagină.

Fără pipe, această comandă ar fi mult mai greu de utilizat.

Caseta „În spatele cortinei”

Ce creează Bash?

Înainte de lansarea comenzilor, Bash apelează sistemul de operare pentru a crea un **pipe anonim**.

Acest pipe are două capete:

- unul pentru scriere;
- unul pentru citire.

Apoi:

- `stdout` al primului proces este conectat la capătul de scriere;
- `stdin` al celui de-al doilea proces este conectat la capătul de citire.

Abia apoi Bash lansează cele două procese.

Acest mecanism este implementat în kernel și reprezintă una dintre cele mai elegante forme de comunicare între procese.

Notă pentru autor: Nu este nevoie să introducem încă apelurile de sistem `pipe()`, `fork()`, `dup2()` și `execve()`. Le vom prezenta într-un capitol dedicat proceselor și IPC. Aici este suficient ca cititorul să înțeleagă că Bash pregătește conexiunile înainte de lansarea programelor.

Pipe-ul transmite doar `stdout`

Acesta este un detaliu foarte important.

Executăm:

```
find / -name passwd | grep etc
```

Rezultatele normale ajung la `grep`.

Dar mesajele:

```
Permission denied
```

apar în continuare pe terminal.

De ce?

Pentru că pipe-ul conectează:

```
stdout
```

nu:

```
stderr
```

Acest comportament explică multe situații întâlnite în practică.

Greșeală din lumea reală

Un administrator execută:

```
find / -name passwd | wc -l
```

și observă că pe ecran apar în continuare erori.

Crede că pipe-ul nu funcționează.

În realitate, funcționează perfect.

Doar că mesajele de eroare continuă să fie trimise pe descriptorul 2.

Dacă dorește ca și acestea să intre în pipe, trebuie mai întâi să le redirecționeze către `stdout`.

Vom vedea imediat cum se combină pipe-urile cu redirecționările.

Administrator Notes

În filosofia Unix există un principiu celebru:

Construiește soluții complexe prin combinarea unor programe simple.

Pipe-ul este mecanismul care face posibil acest lucru.

În loc să cauți o comandă gigantică care face totul, încearcă să construiești un lanț de comenzi simple.

Acesta este stilul clasic de lucru în Unix și Linux.

Legătura cu LPIC

Pipe-urile reprezintă unul dintre subiectele fundamentale ale LPIC-1.

Trebuie să înțelegi:

- ce flux este conectat;
 - diferența dintre pipe și redirectionare;
 - faptul că procesele comunică prin fluxuri de date.
-

Cum aș depana această problemă?

Problema

Comanda din dreapta pipe-ului nu primește datele așteptate.

Pașii

1. Execut separat prima comandă și verific dacă produce rezultatul dorit.
 2. Confirm că informația este trimisă pe `stdout`, nu pe `stderr`.
 3. Execut separat a doua comandă cu date de test.
 4. Verific dacă ordinea comenzilor din pipe este corectă.
-

Laborator 7.7

Obiectiv

Utilizarea pipe-urilor pentru conectarea proceselor.

Cerințe

Execută:

```
ls | sort
```

Apoi:

```
ls -l | wc -l
```

Execută:

```
ps aux | grep ssh
```

și:

```
dmesg | less
```

(opțional, în funcție de permisiuni și distribuție)

Răspunde:

1. Care proces produce datele?
 2. Care proces le consumă?
 3. Ce flux este conectat prin pipe?
 4. De ce mesajele de eroare nu trec automat prin pipe?
-

Soluția laboratorului

- Primul proces produce date pe `stdout`.
 - Bash creează un pipe anonim între cele două procese.
 - Al doilea proces citește datele prin `stdin`.
 - Pipe-ul transmite doar fluxul `stdout`; `stderr` rămâne independent dacă nu este redirecționat explicit.
-

Ce trebuie să reții

- Pipe-ul (`|`) conectează `stdout` al unui proces la `stdin` al altui proces.
- Nu este necesar un fișier temporar.
- Procesele comunică printr-un mecanism oferit de kernel.
- În mod implicit, doar `stdout` este transmis prin pipe.
- Pipe-urile reprezintă una dintre ideile fundamentale ale filozofiei Unix și una dintre cele mai puternice tehnici din Bash.

Capitolul 7

7.8 tee – Cum salvăm și afișăm simultan rezultatul unei comenzi

„În mod normal, un pipe transmite datele către următoarea comandă, iar rezultatul nu mai este afișat pe terminal. Utilitarul `tee` rezolvă această limitare, permițând atât afișarea datelor, cât și salvarea lor într-un fișier.”

Ce vei învăța

În acest capitol vei înțelege rolul utilitarului `tee`, modul în care funcționează împreună cu pipe-urile și situațiile practice în care este preferat față de redirectionările clasice.

Introducere

Să presupunem că executăm:

```
ls | sort
```

Rezultatul este afișat pe ecran.

Acum dorim și să păstrăm rezultatul într-un fișier.

O soluție ar fi:

```
ls | sort > lista.txt
```

Problema?

Rezultatul nu mai apare pe ecran.

El ajunge doar în fișier.

Apare o întrebare firească.

Putem vedea rezultatul și, în același timp, să îl salvăm?

Da.

Pentru aceasta există:

```
tee
```

Ideea fundamentală

`tee` citește date de pe **stdin**.

Apoi:

- trimite datele către **stdout**;
- scrie aceleași date într-un fișier.

Practic, realizează o copie a fluxului de date.

Primul exemplu

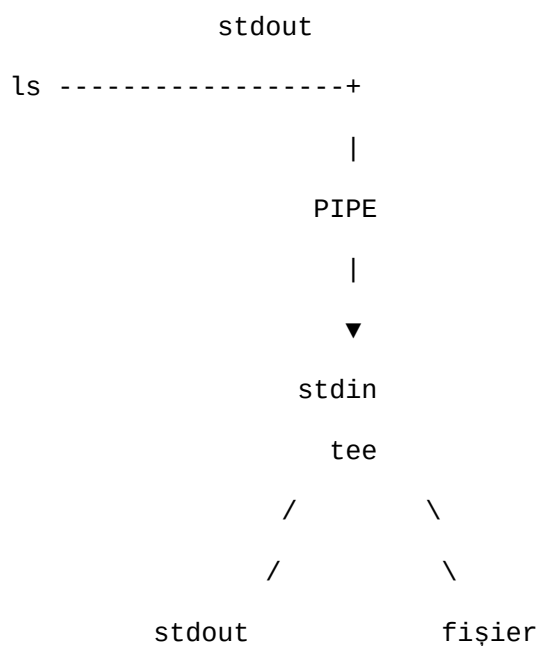
Executăm:

```
ls | tee fisiere.txt
```

Ce se întâmplă?

1. `ls` produce lista fișierelor.
 2. Pipe-ul transmite datele către `tee`.
 3. `tee` scrie aceleași date:
 - pe ecran;
 - în fișierul `fisiere.txt`.
-

Modelul mental



|



Terminal

Observă că `tee` are două ieșiri.

Acesta este motivul existenței sale.

De ce se numește `tee`?

Denumirea provine din instalațiile sanitare.

Piesa în formă de litera:

T

împarte conducta în două direcții.

Exact acest lucru îl face și utilitarul `tee`.

Un flux de intrare.

Două fluxuri de ieșire.

Exemplu practic

Executăm:

```
dmesg | tee boot.log
```

Mesajele de boot:

- sunt afișate pe terminal;
- sunt salvate și în:

`boot.log`

Putem analiza ulterior fișierul fără să pierdem informațiile afișate în timpul execuției.

Adăugarea la un fișier

Implicit, `tee` suprascrie fișierul.

Dacă dorim adăugarea informațiilor folosim:


```
tee -a jurnal.log
```

Exemplu:

```
date | tee -a jurnal.log
```

Acum fiecare execuție adaugă o nouă linie în jurnal.

Un exemplu din administrarea sistemelor

Dorim să verificăm spațiul ocupat pe disc și să păstrăm raportul.

Executăm:

```
df -h | tee raport-disc.txt
```

Administratorul vede imediat rezultatul.

În același timp, raportul este salvat pentru analiză sau arhivare.

Caseta „În spatele cortinei”

tee este un program obișnuit

Spre deosebire de operatorii `>`, `>>` sau `|`, `tee` nu este un mecanism intern al shell-ului.

Este un program obișnuit din spațiul utilizatorului.

Bash creează pipe-ul și pornește procesul `tee`.

De acolo înainte, `tee` citește de pe `stdin` și scrie simultan pe `stdout` și în fișier.

Acest exemplu ilustrează foarte bine filosofia Unix: un utilitar simplu, construit pentru o singură sarcină, care poate fi combinat cu alte programe.

Greșeală din lumea reală

Un administrator execută:

```
find / -name "*.conf" | tee configuratii.txt
```

și este surprins că mesajele:

```
Permission denied
```

nu apar în fișier.

Explicația este simplă.

Pipe-ul transmite doar:

`stdout`

Mesajele de eroare continuă să fie trimise pe:

`stderr`

Dacă dorim ca și acestea să ajungă în `tee`, trebuie să le redirecționăm mai întâi către `stdout`.

Exemplu:

```
find / -name "*.conf" 2>&1 | tee configuratii.txt
```

Acum `tee` primește ambele tipuri de mesaje.

Administrator Notes

`tee` este foarte util în următoarele situații:

- instalarea și configurarea serviciilor;
- rularea scripturilor lungi;
- depanare;
- colectarea jurnalelor;
- demonstrații și cursuri.

În multe cazuri, este preferabil să vezi rezultatul în timp real decât să consulți fișierul abia după terminarea comenzii.

Legătura cu LPIC

LPIC include `tee` în obiectivele privind redirecționările și pipe-urile.

Trebuie să cunoști:

- rolul său într-un pipeline;
 - diferența dintre `tee` și `>`;
 - opțiunea `-a`.
-

Cum aş depana această problemă?

Problema

Fișierul creat de `tee` nu conține toate mesajele afișate pe ecran.

Pași

1. Verific dacă mesajele provin din `stdout` sau `stderr`.
2. Dacă este necesar, folosesc:

```
2>&1
```

înaintea pipe-ului.

3. Verific dacă am utilizat `tee` sau `tee -a`.
 4. Confirm drepturile de scriere asupra fișierului.
-

Laborator 7.8

Obiectiv

Utilizarea utilitarului `tee`.

Cerințe

Execută:

```
ls | tee lista.txt
```

Observă că rezultatul apare și pe terminal, și în fișier.

Execută apoi:

```
date | tee jurnal.txt
```

Repetă comanda folosind:

```
date | tee -a jurnal.txt
```

Compară conținutul fișierului.

Execută:

```
find /etc -name "*.conf" 2>&1 | tee configuratii.log
```

Analizează rezultatul.

Întrebări

1. Ce citește `tee`?
 2. Ce produce `tee`?
 3. De ce `tee` este utilizat împreună cu pipe-urile?
 4. Care este diferența dintre `tee` și `tee -a`?
-

Soluția laboratorului

- `tee` citește datele din `stdin`.
 - Scrie simultan pe `stdout` și într-un fișier.
 - `tee` este utilizat pentru a vizualiza și salva același flux de date.
 - Opțiunea `-a` adaugă datele la sfârșitul fișierului în loc să îl suprascrie.
-

Ce trebuie să reții

- `tee` este un utilitar care copiază un flux de date.
- Funcționează în mod natural împreună cu pipe-urile.
- Poate afișa rezultatul și îl poate salva simultan într-un fișier.
- `tee -a` adaugă informațiile la sfârșitul fișierului.
- Pentru a salva și mesajele de eroare, acestea trebuie redirecționate către `stdout` înainte de pipe.

Capitolul 7

7.9 Here Document (<<) – Cum furnizăm blocuri de text unui program

„Uneori un program are nevoie de mai multe linii de intrare. În loc să folosim un fișier temporar, Bash permite includerea directă a textului în script. Acest mecanism poartă numele de Here Document.”

Ce vei învăța

În acest capitol vei înțelege ce este un **Here Document**, cum funcționează și în ce situații este preferabil față de utilizarea fișierelor temporare.

Introducere

Presupunem că dorim să transmitem următorul text către comanda `cat`:

```
Linux
Bash
LPIC-1
```

O soluție ar fi:

1. să creăm un fișier;
2. să scriem textul în fișier;
3. să executăm:

```
cat < fisier.txt
```

Funcționează.

Dar este nevoie de un fișier temporar.

Bash oferă o soluție mai elegantă.

Ideea fundamentală

Un **Here Document** spune lui Bash:

„Până când întâlnești acest marcaj, consideră toate liniile următoare ca fiind intrarea standard a programului.”

Sintaxa

Forma generală este:

```
comanda <<DELIMITATOR  
linie 1  
linie 2  
linie 3  
DELIMITATOR
```

Marcajul poate avea aproape orice nume.

Cele mai întâlnite sunt:

```
EOF  
END  
STOP  
TEXT
```

Important este ca marcajul de început și cel de sfârșit să fie identice.

Primul exemplu

```
cat <<EOF  
Linux  
Bash
```

LPIC-1
EOF

Rezultatul:

Linux
Bash
LPIC-1

Observă că nu există niciun fișier.

Totuși `cat` citește din `stdin`.

Modelul mental

Textul din script

|



Bash

|



`stdin`

|



Program

Programul nu știe că datele provin din script.

El citește pur și simplu din `stdin`.

Un exemplu practic

Crearea unui fișier de configurare:

```
cat > configuratie.conf <<EOF
server=192.168.1.10
port=8080
protocol=https
EOF
```

Bash transmite blocul de text către `cat`, iar `cat` îl scrie în fișier.

Această tehnică este foarte întâlnită în scripturile de instalare și configurare.

Parameter Expansion într-un Here Document

Implicit, Bash aplică expansiunile în interiorul unui Here Document.

Exemplu:

```
NUME="Daniel"
```

```
cat <<EOF
Salut, $NUME!
EOF
```

Rezultatul:

```
Salut, Daniel!
```

La fel funcționează și:

- Command Substitution;
- Arithmetic Expansion.

Exemplu:

```
cat <<EOF
Astăzi este $(date +%F)
2 + 3 = $((2+3))
EOF
```

Cum dezactivăm expansiunile?

Uneori dorim ca textul să fie transmis exact așa cum este scris.

În acest caz delimitatorul este pus între ghilimele simple:

```
cat <<'EOF'
$HOME
$(date)
$((2+3))
EOF
```

Rezultatul va fi:

```
$HOME
$(date)
$((2+3))
```

Bash nu mai aplică nicio expansiune.

Această tehnică este foarte utilă atunci când generăm scripturi Bash sau fișiere de configurare care conțin caractere speciale.

Caseta „În spatele cortinei”

Cum funcționează?

Când Bash întâlnește operatorul <<:

1. citește toate liniile până la delimitator;
2. aplică sau nu expansiunile, în funcție de quoting-ul delimitatorului;
3. pregătește aceste date ca intrare standard;
4. lansează programul.

Programul citește din `stdin` fără să știe dacă datele provin din tastatură, dintr-un fișier sau dintr-un Here Document.

Greșeală din lumea reală

Un administrator scrie:

```
cat <<EOF  
HOME=$HOME  
EOF
```

și este surprins că apare valoarea variabilei.

Intenția lui era să creeze un fișier care să conțină textul literal:

```
HOME=$HOME
```

Soluția este:

```
cat <<'EOF'  
HOME=$HOME  
EOF
```

Acum Bash nu mai efectuează expansiunea variabilei.

Administrator Notes

Here Document este una dintre cele mai utilizate tehnici în:

- scripturi de instalare;
- generarea fișierelor de configurare;
- automatizarea comenzilor interactive;
- documentarea exemplelor în scripturi.

Dacă blocul de text conține variabile care trebuie evaluate, folosește delimitatorul fără quoting.

Dacă dorești text literal, folosește delimitatorul între ghilimele simple.

Legătura cu LPIC

LPIC include Here Documents în obiectivele referitoare la scripting Bash.

Trebuie să cunoști:

- sintaxa <<;
 - rolul delimitatorului;
 - diferența dintre delimitatorul simplu și cel pus între ghilimele.
-

Cum aş depana această problemă?

Problema

Variabilele din Here Document sunt expandate atunci când nu ar trebui.

Paşii

1. Verific delimitatorul (EOF sau 'EOF').
 2. Confirm dacă expansiunile sunt dorite.
 3. Testez conținutul rezultat.
 4. Dacă este necesar text literal, folosesc delimitatorul între ghilimele simple.
-

Laborator 7.9

Obiectiv

Utilizarea Here Documents.

Cerințe

Execută:

```
cat <<EOF
Linux
Bash
LPIC
EOF
```

Apoi:

```
NUME="Student"
```

```
cat <<EOF  
Salut, $NUME!  
EOF
```

Compară cu:

```
NUME="Student"
```

```
cat <<'EOF'  
Salut, $NUME!  
EOF
```

În final creează un fișier:

```
cat > mesaj.txt <<EOF  
Acesta este un fișier creat prin Here Document.  
Data: $(date +%F)  
EOF
```

Analizează conținutul fișierului.

Soluția laboratorului

- << transmite un bloc de text către stdîn.
 - Delimitatorul stabilește începutul și sfârșitul blocului.
 - Dacă delimitatorul nu este citat, Bash aplică expansiunile.
 - Dacă delimitatorul este între ghilimele simple, textul este transmis literal.
-

Ce trebuie să reții

- Here Document (<<) furnizează un bloc de text ca intrare standard.
- Nu este necesar un fișier temporar.
- Bash poate aplica Parameter Expansion, Command Substitution și Arithmetic Expansion în interiorul blocului.
- Delimitatorul citat (' EOF ') dezactivează aceste expansiuni.
- Here Documents sunt folosite frecvent pentru generarea fișierelor de configurare și pentru automatizarea scripturilor.

Capitolul 7.11

Cum aleg mecanismul potrivit de transfer al datelor?

„Bash oferă mai multe mecanisme pentru transmiterea datelor către un program. Deși uneori pot produce același rezultat, fiecare dintre ele este proiectat pentru un anumit tip de problemă.”

Situația 1 – Datele există deja într-un fișier

Folosește:

<

Exemplu:

```
sort < utilizatori.txt
```

Aici fișierul există deja.

Nu trebuie creat.

Situația 2 – Datele sunt generate de alt program

Folosește:

|

Exemplu:

```
ps aux | grep ssh
```

Nu are sens să creezi un fișier temporar.

Datele există deja în memorie.

Situația 3 – Ai doar un șir scurt de caractere

Folosește:

<<<

Exemplu:

```
grep root <<< "root:x:0:0"
```

Nu merită creat un fișier.

Nu merită un Here Document.

Ai o singură valoare.

Situația 4 – Ai mai multe linii de text

Folosește:

<<

Exemplu:

```
cat <<EOF  
Linia 1  
Linia 2  
Linia 3  
EOF
```

Acesta este exact scopul Here Document.

Situația 5 – Vrei și să vezi rezultatul și să-l salvezi

Folosește:

tee

Exemplu:

```
dmesg | tee boot.log
```

Situația 6 – Nu te interesează rezultatul

Folosește:

```
> /dev/null
```

sau

```
2> /dev/null
```

sau

```
> /dev/null 2>&1
```

în funcție de fluxul pe care dorești să îl ignori.

Tabel de sinteză

Situație	Soluția recomandată	Motiv
Date într-un fișier	<	Programul citește din <code>stdin</code> , iar Bash conectează fișierul la descriptorul 0.
Date produse de alt program	,	,
Un singur șir de caractere	<<<	Sintaxă simplă pentru intrări scurte.
Mai multe linii de text	<<	Ideal pentru scripturi și generarea de fișiere de configurare.
Afișare și salvare simultană	tee	Copiază fluxul către terminal și fișier.
Ignorarea rezultatului	/dev/null	Elimină datele fără a afecta execuția programului.

Cum gândește un administrator experimentat?

Administratorul experimentat nu pornește de la sintaxă.

El își pune câteva întrebări simple:

- De unde provin datele?
- Unde trebuie să ajungă?
- Trebuie păstrate?
- Trebuie procesate de alt program?
- Mă interesează și mesajele de eroare?

Răspunsurile la aceste întrebări conduc aproape întotdeauna la alegerea corectă a mecanismului.

Administrator Notes

Una dintre cele mai frecvente greșeli ale începătorilor este utilizarea fișierelor temporare acolo unde un pipe sau un Here Document ar fi suficiente.

De exemplu:

```
ps aux > procese.txt  
grep ssh procese.txt  
rm procese.txt
```

poate fi înlocuit elegant cu:

```
ps aux | grep ssh
```

Scriptul devine:

- mai scurt;
- mai rapid;
- mai ușor de întreținut;
- mai apropiat de stilul clasic Unix.

Capitolul 7

7.10 Here String (<<<) – Cum transmitem rapid un șir de caractere către un program

„Uneori dorim să transmitem un singur șir de caractere unui program fără a crea un fișier și fără a construi un Here Document. Pentru aceste situații, Bash oferă operatorul <<<, denumit Here String.”

Ce vei învăța

În acest capitol vei înțelege ce este un Here String, cum funcționează și când reprezintă alegerea potrivită față de un pipe sau un Here Document.

Introducere

În capitolul anterior am văzut că un Here Document permite transmiterea unui bloc întreg de text.

Dar dacă avem doar o singură valoare?

Exemplu:

```
Linux
```

sau

```
root:x:0:0
```

Crearea unui Here Document ar fi inutil de complicată.

În această situație folosim:

<<<

Ideea fundamentală

Un **Here String** spune lui Bash:

„Ia acest șir de caractere și furnizează-l ca intrare standard (**stdin**) programului.”

Sintaxa

Forma generală este:

comanda <<< "text"

Exemplu:

cat <<< "Linux"

Rezultatul este:

Linux

Programul citește din **stdin**.

Primul exemplu

Executăm:

wc -c <<< "Linux"

Rezultatul este:

6

Mulți cursanți se așteaptă la 5.

De ce apare 6?

Pentru că Bash transmite șirul urmat de un caracter **newline** (**\n**).

Acesta este un detaliu important și merită explicat explicit.

Modelul mental

"Linux"

|

▼

Bash

|

▼

stdin

|

▼

Program

Programul nu știe că datele provin dintr-un Here String.

El vede doar stdin.

Un exemplu practic

Executăm:

```
grep root <<< "root:x:0:0:root:/root:/bin/bash"
```

Rezultatul:

```
root:x:0:0:root:/root:/bin/bash
```

Nu este necesar un fișier.

Nu este necesar un pipe.

Here String și variabile

Exemplu:

```
UTILIZATOR="daniel"
```

```
grep dan <<< "$UTILIZATOR"
```

Mai întâi Bash realizează **Parameter Expansion**.

Apoi transmite rezultatul către stdin.

Here String versus Pipe

Compară:

```
echo "Linux" | wc -c
```

cu:

```
wc -c <<< "Linux"
```

Ambele funcționează.

Dar mecanismul este diferit.

În primul caz:

- Bash pornește două procese;
- creează un pipe între ele.

În al doilea caz:

- Bash pregătește direct `stdin` pentru un singur proces.

Pentru un șir scurt, Here String este mai simplu și exprimă mai clar intenția.

Notă pentru autor: Nu aș afirma că Here String este întotdeauna „mai rapid”. Diferențele de performanță sunt de regulă neglijabile pentru astfel de exemple. Este mai corect să spunem că este **mai expresiv** pentru această categorie de probleme.

Here String versus Here Document

Dacă avem:

o singură valoare

↓

```
grep root <<< "$linie"
```

Dacă avem mai multe linii

↓

```
cat <<EOF
Linia 1
Linia 2
Linia 3
EOF
```

Alegerea devine imediat evidentă.

Caseta „În spatele cortinei”

Cum funcționează?

Când Bash întâlnește operatorul:

```
<<<
```

el:

1. evaluează expansiunile necesare;
2. adaugă un caracter newline la sfârșitul șirului;
3. pregătește acel text ca intrare standard;
4. lansează programul.

Programul citește datele din `stdin` fără să știe că acestea provin dintr-un Here String.

Greșeală din lumea reală

Un cursant scrie:

```
wc -c <<< "abc"
```

și este surprins că rezultatul este:

```
4
```

Nu este o eroare.

Caracterul newline adăugat automat este numărat și el.

Acest comportament este normal și trebuie cunoscut.

Administrator Notes

Folosește Here String atunci când:

- ai o singură valoare;
- vrei să o trimiți către un program care citește din `stdin`;
- nu ai nevoie de un bloc mare de text.

Dacă observi că Here String începe să conțină mai multe linii sau logică complexă, este probabil momentul să treci la un Here Document.

Legătura cu LPIC

Here String apare în obiectivele dedicate shell scripting.

Trebuie să cunoști:

- sintaxa <<<;
- diferența față de <<;
- faptul că furnizează date prin `stdin`.

De asemenea, este util să știi că <<< este o **extensie Bash** și nu face parte din standardul POSIX.

Cum aș depana această problemă?

Problema

Programul primește un caracter în plus la sfârșitul datelor.

Pașii

1. Verific dacă folosesc <<<.
 2. Îmi amintesc că Bash adaugă automat un caracter newline.
 3. Confirm dacă programul interpretează acest caracter.
 4. Dacă este nevoie de control strict asupra octeților transmiși, folosesc o altă metodă (de exemplu `printf` și un pipe).
-

Laborator 7.10

Obiectiv

Utilizarea Here String.

Cerințe

Execută:

```
cat <<< "Linux"
```

Apoi:

```
wc -c <<< "Linux"
```

Execută:

```
UTILIZATOR="daniel"
```

```
grep dan <<< "$UTILIZATOR"
```

Compară cu:

```
echo "$UTILIZATOR" | grep dan
```

Răspunde:

1. Ce primește programul pe `stdin`?
 2. De ce `WC -c` afișează un caracter în plus?
 3. Când este mai potrivit Here String decât Here Document?
-

Soluția laboratorului

- Here String transmite un singur șir către `stdin`.
 - Bash adaugă automat un caracter newline la sfârșitul șirului.
 - Este ideal pentru intrări scurte și simple.
 - Pentru blocuri de text cu mai multe linii este preferabil Here Document.
-

Ce trebuie să reții

- `<<<` transmite un șir de caractere către `stdin`.
- Este o extensie specifică Bash.
- Bash adaugă automat un caracter newline la finalul șirului.
- Here String este potrivit pentru date scurte; Here Document este alegerea potrivită pentru blocuri de text.

Capitolul 8

8.1 Ce este un script Bash?

„Un script Bash este un fișier text care conține o succesiune de comenzi interpretate de shell. El permite automatizarea unor activități repetitive și reprezintă una dintre cele mai importante unelte ale administratorului Linux.”

Ce vei învăța

În acest capitol vei înțelege ce este un script Bash, cum diferă de executarea comenzilor interactive și de ce scriptingul reprezintă una dintre competențele fundamentale ale unui administrator Linux.

Introducere

Până acum am executat comenzile una câte una.

Exemplu:

```
date
```

```
whoami
```

```
pwd
```

```
ls
```

Fiecare comandă era introdusă manual.

Dar să presupunem că în fiecare dimineață executăm aceeași secvență de comenzi:

```
date
```

```
hostname
```

```
uptime
```

```
df -h
```

```
free -h
```

```
ip addr
```

După câteva zile apare întrebarea firească:

De ce să introduc manual aceleași comenzi de fiecare dată?

Răspunsul este simplu.

Le putem pune într-un fișier.

Acest fișier se numește **script**.

Ideea fundamentală

Un script Bash este:

un fișier text care conține comenzi Bash executate în ordinea în care apar.

Nu este compilat.

Nu produce un fișier executabil în sensul limbajelor C sau C++.

Este interpretat.

Primul exemplu

Fișierul:

informatii.sh

conține:

date
hostname
uptime

La execuție se întâmplă exact ceea ce s-ar întâmpla dacă utilizatorul ar introduce aceste trei comenzi una după alta.

Script versus sesiune interactivă

Într-o sesiune interactivă:

utilizator

↓

introduce o comandă

↓

Bash o execută

↓

așteaptă următoarea comandă

Într-un script:

fișier

↓

Bash citește prima comandă

↓

o execută

↓

citește următoarea

↓

o execută

↓

...

↓

sfârșitul fișierului

Observă că mecanismul de execuție este același.

Se schimbă doar sursa comenzilor.

Scriptul nu este un limbaj diferit

Aceasta este una dintre cele mai frecvente confuzii.

Unii cursanți cred că există:

- comenzi Bash;
- scripting Bash.

Nu.

Scripturile folosesc exact aceleași comenzi pe care le-ai învățat deja.

Diferența este că acum ele sunt:

- organizate;
 - reutilizabile;
 - automatizabile.
-

De ce folosim scripturi?

Există cel puțin cinci motive.

1. Automatizare

În loc să executăm manual zece comenzi:

`backup.sh`

execută tot procesul.

2. Repetabilitate

Aceeași secvență de pași este executată identic de fiecare dată.

Acest lucru reduce riscul erorilor umane.

3. Documentare

Un script bine scris documentează procesul.

Este mai ușor să citești:

`backup.sh`

decât să încerci să îți amintești zece comenzi executate manual.

4. Partajare

Un script poate fi trimis unui coleg.

Acesta poate executa exact aceeași procedură.

5. Întreținere

Dacă procedura se modifică, schimbarea se face într-un singur loc.

Toți utilizatorii beneficiază imediat de noua versiune.

Caseta „În spatele cortinei”

Un script este doar un fișier text

Acesta este un lucru foarte important.

Un script Bash nu are un format binar special.

Poți verifica acest lucru:

`file script.sh`

vei obține un rezultat asemănător cu:

`script.sh: Bourne-Again shell script, ASCII text executable`

sau

`ASCII text`

în funcție de conținut și de distribuția Linux.

Aceasta explică de ce scripturile pot fi editate cu orice editor de text.

Interpretare versus compilare

În limbajul C:

cod sursă

↓

compilator

↓

fișier executabil

În Bash:

script

↓

bash

↓

execuție

Nu există o etapă de compilare.

Interpreterul citește comenzile și le execută.

Notă pentru autor: Aș evita expresia „Bash execută linie cu linie”, deoarece nu este întotdeauna exactă. Bash analizează și interpretează scriptul conform gramaticii sale, iar unele construcții sunt procesate ca blocuri logice. Pentru nivelul LPIC putem spune că execută comenzile în ordinea în care apar, respectând structurile de control.

Greșeală din lumea reală

Un începător modifică un script și întreabă:

„Trebuie să îl compilez din nou?”

Nu.

Este suficient să salvezi fișierul.

La următoarea execuție, Bash va interpreta noua versiune.

Administrator Notes

Scripturile trebuie privite ca instrumente de automatizare.

Dacă observi că repeți aceeași succesiune de comenzi de mai multe ori, întreabă-te:

Merită să transform această procedură într-un script?

În majoritatea cazurilor, răspunsul este „da”.

Legătura cu LPIC

LPIC presupune că înțelegi:

- ce este un script Bash;
 - diferența dintre interpretare și compilare;
 - rolul interpreterului Bash;
 - avantajele automatizării prin scripting.
-

Cum aş proiecta această automatizare?

Problema

Execut zilnic aceeași succesiune de comenzi.

Pașii

1. Identific comenzile care se repetă.
2. Le grupez într-un fișier text.
3. Testez execuția lor în aceeași ordine.
4. Adaug comentarii și mesaje pentru a face scriptul ușor de înțeles și întreținut.
5. Transform procedura manuală într-o procedură reutilizabilă.

Observă că încă nu am discutat despre `#!/bin/bash` sau despre permisiunile de execuție. Acestea vor veni în capitolele următoare.

Laborator 8.1

Nivel 1 – Înțelegere

Creează fișierul:

`informatii.sh`

și adaugă:

```
date  
hostname  
pwd  
whoami
```

Rulează comenzile manual.

Apoi rulează-le din fișier (vom vedea în capitolele următoare modalitățile de execuție).

Observă că rezultatele sunt identice.

Nivel 2 – Aplicare

Creează un script care afișează:

- data și ora;
- utilizatorul curent;
- directorul curent;
- timpul de funcționare al sistemului (`uptime`).

Adaugă comentarii care explică fiecare secțiune.

Nivel 3 – Administrator

Imaginează-ți că în fiecare dimineață verifici starea unui server.

Realizează un script care execută:

```
date  
  
hostname  
  
uptime  
  
free -h  
  
df -h  
  
ip addr show
```

La acest nivel nu urmărim încă formatarea avansată sau tratarea erorilor. Scopul este să construim primul script util pentru administrarea unui sistem Linux.

Ce trebuie să reții

- Un script Bash este un fișier text care conține comenzi Bash.

- Scripturile permit automatizarea activităților repetitive.
- Bash interpretează scripturile; acestea nu necesită compilare.
- Comenzile dintr-un script sunt aceleași comenzi folosite în sesiunea interactivă.
- Un script bine scris economisește timp, reduce erorile și facilitează administrarea sistemelor.

Capitolul 8

8.2 Shebang (#!) – Cum știe Linux cu ce interpretor trebuie executat un script

„Un fișier text nu conține informații despre modul în care trebuie executat. Linia Shebang îi spune nucleului Linux ce program trebuie folosit pentru interpretarea scriptului.”

Ce vei învăța

În acest capitol vei înțelege ce este Shebang-ul, de ce trebuie să apară pe prima linie a unui script și cum colaborează kernelul Linux cu Bash pentru executarea unui fișier text.

Introducere

Să presupunem că avem fișierul:

```
backup.sh
```

care conține:

```
date  
hostname  
uptime
```

Întrebarea este simplă.

Cum știe sistemul de operare că acest fișier trebuie executat de Bash și nu de Python sau Perl?

Răspunsul este:

```
#!/bin/bash
```

Această linie poartă numele de **Shebang**.

De unde vine numele?

Denumirea provine din primele două caractere:

`#!`

În jargonul Unix:

- `#` se numește **hash**;
- `!` se numește **bang**.

Împreună formează:

shebang

Este un termen tradițional în lumea Unix și Linux.

Primul exemplu

```
#!/bin/bash
```

```
date  
hostname  
uptime
```

Aceasta este forma clasică.

O observație importantă

Shebang-ul trebuie să fie:

- prima linie din fișier;
- primele două caractere ale fișierului.

Nu trebuie să existe:

- spații înaintea lui;
- linii goale înaintea lui;
- comentarii înaintea lui.

Exemplu corect:

```
#!/bin/bash  
echo "Salut!"
```

Exemplu incorect:

```
#!/bin/bash
```

sau

```
#!/bin/bash
```

În aceste cazuri kernelul nu va recunoaște Shebang-ul.

Cine interpretează Shebang-ul?

Mulți utilizatori cred că:

Bash citește Shebang-ul.

Nu.

Acesta este un aspect foarte important.

Kernelul Linux este cel care citește Shebang-ul.

Modelul mental

Utilizator

|

▼

./backup.sh

|

▼

Kernel Linux

|

▼

Citește prima linie

|

▼

#!/bin/bash

|

▼

Pornește /bin/bash

|

▼

Bash interpretează scriptul

Observă ordinea.

Bash nici măcar nu rulează încă atunci când este citit Shebang-ul.

Ce se întâmplă în spatele cortinei?

Atunci când executăm:

```
./backup.sh
```

Bash solicită kernelului:

„Execută acest fișier.”

Kernelul verifică:

- este un executabil ELF?
- este un script cu Shebang?
- este alt tip de fișier executabil?

Dacă întâlnește:

```
#!...
```

pornește programul indicat după Shebang și îi transmite scriptul ca argument.

Simplificat, efectul este asemănător cu:

```
/bin/bash backup.sh
```

De aceea, atunci când rulăm:

```
bash backup.sh
```

Shebang-ul nu mai este cel care decide interpretul. Noi l-am ales deja, lansând explicit bash.

De ce este util?

Același mecanism funcționează pentru multe limbaje.

Exemple:

Python

```
#!/usr/bin/python3
```

Perl

```
#!/usr/bin/perl
```

PHP

```
#!/usr/bin/php
```

Ruby

```
#!/usr/bin/ruby
```

Kernelul nu trebuie să cunoască limbajul.

Trebuie doar să știe ce program trebuie lansat.

/bin/bash sau /usr/bin/env bash?

Aceasta este una dintre cele mai discutate întrebări.

Forma clasică:

```
#!/bin/bash
```

Avantaje:

- simplă;
- directă;
- foarte răspândită.

Forma:

```
#!/usr/bin/env bash
```

înseamnă:

„Folosește programul `env` pentru a găsi executabilul `bash` în variabila `PATH`.”

Avantaje:

- mai portabilă între sisteme unde Bash poate fi instalat în locații diferite;
- foarte întâlnită în proiectele open-source.

Dezavantaj:

- depinde de configurarea corectă a variabilei `PATH`.

Notă pentru autor: Pentru LPIC și administrarea sistemelor Linux clasice, aș prezenta mai întâi `#!/bin/bash`, apoi aș explica `#!/usr/bin/env bash` ca variantă orientată spre portabilitate. Astfel, cursantul înțelege întâi mecanismul, apoi compromisul de proiectare.

Caseta „În spatele cortinei”

Ce se întâmplă dacă lipsește Shebang-ul?

Dacă executăm:

```
./script.sh
```

și fișierul este executabil, dar nu are Shebang, comportamentul depinde de modul în care este lansat și de shell-ul utilizat. În multe situații, shell-ul interactiv va încerca să îl execute ca script după ce kernelul refuză execuția directă, însă acest comportament este specific shell-ului și nu trebuie considerat garantat.

În schimb:

```
bash script.sh
```

funcționează deoarece noi lansăm explicit interpretul Bash.

De aceea Shebang-ul este recomandat pentru toate scripturile destinate reutilizării.

Greșeală din lumea reală

Un administrator scrie:

```
#!/bin/bash
```

dar execută scriptul astfel:

```
sh script.sh
```

și observă că anumite funcționalități Bash nu mai merg.

Explicația este simplă.

Shebang-ul este ignorat.

Interpretul este deja ales:

```
sh
```

Acesta poate fi:

- `dash`;
- `bash` în mod compatibil POSIX;
- alt shell.

Dacă scriptul utilizează extensii Bash, trebuie executat cu Bash.

Acesta este unul dintre motivele pentru care vom avea mai târziu un capitol despre diferențele dintre POSIX `sh` și Bash.

Administrator Notes

Adaugă întotdeauna un Shebang explicit în scripturile pe care intenționezi să le reutilizezi sau să le distribuie.

Acest lucru:

- clarifică interpretul dorit;
- reduce ambiguitățile;
- facilitează depanarea.

Legătura cu LPIC

Shebang-ul este un subiect clasic în LPIC.

Trebuie să cunoști:

- sintaxa `#!`;
- rolul interpretului;
- diferența dintre executarea:

`./script.sh`

și

`bash script.sh`

Cum aş depana această problemă?

Problema

Scriptul produce erori de sintaxă pe un sistem, deși funcționează pe altul.

Pașii

1. Verific Shebang-ul.
2. Confirm că interpretul există în locația indicată.
3. Verific dacă scriptul este lansat prin `./script.sh` sau prin `sh script.sh`.
4. Dacă scriptul folosește extensii Bash, mă asigur că este executat de Bash.

Laborator 8.2

Nivel 1 – Înțelegere

Creează fișierul:

```
#!/bin/bash
```

```
echo "Salut din Bash!"
```

Execută-l după ce îi vei acorda permisiuni de execuție (în capitolul următor).

Nivel 2 – Aplicare

Creează două scripturi identice:

Primul:

```
#!/bin/bash
```

Al doilea:

```
#!/usr/bin/env bash
```

Folosește:

```
which bash
```

și:

```
echo "$PATH"
```

pentru a înțelege de ce ambele variante funcționează pe sistemul tău.

Nivel 3 – Administrator

Scrie un script care afișează:

- calea completă a interpretului Bash (`which bash`);
- versiunea Bash (`bash --version`);
- utilizatorul curent;
- data și ora.

Adaugă un Shebang corespunzător și pregătește scriptul pentru distribuirea către un alt administrator.

Ce trebuie să reții

- Shebang-ul (`#!`) indică interpretul care trebuie folosit pentru executarea unui script.
- Este interpretat de kernel înainte ca Bash să înceapă execuția scriptului.
- Trebuie să fie prima linie și primele caractere din fișier.
- `#!/bin/bash` este forma clasică; `#!/usr/bin/env bash` favorizează portabilitatea.
- Lansarea explicită a unui interpret (`bash script.sh`, `sh script.sh`) ocolește rolul Shebang-ului în alegerea interpretului.

Capitolul 8

8.3 Permisele de execuție ale unui script

„Un script este un fișier text. Pentru a putea fi executat direct, utilizatorul trebuie să aibă dreptul de execuție asupra fișierului.”

Ce vei învăța

În acest capitol vei înțelege de ce un script nu poate fi executat direct imediat după creare, ce rol are permisiunea de execuție și care este diferența dintre executarea unui script prin Bash și executarea sa directă.

Introducere

Presupunem că avem scriptul:

```
#!/bin/bash  
  
echo "Salut!"
```

Încercăm să-l executăm:

```
./script.sh
```

Rezultatul poate fi:

```
Permission denied
```

De ce?

Fișierul există.

Conținutul este corect.

Shebang-ul este corect.

Problema este că fișierul nu are permisiunea de execuție.

Verificarea permisiunilor

Executăm:

```
ls -l script.sh
```

Rezultat:

```
-rw-r--r-- 1 student student 45 Jul 31 10:15 script.sh
```

Observăm că lipsesc biții de execuție (x).

Acordarea permisiunii de execuție

Executăm:

```
chmod +x script.sh
```

Verificăm din nou:

```
ls -l script.sh
```

Rezultat:

```
-rwxr-xr-x 1 student student 45 Jul 31 10:16 script.sh
```

Acum scriptul poate fi executat direct.

Executarea scriptului

```
./script.sh
```

Rezultat:

```
Salut !
```

Două moduri de executare

Există două metode frecvente.

Metoda 1

Executare directă:

```
./script.sh
```

Condiții:

- fișier executabil;
 - Shebang valid.
-

Metoda 2

Executare prin Bash:

```
bash script.sh
```

În acest caz:

- nu este necesară permisiunea de execuție;
 - Shebang-ul nu este folosit pentru alegerea interpretului, deoarece acesta este deja specificat.
-

Comparație

Comandă	Permișiune de execuție	Shebang necesar
<code>./script.sh</code>	Da	Da (recomandat)
<code>bash script.sh</code>	Nu	Nu (dar recomandat)

Caseta „În spatele cortinei”

Atunci când executăm:

```
bash script.sh
```

Bash deschide fișierul și îl interpretează direct.

Atunci când executăm:

```
./script.sh
```

kernelul verifică permisiunile fișierului și, dacă execuția este permisă, folosește Shebang-ul pentru a lansa interpretul corespunzător.

Greșeală din lumea reală

Administratorul creează un script și îl testează astfel:

```
bash backup.sh
```

Scriptul funcționează.

Îl trimite unui coleg, care execută:

```
./backup.sh
```

și primește:

```
Permission denied
```

Motivul este simplu.

Fișierul nu avea permisiunea de execuție.

Administrator Notes

În proiectele reale este recomandat ca toate scripturile destinate executării directe să aibă:

- un Shebang corect;
- permisiunea de execuție activată.

Astfel ele pot fi lansate direct din shell fără a specifica explicit interpretul.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul permisiunii de execuție;
- utilizarea comenzii `chmod +x`;
- diferența dintre:

```
./script.sh
```

și

```
bash script.sh
```

Laborator 8.3

Nivel 1 – Înțelegere

Creează fișierul:

```
#!/bin/bash
```

```
echo "Primul meu script"
```

Încearcă să îl execuți:

```
./script.sh
```

Observă rezultatul.

Acordă apoi permisiunea de execuție:

```
chmod +x script.sh
```

Execută din nou scriptul.

Nivel 2 – Aplicare

Execută același script folosind:

```
bash script.sh
```

și apoi:

```
./script.sh
```

Compară cele două metode și notează diferențele.

Nivel 3 – Administrator

Creează un script care afișează:

- data;
- utilizatorul curent;
- numele sistemului.

Testează-l folosind ambele metode de execuție și verifică permisiunile înainte și după `chmod +x`.

Soluția laboratorului

- Un script executat prin `./script.sh` necesită permisiunea de execuție.
 - Comanda `chmod +x` adaugă această permisiune.
 - Executarea prin `bash script.sh` nu necesită permisiunea de execuție, deoarece Bash citește direct fișierul.
 - Pentru distribuirea și reutilizarea scripturilor este recomandată folosirea unui Shebang și acordarea permisiunii de execuție.
-

Ce trebuie să reții

- Un script este un fișier text.
- Pentru executarea directă este necesară permisiunea de execuție.
- `chmod +x` acordă dreptul de execuție.
- `./script.sh` și `bash script.sh` sunt două moduri diferite de lansare a unui script.
- Un script bine pregătit are atât Shebang, cât și permisiunea de execuție.

Capitolul 8

8.4 Variabile Bash – Introducere

„O variabilă este un nume asociat unei valori. Ea permite stocarea și reutilizarea informațiilor pe parcursul execuției unui script.”

Ce vei învăța

În acest capitol vei înțelege ce este o variabilă Bash, cum este creată, cum este utilizată și de ce reprezintă unul dintre cele mai importante mecanisme din scripting.

Introducere

Până acum am executat comenzi care produceau rezultate.

Exemplu:

`date`

sau

hostname

Rezultatul era afișat pe ecran și apoi era pierdut.

Dar dacă dorim să folosim acel rezultat mai târziu în script?

Avem nevoie de un mecanism de memorare.

Acest mecanism este variabila.

Ce este o variabilă?

O variabilă este un identificator căruia îi este asociată o valoare.

Exemplu:

```
NUME="Daniel"
```

Aici:

- NUME este numele variabilei;
 - "Daniel" este valoarea ei.
-

Modelul mental

Nu te gândi la variabilă ca la o formulă matematică.

În Bash este mai util să o privești ca pe o etichetă.

```
NUME
```

```
|
```

```
▼
```

```
"Daniel"
```

Ori de câte ori Bash întâlnește:

```
$NUME
```

înlocuiește numele variabilei cu valoarea ei.

Acest proces poartă numele de **Parameter Expansion** și va fi studiat în detaliu într-un capitol ulterior.

Crearea unei variabile

Sintaxa este:

```
NUME="Daniel"
```

Observă că:

```
NUME = "Daniel"
```

este greșit.

Nu se folosesc spații în jurul caracterului:

```
=
```

Aceasta este una dintre cele mai frecvente greșeli ale începătorilor.

Utilizarea unei variabile

Executăm:

```
NUME="Daniel"
```

```
echo "$NUME"
```

Rezultatul:

```
Daniel
```

Semnul:

```
$
```

îi spune lui Bash:

Înlocuiește numele variabilei cu valoarea ei.

Modificarea valorii

O variabilă poate primi o nouă valoare.

Exemplu:

```
NUME="Daniel"
```

```
NUME="Maria"
```

```
echo "$NUME"
```

Rezultatul este:

Maria

Ultima atribuire este cea care rămâne valabilă.

Variabilele sunt sensibile la litere mari și mici

```
NUME="Daniel"
```

```
nume="Maria"
```

Acestea sunt două variabile diferite.

Bash face diferența între:

```
NUME
```

și

```
nume
```

Denumirea variabilelor

Un nume de variabilă:

- poate conține litere;
- poate conține cifre (dar nu ca prim caracter);
- poate conține caracterul `_`.

Exemple corecte:

```
NUME
```

```
utilizator
```

```
HOME_DIR
```

```
backup2026
```

Exemple incorecte:

```
2026backup
```

```
nume-utilizator
```

```
prenume.nume
```

Caseta „În spatele cortinei”

Atunci când executăm:

```
NUME="Daniel"
```

Bash memorează asocierea dintre nume și valoare în contextul shell-ului curent.

Variabila există atât timp cât shell-ul sau scriptul în care a fost creată este activ, dacă nu este exportată către procesele copil (subiect tratat într-un capitol ulterior).

Greșeală din lumea reală

Administratorul scrie:

```
NUME = "Daniel"
```

și primește:

```
bash: NUME: command not found
```

Bash nu interpretează această instrucțiune ca pe o atribuire.

El încearcă să execute o comandă numită:

```
NUME
```

Aceasta nu există.

Administrator Notes

Folosește nume descriptive pentru variabile.

De exemplu:

Mai puțin recomandat:

```
x="server01"
```

Mai clar:

```
SERVER="server01"
```

Un script bine denumit este mai ușor de citit și de întreținut.

Legătura cu LPIC

Pentru LPIC trebuie să cunoști:

- sintaxa atribuirii;
 - utilizarea operatorului \$;
 - regulile de denumire a variabilelor;
 - sensibilitatea la litere mari și mici.
-

Laborator 8.4

Nivel 1 – Înțelegere

Creează variabilele:

```
NUME="Linux"
```

```
CURS="LPIC-1"
```

Afișează valorile lor folosind:

```
echo "$NUME"
```

```
echo "$CURS"
```

Nivel 2 – Aplicare

Creează variabile pentru:

- utilizator;
- oraș;
- distribuția Linux preferată.

Afișează un mesaj folosind toate cele trei variabile.

Exemplu:

```
Salut, Daniel!
```

```
Folosești Ubuntu.
```

```
Locuiești în București.
```

Nivel 3 – Administrator

Scrie un script în care informațiile importante sunt păstrate în variabile:

- numele serverului;
- directorul de backup;
- fișierul jurnal;
- data curentă.

Afișează toate aceste informații într-un format clar.

În capitolele următoare aceste variabile vor fi folosite pentru automatizarea proceselor de backup și administrare.

Ce trebuie să reții

- O variabilă stochează o valoare asociată unui nume.
- Atribuirea se face fără spații în jurul operatorului =.
- Valoarea este accesată folosind caracterul \$.
- Variabilele sunt sensibile la litere mari și mici.
- Alegerea unor nume descriptive face scripturile mai ușor de înțeles și întreținut.

Capitolul 8

8.5 Variabile de mediu (Environment Variables)

„Pe lângă variabilele create într-un script, Bash primește și utilizează un set de variabile furnizate de sistemul de operare. Acestea formează mediul de execuție al procesului și influențează comportamentul programelor.”

Ce vei învăța

În acest capitol vei înțelege ce sunt variabilele de mediu, cum pot fi afișate și utilizate și care este diferența dintre o variabilă Bash și o variabilă de mediu.

Introducere

Până acum am creat variabile precum:

```
NUME="Daniel"
```

Noi am ales numele.

Noi am ales valoarea.

Dar execută:

```
echo "$HOME"
```

sau

```
echo "$USER"
```

Observăm că aceste variabile există deja.

Cine le-a creat?

Răspunsul este:

mediul în care rulează shell-ul.

Ce este o variabilă de mediu?

O variabilă de mediu este o variabilă disponibilă procesului curent și, de regulă, proceselor pe care acesta le lansează.

Aceste variabile conțin informații utile despre:

- utilizator;
 - sistem;
 - directoare importante;
 - limbă și localizare;
 - căutarea comenzilor;
 - configurarea aplicațiilor.
-

Exemple de variabile de mediu

Unele dintre cele mai utilizate sunt:

Variabilă	Rol
HOME	Directorul personal al utilizatorului
USER	Numele utilizatorului curent

Variabilă	Rol
PATH	Directoarele în care shell-ul caută comenzile
PWD	Directorul curent
SHELL	Shell-ul utilizatorului
LANG	Limba și localizarea sistemului
HOSTNAME	Numele sistemului (dacă este definit în mediu)

Afișarea unei variabile

Executăm:

```
echo "$HOME"
```

Rezultat posibil:

```
/home/student
```

Executăm:

```
echo "$PATH"
```

Rezultat:

```
/usr/local/bin:/usr/bin:/bin:...
```

Afișarea tuturor variabilelor de mediu

Putem utiliza:

```
env
```

sau

```
printenv
```

Ambele afișează mediul procesului curent.

Diferența dintre env și printenv

În practică, ambele sunt folosite pentru afișarea variabilelor de mediu.

Exemple:

```
env
```

```
printenv
```

Pentru o singură variabilă:

```
printenv HOME
```

Variabilă Bash versus variabilă de mediu

Executăm:

```
NUME="Daniel"
```

Apoi:

```
env
```

Variabila:

```
NUME
```

nu apare.

De ce?

Pentru că este o variabilă Bash.

Nu face parte din mediul procesului.

Comanda export

Pentru a transforma o variabilă Bash într-o variabilă de mediu folosim:

```
export NUME
```

sau direct:

```
export NUME="Daniel"
```

Acum:

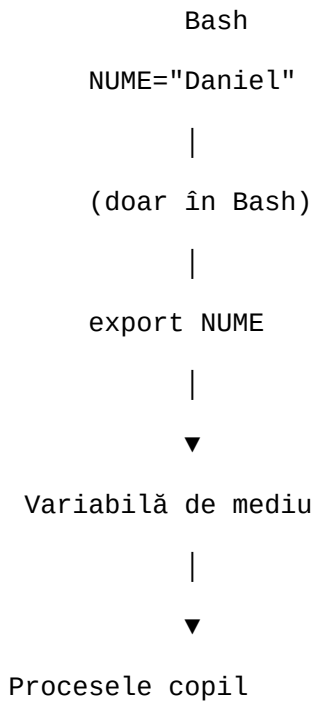
```
printenv NUME
```

afișează:

```
Daniel
```

Variabila face parte din mediul procesului.

Modelul mental



Caseta „În spatele cortinei”

Atunci când Bash lansează un proces nou, acesta primește o copie a mediului de execuție.

Doar variabilele exportate sunt incluse în această copie.

Variabilele Bash care nu au fost exportate rămân disponibile doar în shell-ul curent.

Greșeală din lumea reală

Administratorul execută:

```
DIRECTOR="/backup"
```

Apoi lansează un script care încearcă să utilizeze:

```
$DIRECTOR
```

Scriptul nu găsește variabila.

Motivul:

Variabila nu a fost exportată.

Soluția:

```
export DIRECTOR
```

Administrator Notes

Nu exporta toate variabilele.

Exportă doar variabilele care trebuie să fie disponibile și proceselor lansate de shell.

Acest lucru păstrează mediul de execuție mai clar și mai ușor de administrat.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- diferența dintre variabile Bash și variabile de mediu;
 - utilizarea comenzilor `env` și `printenv`;
 - rolul comenzii `export`;
 - câteva variabile de mediu importante (HOME, PATH, USER, PWD, LANG).
-

Laborator 8.5

Nivel 1 – Înțelegere

Afișează valorile variabilelor:

```
echo "$HOME"
```

```
echo "$USER"
```

```
echo "$PATH"
```

```
echo "$PWD"
```

Execută apoi:

```
env
```

și identifică aceste variabile în rezultat.

Nivel 2 – Aplicare

Creează o variabilă:

```
NUME="LPIC"
```

Verifică dacă apare în:

```
printenv
```

Export-o:

```
export NUME
```

Verifică din nou folosind:

```
printenv NUME
```

Observă diferența.

Nivel 3 – Administrator

Scrie un script care afișează:

- utilizatorul curent;
- directorul HOME;
- directorul curent;
- valoarea variabilei PATH.

Adaugă apoi o variabilă proprie:

```
export BACKUP_DIR="/backup"
```

și afișează valoarea ei în cadrul aceluiași script.

Soluția laboratorului

- Variabilele de mediu există deja în mediul procesului.
 - `env` și `printenv` afișează variabilele de mediu.
 - O variabilă Bash devine variabilă de mediu după utilizarea comenzii `export`.
 - Doar variabilele exportate sunt disponibile proceselor copil.
-

Ce trebuie să reții

- Variabilele de mediu descriu mediul în care rulează un proces.
- `env` și `printenv` afișează aceste variabile.
- `export` face o variabilă disponibilă și proceselor lansate ulterior.
- Nu toate variabilele Bash sunt variabile de mediu.
- HOME, PATH, USER și PWD sunt printre cele mai utilizate variabile de mediu.

Capitolul 8

8.6 Command Substitution – Cum folosim rezultatul unei comenzi

„Până acum am executat comenzi pentru a afișa informații pe ecran. În multe situații însă, rezultatul unei comenzi trebuie reutilizat într-un script. Command Substitution permite capturarea rezultatului unei comenzi și utilizarea acestuia ca valoare.”

Ce vei învăța

În acest capitol vei înțelege ce este **Command Substitution**, cum funcționează și cum poate fi utilizat pentru a salva rezultatul unei comenzi într-o variabilă sau într-o altă expresie Bash.

Introducere

Executăm:

```
date
```

Rezultatul este afișat pe ecran.

Dar dacă dorim să folosim această dată într-un nume de fișier?

De exemplu:

```
backup-2026-08-04.tar.gz
```

Nu este suficient ca data să fie afișată.

Trebuie să o memorăm.

Acesta este rolul **Command Substitution**.

Ideea fundamentală

Command Substitution înseamnă:

Execută o comandă și înlocuiește expresia cu rezultatul produs de acea comandă.

Sintaxa recomandată

Forma modernă este:

```
$(comanda)
```

Exemplu:

```
DATA=$(date)
```

Acum variabila:

```
$DATA
```

conține rezultatul comenzii `date`.

Primul exemplu

```
UTILIZATOR=$(whoami)
```

```
echo "$UTILIZATOR"
```

Rezultatul:

```
student
```

Nu am introdus manual această valoare.

Ea a fost obținută prin executarea unei comenzi.

Modelul mental

whoami

|



rezultat produs

|



Command Substitution

|



variabila UTILIZATOR

Observă că Bash execută mai întâi comanda.

Abia apoi atribuie rezultatul variabilei.

Un exemplu practic

Dorim să construim un nume de fișier care conține data curentă.

```
RAPORT="raport-$(date +%F).txt"
```

```
echo "$RAPORT"
```

Rezultat:

```
raport-2026-08-04.txt
```

Aceasta este una dintre cele mai frecvente utilizări ale Command Substitution.

Utilizarea directă în comenzi

Nu este obligatoriu să folosim o variabilă.

Exemplu:

```
echo "Astăzi este $(date +%A)"
```

sau:

```
mkdir backup-$(date +%Y%m%d)
```

Rezultatul comenzii este inserat direct în argumentele altei comenzi.

Comenzi multiple

Putem combina mai multe substituții:

```
echo "Utilizator: $(whoami)"  
echo "Director: $(pwd)"  
echo "Sistem: $(hostname)"
```

Fiecare comandă este executată separat, iar rezultatul este inserat în textul final.

Sintaxa veche

În documentația mai veche putem întâlni:

```
`date`
```

(backticks)

Aceasta funcționează și astăzi.

Totuși, forma recomandată este:

```
$(date)
```

deoarece:

- este mai ușor de citit;
 - permite imbricarea comenzilor;
 - este mai clară în expresiile complexe.
-

Caseta „În spatele cortinei”

Bash execută comanda dintre paranteze și capturează ceea ce aceasta scrie pe **stdout**.

Rezultatul este apoi introdus în locul expresiei:

```
$(...)
```

Observă că este capturat **stdout**.

Dacă o comandă afișează doar mesaje de eroare pe **stderr**, acestea nu devin automat rezultatul substituției.

Greșeală din lumea reală

Administratorul scrie:

```
DATA=date
```

și se așteaptă ca variabila să conțină data curentă.

Rezultatul este:

```
date
```

Bash nu execută comanda.

El atribuie textul literal.

Forma corectă este:

DATA=\$(date)

Administrator Notes

Folosește Command Substitution atunci când valoarea unei variabile trebuie obținută din rezultatul unei comenzi.

Exemple frecvente:

- data curentă;
 - utilizatorul curent;
 - directorul curent;
 - numele sistemului;
 - lista fișierelor;
 - rezultatul unei căutări.
-

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- sintaxa \$(. . .);
 - diferența față de backticks;
 - utilizarea Command Substitution în variabile și în argumentele comenzilor.
-

Laborator 8.6

Nivel 1 – Înțelegere

Creează variabilele:

```
UTILIZATOR=$(whoami)
```

```
DIRECTOR=$(pwd)
```

```
SISTEM=$(hostname)
```

Afișează valorile folosind:

```
echo "$UTILIZATOR"
```

```
echo "$DIRECTOR"
```

```
echo "$SISTEM"
```

Nivel 2 – Aplicare

Construiește un nume de fișier care conține data curentă:

```
backup-$(date +%F).tar.gz
```

Afișează rezultatul și explică modul în care a fost construit.

Nivel 3 – Administrator

Scrie un script care afișează un raport de sistem în forma:

```
=====
Raport sistem
=====

Data: ...
Utilizator: ...
Hostname: ...
Director curent: ...

=====
```

Toate informațiile trebuie obținute prin **Command Substitution**.

Soluția laboratorului

- `$(...)` execută o comandă și capturează rezultatul de pe `stdout`.
 - Rezultatul poate fi atribuit unei variabile sau utilizat direct într-o altă comandă.
 - Forma modernă `$(...)` este recomandată în locul sintaxei cu backticks.
-

Ce trebuie să reții

- Command Substitution execută o comandă și utilizează rezultatul ei.
- Sintaxa recomandată este `$(...)`.
- Rezultatul provine din `stdout`.
- Poate fi folosit atât în atribuiri, cât și direct în argumentele comenzilor.
- Este una dintre cele mai utilizate tehnici în scripting Bash.

Capitolul 8

8.7 Parametrii speciali Bash (Special Parameters)

„Bash pune la dispoziția scripturilor un set de parametri speciali al căror conținut este gestionat automat de shell. Aceștia oferă informații despre execuția scriptului, argumentele primite și procesele implicate.”

Ce vei învăța

În acest capitol vei înțelege rolul parametrilor speciali ai shell-ului și modul în care aceștia sunt utilizați în scripting.

Introducere

Până acum am creat propriile noastre variabile:

```
NUME="Daniel"
```

și am utilizat variabile de mediu precum:

```
$HOME  
$PATH  
$USER
```

Aceste variabile au o caracteristică comună.

Ele pot fi create sau modificate de utilizator.

Există însă o categorie diferită.

Parametri pe care Bash îi creează și îi actualizează automat.

Aceștia se numesc **Special Parameters**.

Ce sunt parametrii speciali?

Parametrii speciali sunt valori gestionate intern de Bash.

Ei nu sunt creați prin atribuiri și, în general, nu pot fi modificați de utilizator.

Exemple:

\$?
\$\$
\$!
\$#
\$*
\$@
\$0
\$1
\$2
...

Fiecare are un rol bine definit.

În subcapitolele următoare îi vom studia individual.

De ce sunt importanți?

Un script trebuie să poată răspunde la întrebări precum:

- Care a fost rezultatul ultimei comenzi?
- Câte argumente am primit?
- Care este numele scriptului?
- Care este PID-ul shell-ului curent?
- Care este PID-ul ultimului proces lansat în fundal?

Toate aceste informații sunt furnizate de parametri speciali.

Clasificarea parametrilor speciali

În scop didactic îi vom grupa în patru categorii.

Parametri privind execuția

\$?
\$\$
\$!

Parametri privind argumentele

\$#
\$*
\$@
\$1
\$2
...

Parametrul care identifică scriptul

\$0

Parametri avansați

Aceștia vor fi prezentați în capitolele dedicate funcțiilor și shell-ului interactiv.

Administrator Notes

Nu încerca să memorezi toți parametrii speciali de la început.

Este mai eficient să îi înveți pe măsură ce apar în exemple și scripturi.

În capitolele următoare fiecare parametru va fi studiat separat și utilizat în contexte reale.

Legătura cu LPIC

Parametrii speciali reprezintă un obiectiv important al examenului LPIC-1.

Înțelegerea lor este esențială pentru scrierea scripturilor Bash și pentru interpretarea exemplelor din documentația oficială.

Laborator 8.7

Nivel 1 – Înțelegere

Execută:

```
echo "$$"
echo "$?"
```

Observă că Bash afișează valori fără ca acestea să fi fost definite de utilizator.

Nivel 2 – Aplicare

Execută un script simplu și observă valorile parametrilor speciali pe parcursul execuției.

(Valorile individuale vor fi analizate în capitolele următoare.)

Ce trebuie să reții

- Parametrii speciali sunt gestionați automat de Bash.
- Ei oferă informații despre execuția shell-ului și a scripturilor.
- Nu sunt definiți prin atribuiri obișnuite.
- Fiecare parametru are un rol precis.

Capitolul 8

8.7.1 Parametrul special \$? – Codul de ieșire al ultimei comenzi

„Fiecare comandă executată în Bash se încheie cu un cod de ieșire. Acest cod indică dacă execuția s-a încheiat cu succes sau dacă a apărut o eroare. Parametrul special \$? conține întotdeauna codul de ieșire al ultimei comenzi executate.”

Ce vei învăța

În acest capitol vei înțelege ce reprezintă codul de ieșire al unei comenzi, cum este accesat prin parametrul special \$? și de ce acesta stă la baza deciziilor din scripturile Bash.

Introducere

Până acum am executat numeroase comenzi:

ls

pwd

date

Observam doar rezultatul afișat pe ecran.

Dar fiecare comandă produce încă o informație.

Un **cod de ieșire** (*Exit Status*).

Acest cod este memorat de Bash în parametrul special:

\$?

Ideea fundamentală

După terminarea fiecărei comenzi, Bash memorează codul de ieșire al acesteia.

Acest cod poate fi afișat imediat folosind:

```
echo $?
```

Primul exemplu

Executăm:

```
pwd
```

Apoi:

```
echo $?
```

Rezultat:

```
0
```

Valoarea:

```
0
```

Înseamnă că executarea comenzii s-a încheiat cu succes.

Un exemplu de eroare

Executăm:

```
ls fisier_inexistent
```

Rezultatul:

```
ls: cannot access 'fisier_inexistent': No such file or directory
```

Apoi executăm:

```
echo $?
```

Rezultat posibil:

```
2
```


Comanda nu s-a încheiat cu succes.

Bash memorează acest cod.

Modelul mental

Comandă

|



Execuție

|



Exit Status

|



Bash

|



\$?

Ce înseamnă valoarea 0?

În Bash și în majoritatea sistemelor Unix:

0

înseamnă:

Succes.

Acesta este unul dintre primele lucruri pe care trebuie să le accepte un cursant.

În multe limbaje de programare:

0

înseamnă *False*.

În Bash:

pentru codurile de ieşire:

0

înseamnă:

operaţia a reuşit.

Valorile diferite de 0

Orice valoare diferită de:

0

indică faptul că programul semnalează o situaţie de eroare sau o condiţie specială.

Exemple:

1

2

126

127

Semnificaţia exactă depinde de programul executat.

Vom studia codurile de ieşire într-un capitol dedicat.

Atenţie!

Parametrul:

\$?

este actualizat după **fiecare comandă**.

Exemplu:

ls

echo \$?

Aici \$? conţine codul de ieşire al lui ls.

Dar dacă executăm:

ls

date

```
echo $?
```

acum `$?` conține codul de ieșire al comenzii:

```
date
```

nu al lui `ls`.

Acesta este unul dintre cele mai frecvente motive de confuzie în scripting.

Caseta „În spatele cortinei”

Bash nu calculează codul de ieșire.

Programul executat îl întoarce către sistemul de operare atunci când își încheie execuția.

Bash îl preia și îl pune la dispoziția scriptului prin parametrul special:

```
$?
```

Greșeală din lumea reală

Administratorul execută:

```
cp fisier1 backup/
```

```
echo "Backup terminat"
```

```
echo $?
```

Observă valoarea:

```
0
```

și concluzionează că operația de copiere a reușit.

În realitate, `$?` reprezintă codul de ieșire al ultimei comenzii:

```
echo "Backup terminat"
```

nu al comenzii:

```
cp
```

Dacă dorim să verificăm rezultatul lui `cp`, trebuie să consultăm `$?` imediat după executarea acesteia.

Administrator Notes

Verifică întotdeauna \$? imediat după comanda al cărei rezultat te interesează.

Orice comandă executată între timp va înlocui valoarea memorată.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul parametrului special \$?;
 - faptul că valoarea 0 indică succesul;
 - faptul că orice altă valoare indică o condiție diferită de succes;
 - faptul că Bash actualizează \$? după fiecare comandă executată.
-

Laborator 8.7.1

Nivel 1 – Înțelegere

Execută:

```
pwd
```

```
echo $?
```

Apoi:

```
ls fisier_inexistent
```

```
echo $?
```

Compară rezultatele.

Nivel 2 – Aplicare

Execută:

```
mkdir test_lpic
```

```
echo $?
```

```
rmdir test_lpic
```

```
echo $?
```

Observă codurile de ieșire.

Repetă exercițiul încercând să ștergi un director care nu există.

Nivel 3 – Administrator

Scrie un script care:

1. creează un director;
2. verifică imediat codul de ieșire;
3. afișează un mesaj dacă operația a reușit;
4. afișează un alt mesaj dacă operația a eșuat.

(În capitolul despre `if` vom automatiza această verificare folosind structuri de control.)

Soluția laboratorului

- `$?` conține codul de ieșire al ultimei comenzi executate.
 - Valoarea `0` indică succesul execuției.
 - Orice comandă nouă modifică valoarea lui `$?`.
 - Pentru verificarea unei operații, `$?` trebuie consultat imediat după executarea acelei comenzi.
-

Ce trebuie să reții

- `$?` este parametrul special care conține codul de ieșire al ultimei comenzi.
- Valoarea `0` înseamnă succes.
- Valorile diferite de `0` indică o condiție diferită de succes.
- Bash actualizează `$?` după fiecare comandă.
- Verifică întotdeauna `$?` imediat după comanda care te interesează.

Capitolul 8

8.7.2 Parametrul special \$\$ – Identificatorul procesului shell-ului curent

„Parametrul special \$\$ conține identificatorul procesului (Process ID - PID) shell-ului curent. Acesta este atribuit de sistemul de operare și identifică în mod unic procesul Bash aflat în execuție.”

Ce vei învăța

În acest capitol vei înțelege ce reprezintă parametrul special \$\$, ce este un PID și în ce situații este utilizat în scripturile Bash.

Introducere

Fiecare proces care rulează într-un sistem Linux primește un număr unic.

Acest număr se numește:

Process ID (PID).

Bash nu face excepție.

Și el rulează ca un proces.

Parametrul special:

\$\$

conține PID-ul shell-ului curent.

Primul exemplu

Executăm:

```
echo $$
```

Rezultat posibil:

```
25431
```

Acesta este PID-ul procesului Bash în care executăm comanda.

Verificarea rezultatului

Executăm:

```
echo $$
```

Apoi:

```
ps -p $$
```

Rezultat posibil:

PID	TTY	TIME	CMD
25431	pts/0	00:00:00	bash

Observăm că valoarea lui \$\$ corespunde PID-ului procesului Bash.

Modelul mental

Kernel Linux

|



creează procesul Bash

|



PID = 25431

|



\$\$ = 25431

Utilizare practică

Un exemplu frecvent este generarea unui nume temporar de fișier.

```
TEMP="backup.$$"
```

Rezultat posibil:

```
backup.25431
```

Dacă două scripturi sunt executate simultan, fiecare va avea un PID diferit, reducând riscul ca ambele să încerce să folosească același fișier temporar.

Un alt exemplu

```
LOG="/tmp/script-$$ .log"
```

Rezultat posibil:

```
/tmp/script-25431.log
```

Această tehnică este întâlnită frecvent în scripturile de administrare.

Caseta „În spatele cortinei”

PID-ul nu este ales de Bash.

El este atribuit de kernel atunci când procesul este creat.

Bash doar pune această valoare la dispoziția scriptului prin parametrul special:

```
$$
```

Greșeală din lumea reală

Un administrator presupune că:

```
$$
```

reprezintă PID-ul ultimei comenzi executate.

Nu este adevărat.

\$\$ reprezintă întotdeauna PID-ul shell-ului curent.

PID-ul altor procese poate fi diferit.

Administrator Notes

Folosește \$\$ atunci când ai nevoie de un identificator unic pentru:

- fișiere temporare;
- fișiere jurnal;

- directoare de lucru temporare.

Nu utiliza valori fixe atunci când există posibilitatea executării simultane a mai multor instanțe ale aceluiași script.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- semnificația parametrului special \$\$;
 - faptul că reprezintă PID-ul shell-ului curent;
 - utilizarea lui în scripturi.
-

Laborator 8.7.2

Nivel 1 – Înțelegere

Execută:

```
echo $$
```

Apoi:

```
ps -p $$
```

Compară rezultatele.

Nivel 2 – Aplicare

Construiește un nume de fișier folosind PID-ul:

```
echo "backup-$$ .log"
```

Execută comanda în două terminale diferite și observă diferența.

Nivel 3 – Administrator

Scrie un script care creează un fișier jurnal folosind PID-ul shell-ului:

```
/var/tmp/backup-<PID>.log
```

Afișează numele fișierului creat.

Soluția laboratorului

- \$\$ conține PID-ul shell-ului curent.
 - PID-ul este atribuit de kernel.
 - Fiecare proces Bash are propriul PID.
 - \$\$ este utilizat frecvent pentru generarea numelor unice de fișiere temporare și jurnale.
-

Ce trebuie să reții

- \$\$ reprezintă PID-ul shell-ului curent.
- Valoarea este atribuită de kernel.
- Fiecare proces Bash are un PID diferit.
- \$\$ este util pentru construirea unor nume unice de fișiere în scripturi.

Capitolul 8

8.7.3 Parametrul special \$! – PID-ul ultimului proces executat în fundal

„Parametrul special \$! conține identificatorul procesului (PID) al ultimei comenzi lansate în fundal de shell-ul curent.”

Ce vei învăța

În acest capitol vei înțelege ce reprezintă parametrul special \$!, când primește o valoare și în ce situații este utilizat în scripturile Bash.

Introducere

În capitolul anterior am studiat:

\$\$

care reprezintă PID-ul shell-ului curent.

Acum apare o întrebare firească.

Dacă un script pornește un alt proces în fundal, cum poate afla PID-ul acelui proces?

Răspunsul este parametrul special:

\$!

Ideea fundamentală

După lansarea unei comenzi în fundal folosind operatorul:

&

Bash memorează PID-ul acelui proces în:

\$!

Primul exemplu

Executăm:

```
sleep 60 &
```

Shell-ul poate afișa:

```
[1] 18452
```

Acum executăm:

```
echo $!
```

Rezultat:

```
18452
```

Valoarea reprezintă PID-ul procesului:

```
sleep
```

Verificarea rezultatului

Executăm:

```
sleep 60 &
```

Apoi:

```
ps -p $!
```

Rezultat posibil:

PID	TTY	TIME	CMD
18452	pts/0	00:00:00	sleep

Observăm că \$! conține PID-ul procesului lansat în fundal.

Modelul mental

Bash

```
|
|
|— sleep 60 &
|
▼
```

Kernel

```
|
▼
```

PID = 18452

```
|
▼
```

\$! = 18452

Când se modifică?

Parametrul:

\$!

este actualizat numai atunci când Bash lansează un nou proces în fundal.

Dacă executăm o comandă normală:

date

valoarea lui:

\$!

nu se modifică.

Utilizare practică

Un script pornește un proces lung:

```
tar czf backup.tar.gz /home &
```

Imediat după aceea poate salva PID-ul:

```
PID_BACKUP=$!
```

```
echo "Procesul rulează cu PID-ul $PID_BACKUP"
```

Mai târziu, același PID poate fi utilizat pentru monitorizare sau pentru a aștepta terminarea procesului.

Caseta „În spatele cortinei”

Bash actualizează parametrul:

```
$!
```

imediat după lansarea unei comenzi în fundal.

Valoarea rămâne disponibilă până când un alt proces este lansat în fundal.

Greșeală din lumea reală

Administratorul execută:

```
sleep 60
```

Apoi:

```
echo $!
```

și se așteaptă să obțină PID-ul procesului `sleep`.

Nu funcționează.

Comanda nu a fost lansată în fundal.

Operatorul:

```
&
```

lipsește.

Administrator Notes

\$! este folosit frecvent împreună cu:

- procese executate în fundal;
 - comanda `wait`;
 - monitorizarea execuției unor procese lungi;
 - automatizări în care sunt lansate mai multe procese simultan.
-

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- semnificația parametrului special \$!;
 - faptul că reprezintă PID-ul ultimului proces lansat în fundal;
 - relația dintre \$! și operatorul &.
-

Laborator 8.7.3

Nivel 1 – Înțelegere

Execută:

```
sleep 30 &
```

Apoi:

```
echo $!
```

Verifică procesul:

```
ps -p $!
```

Nivel 2 – Aplicare

Lansează două procese:

```
sleep 60 &  
sleep 120 &
```

După fiecare comandă afișează:

```
echo $!
```

Observă cum Bash actualizează parametrul special.

Nivel 3 – Administrator

Scrie un script care:

1. pornește un proces în fundal;
 2. memorează PID-ul folosind \$!;
 3. afișează PID-ul procesului;
 4. verifică existența procesului folosind `ps -p`.
-

Soluția laboratorului

- \$! conține PID-ul ultimului proces lansat în fundal.
 - Valoarea este disponibilă imediat după utilizarea operatorului &.
 - Fiecare nou proces lansat în fundal înlocuiește valoarea anterioară.
 - \$! este utilizat frecvent împreună cu `wait` și cu comenzile de monitorizare a proceselor.
-

Ce trebuie să reții

- \$! reprezintă PID-ul ultimului proces executat în fundal.
- Funcționează împreună cu operatorul &.
- Valoarea este actualizată automat de Bash.
- Este util pentru monitorizarea și controlul proceselor lansate din scripturi.

Capitolul 8

8.7.4 Parametrul special \$0 – Numele scriptului sau al shell-ului

„Parametrul special \$0 conține numele utilizat pentru invocarea shell-ului sau a scriptului. Într-un script Bash, acesta reprezintă, de regulă, numele sau calea prin care scriptul a fost lansat.”

Ce vei învăța

În acest capitol vei înțelege ce reprezintă parametrul special `$0`, cum este stabilită valoarea lui și în ce situații este utilizat în scripturile Bash.

Introducere

Până acum am studiat:

- `$?` – codul de ieșire al ultimei comenzi;
- `$$` – PID-ul shell-ului curent;
- `#!` – PID-ul ultimului proces executat în fundal.

Acum apare o întrebare.

Cum poate un script să afle cum se numește el însuși?

Răspunsul este parametrul special:

`$0`

Ideea fundamentală

`$0` conține numele utilizat la lansarea scriptului sau a shell-ului.

Primul exemplu

Fișierul:

```
#!/bin/bash
```

```
echo "$0"
```

Dacă este executat astfel:

```
./backup.sh
```

rezultatul va fi:

```
./backup.sh
```

Un alt exemplu

Dacă același script este executat astfel:

```
/home/student/scripts/backup.sh
```

rezultatul poate fi:

```
/home/student/scripts/backup.sh
```

Observăm că \$0 păstrează forma utilizată la lansarea scriptului.

Modelul mental

Utilizator

|

▼

```
./backup.sh
```

|

▼

Bash

|

▼

```
$0 = "./backup.sh"
```

Utilizare practică

Deseori un script afișează mesaje de utilizare.

Exemplu:

```
echo "Utilizare: $0 <fișier>"
```

Rezultat:

```
Utilizare: ./backup.sh <fișier>
```

Avantajul este evident.

Dacă scriptul este redenumit:

```
backup.sh
```

↓

```
backup_zilnic.sh
```

mesajul se actualizează automat.

Nu trebuie modificat manual.

Un alt exemplu

```
echo "Script executat: $0"
```

Acest tip de mesaj apare frecvent în fișierele jurnal.

Caseta „În spatele cortinei”

Valoarea lui \$0 este stabilită în momentul lansării scriptului.

Bash nu încearcă să determine numele „real” al fișierului.

El utilizează argumentul prin care scriptul a fost invocat.

Din acest motiv, aceeași aplicație poate avea valori diferite pentru \$0, în funcție de modul în care este lansată.

Greșeală din lumea reală

Administratorul presupune că:

```
$0
```

conține întotdeauna doar numele fișierului.

Nu este adevărat.

Dacă scriptul este lansat folosind o cale completă sau relativă, această cale poate apărea și în valoarea lui \$0.

Administrator Notes

\$0 este utilizat frecvent pentru:

- afișarea mesajelor de utilizare;

- identificarea scriptului în fișiere jurnal;
- generarea mesajelor de eroare.

Nu îl utiliza pentru a determina directorul în care se află scriptul. Vom prezenta această tehnică într-un capitol ulterior.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- semnificația parametrului special \$0;
 - faptul că reprezintă numele utilizat la lansarea scriptului;
 - utilizarea lui în mesajele de utilizare și în jurnalizare.
-

Laborator 8.7.4

Nivel 1 – Înțelegere

Creează scriptul:

```
#!/bin/bash
```

```
echo "$0"
```

Execută-l folosind:

```
./script.sh
```

Observă rezultatul.

Nivel 2 – Aplicare

Execută același script folosind o cale diferită:

```
/home/student/script.sh
```

Compară valoarea afișată.

Nivel 3 – Administrator

Scrie un script care afișează:

```
=====
```

```
Script: ...  
Data: ...  
Utilizator: ...  
=====
```

Folosește:

- \$0;
- \$(date);
- \$(whoami).

Soluția laboratorului

- \$0 conține numele utilizat la lansarea scriptului.
- Valoarea poate fi un nume simplu, o cale relativă sau o cale absolută.
- \$0 este utilizat frecvent pentru afișarea mesajelor de utilizare și pentru jurnalizare.

Ce trebuie să reții

- \$0 reprezintă numele sau calea utilizată pentru lansarea scriptului.
- Valoarea este stabilită la pornirea scriptului.
- Este utilizat frecvent în mesajele informative și în documentarea utilizării unui script.
- Nu trebuie confundat cu directorul în care se află scriptul.

Capitolul 8

8.8 Parametrii poziționali (Positional Parameters)

„Atunci când un script este executat, utilizatorul poate transmite informații suplimentare sub forma argumentelor din linia de comandă. Bash pune aceste argumente la dispoziția scriptului prin intermediul parametrilor poziționali.”

Ce vei învăța

În acest capitol vei înțelege ce sunt parametrii poziționali, cum sunt transmise argumentele unui script și cum pot fi accesate în Bash.

Introducere

Până acum scripturile noastre executau întotdeauna aceleași operații.

Dar în practică apare rapid o întrebare.

Cum îi transmitem unui script informații diferite la fiecare execuție?

Exemplu:

```
backup.sh documente
```

sau

```
backup.sh /home/student
```

sau

```
backup.sh raport.txt
```

Observăm că după numele scriptului apar valori suplimentare.

Acestea se numesc **argumente**.

Ideea fundamentală

Bash preia argumentele introduse după numele scriptului și le asociază automat unor parametri speciali.

Primul argument:

```
$1
```

Al doilea argument:

```
$2
```

Al treilea argument:

```
$3
```

și așa mai departe.

Primul exemplu

Script:

```
#!/bin/bash
```

```
echo "Primul argument: $1"
```

```
echo "Al doilea argument: $2"
```

Execuție:

```
./script.sh Linux Bash
```

Rezultat:

```
Primul argument: Linux  
Al doilea argument: Bash
```

Modelul mental

```
./script.sh Linux Bash LPIC
```

|



```
$0 → ./script.sh
```

```
$1 → Linux
```

```
$2 → Bash
```

```
$3 → LPIC
```

Observă că:

- \$0 reprezintă scriptul;
 - argumentele încep întotdeauna cu \$1.
-

Dacă lipsesc argumentele?

Executăm:

```
./script.sh
```

Rezultat:

```
Primul argument:
```

```
Al doilea argument:
```

Parametrii există, dar nu au valoare.

Acest comportament este normal.

Argumentele sunt șiruri de caractere

Pentru Bash nu există diferență între:

123

și

Linux

Ambele sunt tratate ca șiruri de caractere.

Interpretarea lor depinde de script.

Caseta „În spatele cortinei”

În momentul lansării scriptului, Bash construiește lista argumentelor și o asociază parametrilor poziționali.

Scriptul nu trebuie să citească manual linia de comandă.

Accesează direct valorile prin:

\$1

\$2

\$3

...

Greșeală din lumea reală

Administratorul execută:

```
./backup.sh /home date
```

și presupune că:

```
$0 = /home
```

Nu.

\$0 este întotdeauna numele utilizat pentru lansarea scriptului.

Primul argument este:

```
$1
```

Administrator Notes

Un script bine proiectat nu presupune că argumentele există.

În capitolul despre structura `if` vom învăța cum verificăm dacă utilizatorul a furnizat argumentele necesare înainte de continuarea execuției.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul parametrilor poziționali;
 - corespondența dintre argumente și `$1`, `$2`, `$3` etc.;
 - relația dintre `$0` și argumentele scriptului.
-

Laborator 8.8

Nivel 1 – Înțelegere

Creează scriptul:

```
#!/bin/bash

echo "Script: $0"
echo "Argument 1: $1"
echo "Argument 2: $2"
```

Execută:

```
./script.sh Linux Bash
```

Analizează rezultatul.

Nivel 2 – Aplicare

Execută același script de mai multe ori:

```
./script.sh Bucuresti

./script.sh Ubuntu Debian

./script.sh LPIC Linux Bash Script
```

Observă cum se modifică valorile parametrilor poziționali.

Nivel 3 – Administrator

Realizează un script care primește:

- numele unui server;
- numele unui utilizator.

Scriptul va afișa:

```
=====
Raport conexiune

Server: ...
Utilizator: ...

=====
```

Valorile trebuie preluate din \$1 și \$2.

Soluția laboratorului

- Argumentele sunt transmise după numele scriptului.
 - Bash asociază automat primul argument lui \$1, al doilea lui \$2 și așa mai departe.
 - \$0 reprezintă numele scriptului și nu face parte din lista argumentelor.
-

Ce trebuie să reții

- Argumentele sunt transmise la lansarea scriptului.
- \$1, \$2, \$3... reprezintă parametrii poziționali.
- \$0 reprezintă numele scriptului.
- Dacă un argument nu este furnizat, parametrul corespunzător este gol.

Capitolul 8

8.8.1 Parametrul special \$# – Numărul argumentelor primite de un script

„Parametrul special \$# conține numărul argumentelor poziționale transmise unui script sau unei funcții.”

Ce vei învăța

În acest capitol vei înțelege rolul parametrului special `$#`, modul în care este calculată valoarea sa și utilizarea acestuia pentru verificarea argumentelor primite de un script.

Introducere

În capitolul anterior am învățat că argumentele sunt accesate prin:

```
$1  
$2  
$3  
...
```

Dar apare imediat o întrebare.

Cum știe scriptul câte argumente a primit?

Răspunsul este parametrul special:

```
$#
```

Ideea fundamentală

`$#` conține numărul parametrilor poziționali primiți de script.

Nu conține argumentele.

Conține doar **numărul lor**.

Primul exemplu

Script:

```
#!/bin/bash  
  
echo "Număr argumente: $#"
```

Execuție:

```
./script.sh Linux Bash LPIC
```

Rezultat:

```
Număr argumente: 3
```

Un alt exemplu

Executăm:

```
./script.sh
```

Rezultat:

Număr argumente: 0

Scriptul nu a primit niciun argument.

Modelul mental

```
./script.sh Linux Bash LPIC
```

|



\$1 → Linux

\$2 → Bash

\$3 → LPIC

\$# → 3

Observă că \$# nu reprezintă un argument.

El reprezintă **numărul argumentelor**.

Utilizare practică

Foarte multe scripturi verifică dacă utilizatorul a furnizat suficiente argumente.

Exemplu:

```
#!/bin/bash
```

```
echo "Scriptul a primit $# argument(e)."
```

Mai târziu, în capitolul despre `if`, vom utiliza această informație pentru validarea liniei de comandă.

Caseta „În spatele cortinei”

În momentul lansării scriptului, Bash construiește lista parametrilor poziționali.

În același timp calculează și numărul lor.

Această valoare este disponibilă permanent prin:

```
$#
```

Greșeală din lumea reală

Administratorul execută:

```
./backup.sh server1
```

și presupune că:

```
$#
```

va avea valoarea:

```
2
```

deoarece există și numele scriptului.

Nu.

Numele scriptului este:

```
$0
```

și **nu este inclus** în numărul argumentelor.

În acest exemplu:

```
$# = 1
```

Administrator Notes

Verificarea numărului de argumente este una dintre primele operații pe care ar trebui să le execute un script.

Acest lucru permite detectarea rapidă a unei utilizări incorecte.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- semnificația parametrului special \$#;
 - faptul că reprezintă numărul argumentelor poziționale;
 - diferența dintre \$# și \$0.
-

Laborator 8.8.1

Nivel 1 – Înțelegere

Creează scriptul:

```
#!/bin/bash  
  
echo "Argumente primite: $#"
```

Execută-l astfel:

```
./script.sh
```

Apoi:

```
./script.sh Linux
```

Și:

```
./script.sh Linux Bash LPIC
```

Compară rezultatele.

Nivel 2 – Aplicare

Extinde scriptul:

```
#!/bin/bash  
  
echo "Script: $0"  
echo "Argumente: $#"  
echo "Primul argument: $1"
```

Execută-l folosind un număr diferit de argumente.

Observă relația dintre \$# și \$1.

Nivel 3 – Administrator

Scrive un script care afișează:

```
=====
Raport execuție

Script: ...
Argumente primite: ...

=====
```

Folosește:

- \$0
- \$#

În capitolul despre `if`, scriptul va verifica dacă numărul argumentelor este cel așteptat.

Soluția laboratorului

- \$# conține numărul argumentelor poziționale.
 - Numele scriptului (\$0) nu este inclus în acest număr.
 - Valoarea este calculată automat de Bash la lansarea scriptului.
-

Ce trebuie să reții

- \$# reprezintă numărul argumentelor primite de script.
- Nu include \$0.
- Este utilizat frecvent pentru validarea liniei de comandă.
- Valoarea este actualizată automat de Bash.

Capitolul 8

8.8.2 Parametrii speciali \$* și @\$

„Parametrii speciali \$ și @\$ reprezintă totalitatea argumentelor poziționale ale unui script. Deși la prima vedere par similari, comportamentul lor diferă în anumite contexte, în special atunci când sunt utilizați între ghilimele.”*

Ce vei învăța

În acest capitol vei înțelege rolul parametrilor speciali `$*` și `$@`, asemănările și diferențele dintre ei și situațiile în care fiecare dintre ei este utilizat.

Introducere

Am învățat deja:

`$1`
`$2`
`$3`

Dar dacă un script primește un număr necunoscut de argumente?

Exemplu:

```
./script.sh unu doi trei patru cinci
```

Nu este practic să accesăm fiecare argument individual.

Avem nevoie de un mecanism care să reprezinte **toate argumentele**.

Bash oferă două astfel de mecanisme:

`$*`

și

`$@`

Ideea fundamentală

Ambele reprezintă lista parametrilor poziționali.

Diferențele apar în modul în care Bash îi extinde, în special atunci când sunt utilizați între ghilimele.

Primul exemplu

Script:

```
#!/bin/bash
```

```
echo $*
```

Execuție:

```
./script.sh Linux Bash LPIC
```

Rezultat:

```
Linux Bash LPIC
```

Același rezultat îl obținem și cu:

```
echo $@
```

La prima vedere par identice.

Unde apare diferența?

Diferența importantă apare atunci când folosim ghilimelele.

```
"$*"
```

și

```
"$@"
```

nu au același comportament.

Comportamentul lui "\$*"

```
"$*"
```

este tratat ca **un singur șir de caractere**.

Exemplu:

Argumente:

```
Linux  
Bash  
LPIC
```

Rezultatul este echivalent cu:

```
"Linux Bash LPIC"
```

Există un singur argument.

Comportamentul lui "\$@"

"\$@"

păstrează fiecare argument separat.

Rezultatul este echivalent cu:

"Linux"

"Bash"

"LPIC"

Există trei argumente distincte.

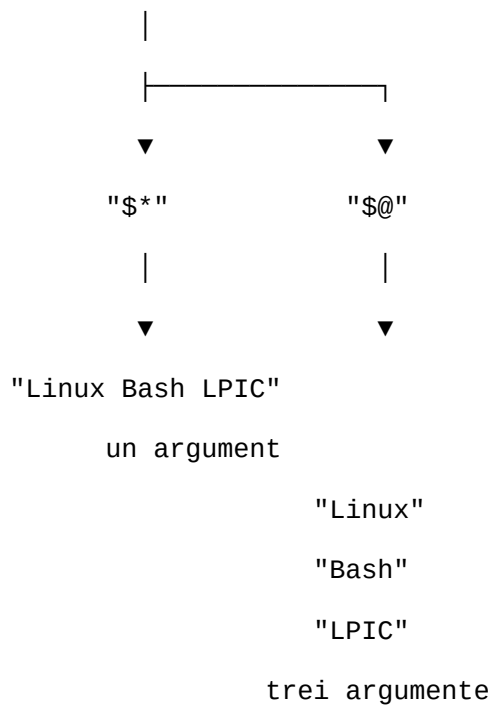
Modelul mental

Argumente

Linux

Bash

LPIC



De ce este important?

Presupunem că unul dintre argumente conține spații.

`./script.sh "Linux Mint" Bash`

Cu:

```
"$@"
```

Bash păstrează:

```
"Linux Mint"
```

```
"Bash"
```

ca două argumente.

Acesta este comportamentul dorit în majoritatea scripturilor.

Caseta „În spatele cortinei”

Conform specificației Bash:

- "\$*" produce un singur șir rezultat din concatenarea argumentelor.
- "\$@" produce câte un argument separat pentru fiecare parametru pozițional.

Aceasta este diferența esențială dintre cele două forme.

Greșeală din lumea reală

Un administrator utilizează:

```
"$*"
```

într-un script care procesează nume de fișiere.

Dacă unul dintre nume conține spații, toate argumentele sunt tratate ca un singur șir.

Rezultatul nu este cel dorit.

În această situație, forma recomandată este:

```
"$@"
```

deoarece păstrează structura originală a argumentelor.

Administrator Notes

În scripturile moderne Bash, atunci când dorești să procesezi toate argumentele primite, este recomandată utilizarea:

```
"$@"
```

Aceasta păstrează fiecare argument exact așa cum a fost transmis utilizatorului.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul parametrilor `$*` și `$@`;
 - diferența dintre formele cu și fără ghilimele;
 - faptul că `"$@"` păstrează argumentele separate.
-

Laborator 8.8.2

Nivel 1 – Înțelegere

Creează scriptul:

```
#!/bin/bash  
  
echo "$*"  
  
echo "$@"
```

Execută:

```
./script.sh Linux Bash LPIC
```

Observă rezultatul.

Nivel 2 – Aplicare

Execută:

```
./script.sh "Linux Mint" Bash
```

Compară comportamentul parametrilor speciali.

Nivel 3 – Administrator

Execută același script folosind argumente care conțin spații:

```
./script.sh "/var/log/apache2 access.log" "/etc/ssh/sshd_config"
```

Observă modul în care Bash tratează argumentele și explică de ce `"$@"` este alegerea recomandată.

Soluția laboratorului

- `$*` și `$@` reprezintă toate argumentele poziționale.
 - Diferența esențială apare în forma între ghilimele.
 - `"$*"` produce un singur șir.
 - `"$@"` păstrează fiecare argument separat.
-

Ce trebuie să reții

- `$*` și `$@` reprezintă lista argumentelor poziționale.
- Fără ghilimele, comportamentul lor este foarte asemănător.
- Diferența importantă apare în forma citată.
- `"$@"` este forma recomandată în majoritatea scripturilor Bash.

Capitolul 9

Shell Expansions (Expansiunile Shell-ului)

„Înainte de a executa o comandă, Bash analizează și transformă linia de comandă printr-o succesiune bine definită de expansiuni. Rezultatul acestor transformări este lista finală de argumente transmisă programului care urmează să fie executat.”

Ce vei învăța

În acest capitol vei înțelege ce sunt expansiunile shell-ului, în ce ordine sunt aplicate și de ce această ordine este importantă pentru interpretarea corectă a comenzilor Bash.

Introducere

Până acum am utilizat numeroase mecanisme Bash:

```
echo $HOME
```

```
echo $(date)
```

```
echo *.txt
```

```
echo {1..5}
```

La prima vedere pare că Bash execută pur și simplu comenzile.

În realitate, înainte de execuție, shell-ul transformă linia de comandă.

Acest proces poartă numele de:

Shell Expansion.

Ideea fundamentală

Atunci când utilizatorul apasă tasta **Enter**, Bash nu execută imediat comanda.

Mai întâi analizează și transformă linia introdusă.

Abia după finalizarea acestor transformări lansează programul.

Exemplu

Executăm:

```
echo $HOME
```

Utilizatorul a introdus:

```
$HOME
```

Programul `echo` nu primește niciodată textul:

```
$HOME
```

Bash îl înlocuiește înainte cu valoarea variabilei.

Programul primește deja:

```
/home/student
```

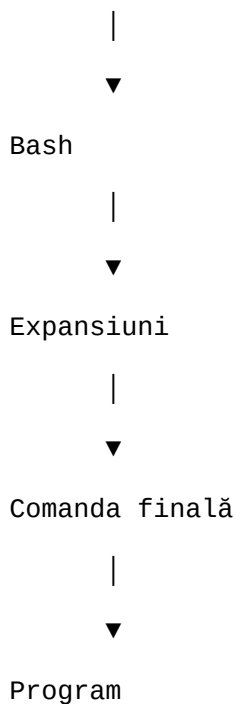
Modelul mental

Utilizator

|



Linia de comandă



Ordinea expansiunilor

Conform documentației oficiale Bash, expansiunile sunt aplicate în următoarea ordine:

1. Brace Expansion
2. Tilde Expansion
3. Parameter Expansion
4. Variable Assignment
5. Command Substitution
6. Arithmetic Expansion
7. Word Splitting
8. Filename Expansion (Pathname Expansion)

La final, Bash elimină caracterele de quoting care și-au îndeplinit rolul.

Notă pentru autor: În manual aș păstra această ordine exact ca în `man bash`. Ea va deveni firul conducător al întregului capitol.

De ce contează ordinea?

Să analizăm:

```
echo "$HOME"
```

Mai întâi Bash execută:

- Parameter Expansion.

Abia la final elimină ghilimelele.

Dacă ordinea ar fi inversă, rezultatul ar fi diferit.

Ordinea expansiunilor explică multe dintre comportamentele aparent ciudate ale shell-ului.

Ce urmează?

Vom studia fiecare tip de expansiune separat:

- Brace Expansion
- Tilde Expansion
- Parameter Expansion
- Command Substitution
- Arithmetic Expansion
- Word Splitting
- Filename Expansion
- Quote Removal

Fiecare reprezintă un mecanism distinct.

Administrator Notes

Expansiunile sunt aplicate de Bash înainte ca programul să fie executat.

Aceasta înseamnă că programe precum:

- `echo`
- `ls`
- `grep`
- `find`

nu cunosc existența variabilelor Bash sau a caracterelor wildcard.

Ele primesc deja rezultatul expansiunilor efectuate de shell.

Legătura cu LPIC

Shell Expansions reprezintă unul dintre subiectele importante din LPIC-1.

Înțelegerea ordinii expansiunilor este esențială pentru interpretarea corectă a comenzilor și pentru depanarea scripturilor.

Laborator 9.1

Nivel 1 – Înțelegere

Execută:

```
echo $HOME
```

Apoi:

```
echo "$HOME"
```

Compară rezultatele.

Nivel 2 – Aplicare

Execută:

```
echo $(date)
```

și:

```
echo *.sh
```

Identifică tipul de expansiune utilizat în fiecare caz.

Nivel 3 – Administrator

Analizează următoarele comenzi și stabilește ce tip de expansiune este aplicat:

```
echo ~/Documente
```

```
echo {1..5}
```

```
echo "$PATH"
```

```
echo $(hostname)
```

```
echo *.conf
```

Nu este necesar să explici încă mecanismele. Identifică doar categoria de expansiune.

Ce trebuie să reții

- Bash aplică o serie de expansiuni înainte de executarea unei comenzi.
- Programele executate primesc rezultatul expansiunilor, nu textul introdus de utilizator.
- Ordinea expansiunilor este bine definită și influențează rezultatul final.
- Fiecare tip de expansiune va fi studiat separat în capitolele următoare.

Capitolul 11

11.2 Mediul de execuție al unei comenzi (Command Execution Environment)

„Orice comandă executată în Bash rulează într-un mediu de execuție. Acesta reprezintă ansamblul informațiilor pe care procesul le primește înainte de începerea execuției.”

Ce vei învăța

În acest capitol vei înțelege ce reprezintă mediul de execuție al unei comenzi, ce informații sunt transmise unui proces nou și de ce modificările efectuate de un proces copil nu afectează shell-ul care l-a lansat.

Introducere

Atunci când executăm:

```
ls
```

sau

```
grep "root" /etc/passwd
```

programul nu pornește într-un mediu gol.

Înainte ca execuția să înceapă, Bash îi transmite un set de informații necesare funcționării.

Acest ansamblu poartă numele de **Command Execution Environment**.

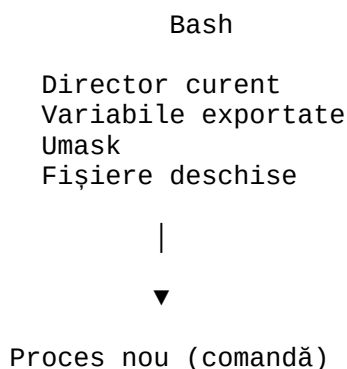
Ce conține mediul de execuție?

În forma sa simplificată, un proces primește:

- directorul curent;
- variabilele de mediu exportate;
- masca de creare a fișierelor (`umask`);
- fișierele deschise și redirectionările active;
- opțiunile și configurările necesare execuției.

Aceste informații permit procesului să ruleze în același context în care a fost lansat.

Modelul mental



Procesul primește o copie a informațiilor necesare pentru execuție.

Un exemplu

Executăm:

```
cd /tmp
```

Apoi:

```
pwd
```

Rezultatul este:

```
/tmp
```

Dacă lansăm acum:

```
ls
```

programul `ls` începe execuția având ca director curent:

/tmp

Acesta face parte din mediul de execuție transmis de shell.

Variabilele exportate

Executăm:

```
export PROIECT="LPIC"
```

Apoi:

```
env | grep PROIECT
```

Rezultatul:

```
PROIECT=LPIC
```

Dacă lansăm un program nou, acesta poate utiliza variabila `PROIECT`, deoarece ea face parte din mediul de execuție.

Ce NU poate face un proces copil?

Acesta este unul dintre cele mai importante aspecte.

Un program executat de Bash **nu poate modifica mediul shell-ului părinte**.

De exemplu, un program extern nu poate schimba directorul curent al shell-ului din care a fost lansat.

Din acest motiv:

```
cd
```

este implementată ca **comandă internă (builtin)** și nu ca program extern.

Dacă ar fi un executabil separat, schimbarea directorului s-ar produce doar în procesul copil și s-ar pierde imediat după terminarea acestuia.

Caseta „În spatele cortinei”

Atunci când Bash execută un program extern, acesta rulează într-un mediu propriu.

Modificările efectuate asupra acelui mediu dispar odată cu terminarea procesului și nu modifică mediul shell-ului care l-a lansat.

Greșeală din lumea reală

Un administrator scrie un program care execută:

```
cd /var/log
```

și se așteaptă ca, după terminarea programului, terminalul să rămână în:

```
/var/log
```

Acest lucru nu se întâmplă.

Programul și shell-ul sunt procese diferite.

Directorul curent al shell-ului nu este modificat.

Administrator Notes

Atunci când observi că o modificare făcută într-un script sau într-un program „nu rămâne” după terminarea acestuia, întreabă-te mai întâi:

Modificarea a fost făcută în shell-ul curent sau într-un proces copil?

În foarte multe cazuri, aceasta este explicația comportamentului observat.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- ce reprezintă mediul de execuție al unei comenzi;
 - faptul că un proces copil moștenește mediul necesar execuției;
 - faptul că modificările efectuate de un proces copil nu afectează shell-ul părinte.
-

Laborator 11.2

Nivel 1 – Înțelegere

Execută:

```
pwd
```

Apoi:

```
ls
```

Observă că programul `ls` rulează în același director curent.

Nivel 2 – Aplicare

Exportă o variabilă:

```
export CURS="Bash"
```

Verifică existența ei folosind:

```
env | grep CURS
```

Nivel 3 – Administrator

Răspunde la următoarele întrebări:

1. De ce un program extern nu poate schimba directorul curent al shell-ului?
 2. De ce `cd` este o comandă builtin și nu un executabil extern?
 3. Ce informații sunt moștenite de un proces nou la lansare?
-

Ce trebuie să reții

- Fiecare comandă rulează într-un mediu de execuție.
- Mediul include informații precum directorul curent, variabilele exportate și fișierele deschise.
- Un proces copil primește o copie a mediului necesar execuției.
- Modificările efectuate de procesul copil nu modifică mediul shell-ului părinte.
- Acesta este motivul pentru care unele comenzi, precum `cd`, trebuie implementate ca **builtin-uri**.

Capitolul 11

11.3 Mediul proceselor (ENVIRONMENT)

„Atunci când Bash execută un program, acesta primește un mediu de execuție reprezentat printr-o colecție de perechi nume=valoare. Aceste informații sunt utilizate de program pe întreaga durată a execuției sale.”

Ce vei învăța

În acest capitol vei înțelege cum este construit mediul unui proces, cum sunt transmise variabilele către programele executate și cum poate fi modificat acest mediu.

Introducere

Am învățat deja că Bash poate crea variabile:

```
NUME="Daniel"
```

și că unele dintre acestea pot fi exportate:

```
export NUME
```

Dar ce primește efectiv programul executat?

Răspunsul este:

un mediu de execuție format din perechi nume=valoare.

Forma mediului

Din punctul de vedere al unui program, mediul este o colecție de înregistrări de forma:

```
HOME=/home/student
```

```
USER=student
```

```
PATH=/usr/local/bin:/usr/bin:/bin
```

```
LANG=ro_RO.UTF-8
```

Fiecare element conține:

- numele variabilei;
 - caracterul =;
 - valoarea variabilei.
-

Construirea mediului

La pornire, Bash citește mediul primit de la procesul părinte.

Pentru fiecare variabilă găsită:

- creează un parametru shell cu același nume;

- îl marchează pentru export către procesele copil.

Astfel, fiecare comandă executată moștenește mediul construit de shell.

Modificarea mediului

Dacă modificăm o variabilă exportată:

```
export EDITOR=vim
```

noua valoare va fi transmisă comenzilor executate ulterior.

Dacă eliminăm o variabilă folosind:

```
unset EDITOR
```

aceasta nu va mai face parte din mediul transmis proceselor noi.

Modelul mental

Proces părinte

|



Bash

|



Construiește mediul

|



HOME=...

PATH=...

LANG=...

EDITOR=...

|



Program executat

Caseta „În spatele cortinei”

Mediul este transmis programului în momentul lansării acestuia.

După începerea execuției, programul lucrează cu propria copie a mediului.

Modificările făcute de acel program nu modifică automat mediul shell-ului care l-a lansat.

Greșeală din lumea reală

Un administrator execută:

```
EDITOR=vim
```

și presupune că toate programele lansate ulterior vor utiliza această valoare.

Nu este întotdeauna adevărat.

Dacă variabila nu este exportată:

```
export EDITOR
```

ea rămâne doar o variabilă a shell-ului și nu este inclusă în mediul proceselor copil.

Administrator Notes

Atunci când un program pare să ignore o variabilă de configurare, verifică mai întâi dacă aceasta a fost exportată în mediu.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- că mediul este format din perechi `nume=valoare`;
 - faptul că Bash construiește mediul proceselor executate;
 - rolul comenzilor `export` și `unset`.
-

Laborator 11.3

Nivel 1 – Înțelegere

Afișează mediul curent:

```
env
```

Identifică variabilele:

- HOME
 - USER
 - PATH
 - LANG
-

Nivel 2 – Aplicare

Execută:

```
export PROIECT="LPIC"
```

```
env | grep PROIECT
```

Apoi elimină variabila:

```
unset PROIECT
```

Verifică din nou:

```
env | grep PROIECT
```

Nivel 3 – Administrator

Creează un script care afișează:

- utilizatorul curent;
- directorul HOME;
- valoarea variabilei PATH;
- valoarea unei variabile definite și exportate de utilizator.

Explică de ce scriptul poate accesa această variabilă.

Ce trebuie să reții

- Mediul unui proces este format din perechi `nume=valoare`.
- Bash construiește mediul pe baza variabilelor exportate.
- Programele executate primesc o copie a acestui mediu.
- `export` adaugă variabile în mediul transmis proceselor copil.
- `unset` elimină variabilele din shell și din mediul proceselor lansate ulterior.

Capitolul 11

11.4 Variabila `BASH_ENV` și shell-urile neinteractive

„Atunci când Bash este pornit în mod neinteractiv, el poate executa automat comenzile din fișierul indicat de variabila de mediu `BASH_ENV`.”

Ce vei învăța

În acest capitol vei înțelege rolul variabilei `BASH_ENV`, când este utilizată și de ce este importantă în execuția scripturilor Bash.

Introducere

Am învățat deja că un shell interactiv citește fișiere precum:

- `/etc/profile`
- `~/.bash_profile`
- `~/.bashrc`

Dar ce se întâmplă atunci când Bash execută un script?

În acest caz, comportamentul este diferit.

Shell neinteractiv

Atunci când Bash este pornit pentru a executa un script, el rulează într-un **shell neinteractiv**.

În această situație, fișierul `~/.bashrc` nu este citit automat.

În schimb, Bash verifică existența variabilei:

```
BASH_ENV
```

Rolul variabilei BASH_ENV

Dacă variabila este definită:

```
export BASH_ENV="$HOME/.bash_env"
```

Bash îi extinde valoarea și execută comenzile din fișierul indicat înainte de rularea scriptului.

Conceptual, comportamentul este echivalent cu:

```
if [ -n "$BASH_ENV" ]; then
    . "$BASH_ENV"
fi
```

Aceasta este chiar logica descrisă în documentația oficială.

Un exemplu

Definim:

```
export BASH_ENV="$HOME/.bash_env"
```

Fișierul:

```
~/ .bash_env
```

conține:

```
export PROIECT="LPIC"
```

Orice script Bash pornit în acel mediu va avea acces la variabila:

```
$PROIECT
```

fără a fi necesară definirea ei în fiecare script.

Atenție

Conform documentației Bash, valoarea lui BASH_ENV este utilizată direct ca nume de fișier.

Bash **nu caută acel fișier în directoarele din PATH**. Fișierul trebuie indicat printr-o cale validă.

Caseta „În spatele cortinei”

BASH_ENV este folosit numai pentru shell-urile Bash neinteractive.

El nu înlocuiește fișierele de inițializare utilizate de shell-urile interactive, precum ~/ .bashrc.

Greșeală din lumea reală

Un administrator adaugă un alias în:

```
~/ .bashrc
```

Apoi execută un script și observă că aliasul nu există.

Explicația este simplă.

Scriptul rulează într-un shell neinteractiv, iar ~/ .bashrc nu este încărcat automat în acest caz.

Administrator Notes

Nu presupune că un script Bash beneficiază de aceeași configurație ca terminalul tău interactiv.

Dacă un script depinde de anumite variabile sau funcții, definește-le explicit sau utilizează BASH_ENV atunci când acest lucru este justificat.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- diferența dintre shell-ul interactiv și cel neinteractiv;
 - rolul variabilei BASH_ENV;
 - faptul că ~/ .bashrc nu este citit automat la executarea unui script Bash.
-

Laborator 11.4

Nivel 1 – Înțelegere

Afișează valoarea variabilei:

```
echo "$BASH_ENV"
```

Observă dacă este definită.

Nivel 2 – Aplicare

Creează fișierul:

```
~/ .bash_env
```

care conține:

```
export TEST_VAR="Bash"
```

Definește apoi:

```
export BASH_ENV="$HOME/ .bash_env"
```

Execută un script care afișează:

```
echo "$TEST_VAR"
```

Nivel 3 – Administrator

Explică de ce un script nu poate utiliza automat aliasurile și funcțiile definite în `~/ .bashrc` și descrie modul în care `BASH_ENV` poate fi utilizat pentru inițializarea shell-urilor neinteractive.

Ce trebuie să reții

- `BASH_ENV` este utilizată de Bash în shell-urile neinteractive.
- Dacă este definită, Bash execută fișierul indicat înainte de rularea scriptului.
- Fișierul este utilizat direct, fără căutare în `PATH`.
- `BASH_ENV` nu înlocuiește `~/ .bashrc`, ci are un rol diferit în inițializarea shell-urilor neinteractive.

Capitolul 11

11.5 Codurile de ieșire (Exit Status)

„Orice comandă executată în Bash se încheie cu un cod de ieșire. Acest cod este utilizat de shell pentru a determina dacă execuția s-a încheiat cu succes sau dacă a apărut o eroare.”

Ce vei învăța

În acest capitol vei înțelege intervalul valorilor posibile ale codurilor de ieșire, semnificația celor mai importante coduri și modul în care Bash utilizează aceste valori.

Introducere

În capitolul dedicat parametrului special:

\$?

am învățat că Bash memorează codul de ieșire al ultimei comenzi executate.

Acum vom răspunde la întrebarea:

Ce reprezintă, de fapt, aceste valori?

Intervalul valorilor

Conform documentației Bash, codurile de ieșire sunt valori întregi cuprinse între:

0

și

255

Aceste valori sunt întoarse către shell la terminarea execuției unei comenzi.

Valoarea 0

Valoarea:

0

înseamnă:

Execuția s-a încheiat cu succes.

Acesta este codul de ieșire pe care majoritatea comenzilor îl întorc atunci când nu apare nicio eroare.

Valorile diferite de 0

Orice valoare diferită de:

0

indică faptul că programul semnalează o eroare sau o situație specială.

Semnificația exactă depinde de programul executat, însă Bash utilizează anumite valori cu semnificație proprie.

Codul 126

Dacă o comandă este găsită, dar nu poate fi executată, Bash întoarce:

126

Exemple:

- fișier fără permisiune de execuție;
 - fișier care nu poate fi executat ca program.
-

Codul 127

Dacă o comandă nu poate fi găsită, Bash întoarce:

127

Exemplu:

comanda_inexistenta

Rezultat:

bash: comanda_inexistenta: command not found

iar codul de ieșire va fi:

127

Terminarea prin semnal

Dacă un proces este întrerupt de un semnal, Bash utilizează valoarea:

128 + numărul semnalului

pentru codul de ieșire.

De exemplu, un proces terminat prin semnalul 9 (SIGKILL) va avea codul de ieșire:

137

Comenzile builtin

Comenzile interne ale Bash respectă aceeași convenție:

- 0 pentru succes;
- valoare diferită de 0 în caz de eroare.

În cazul unei utilizări incorecte (de exemplu opțiuni invalide sau argumente lipsă), multe comenzi builtin întorc:

2

Erori înainte de execuție

Nu toate erorile provin din programul executat.

Dacă apare o eroare în timpul:

- expansiunilor;
- redirectionărilor;

Bash poate încheia comanda cu un cod de ieșire diferit de zero fără ca programul extern să fi fost executat.

Caseta „În spatele cortinei”

Pentru Bash, codul de ieșire reprezintă principalul mecanism prin care programele comunică rezultatul execuției.

Acesta este motivul pentru care structuri precum:

- `if`
- `while`
- `&&`
- `||`

se bazează pe codurile de ieșire și nu pe mesajele afișate pe ecran.

Greșeală din lumea reală

Un administrator verifică succesul unei comenzi analizând textul afișat pe ecran.

Această metodă nu este sigură.

Mesajele pot diferi între versiuni sau pot fi traduse în alte limbi.

Metoda recomandată este verificarea codului de ieșire.

Administrator Notes

Atunci când scrii scripturi, bazează deciziile pe codurile de ieșire și nu pe textul produs de comenzi.

Acest lucru face scripturile mai robuste și independente de limba sistemului sau de formatul mesajelor.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- intervalul codurilor de ieșire;
 - semnificația valorii 0;
 - rolul codurilor 126 și 127;
 - faptul că Bash utilizează `128 + semnal` pentru procesele terminate de un semnal.
-

Laborator 11.5

Nivel 1 – Înțelegere

Execută:

```
pwd  
echo $?
```

Apoi:

```
comanda_inexistenta  
echo $?
```

Compară rezultatele.

Nivel 2 – Aplicare

Creează un script fără permisiune de execuție și încearcă să îl rulezi:

```
./script.sh  
echo $?
```

Observă codul de ieșire.

Nivel 3 – Administrator

Execută următoarele situații și notează codul de ieșire obținut:

- comandă executată cu succes;
- comandă inexistentă;
- fișier fără permisiune de execuție.

Explică diferențele observate.

Ce trebuie să reții

- Codurile de ieșire au valori între 0 și 255.
- 0 înseamnă succes.
- 126 indică o comandă găsită, dar care nu poate fi executată.
- 127 indică o comandă care nu a fost găsită.
- Procesele terminate de un semnal folosesc codul 128 + numărul semnalului.
- Bash utilizează codurile de ieșire pentru controlul fluxului de execuție al scripturilor.

Capitolul 11

11.6 Semnalele (Signals)

„Semnalele reprezintă mecanismul prin care kernelul și procesele comunică evenimente importante unui proces aflat în execuție.”

Ce vei învăța

În acest capitol vei înțelege ce este un semnal, cum ajunge acesta la un proces și care sunt cele mai întâlnite semnale în administrarea sistemelor Linux.

Introducere

Până acum am urmărit ciclul normal al execuției unei comenzi.

Utilizatorul execută:

```
sleep 300
```

Programul începe să ruleze.

Dar apare o întrebare firească.

Cum poate fi oprit acel proces înainte să termine?

Răspunsul este:

prin intermediul **semnalelor**.

Ce este un semnal?

Un semnal este o notificare trimisă unui proces.

Această notificare poate proveni:

- de la kernel;
- de la un alt proces;
- de la utilizator.

Semnalul nu este o comandă.

El reprezintă un eveniment care solicită procesului să efectueze o anumită acțiune.

Modelul mental

Utilizator

|

▼

kill

|

▼

kernel

|

▼

Semnal

|



Proces

Kernelul este cel care livrează semnalul procesului.

Cele mai utilizate semnale

În practică, administratorii întâlnesc foarte des următoarele semnale:

Semnal	Nume	Utilizare
2	SIGINT	întreruperea unui program (Ctrl+C)
9	SIGKILL	terminarea forțată a unui proces
15	SIGTERM	solicitarea normală de terminare
18/19*	SIGCONT / SIGSTOP	continuarea sau oprirea execuției

Numerele pot varia pentru anumite arhitecturi; numele semnalelor sunt cele importante.

SIGINT

Atunci când apăsăm:

Ctrl+C

terminalul nu „omoară” direct programul.

El transmite procesului semnalul:

SIGINT

Procesul decide dacă îl tratează sau își încheie execuția.

SIGTERM

Comanda:

kill PID

trimite implicit:

SIGTERM

Acesta este modul recomandat de terminare a unui proces.

Programul are posibilitatea:

- să elibereze memoria;
 - să închidă fișierele;
 - să salveze datele înainte de oprire.
-

SIGKILL

Dacă procesul nu răspunde, administratorul poate utiliza:

```
kill -9 PID
```

Acest lucru trimite:

SIGKILL

Spre deosebire de SIGTERM, acest semnal nu poate fi interceptat sau ignorat.

Kernelul oprește imediat procesul.

Caseta „În spatele cortinei”

Nu toate semnalele înseamnă „oprirea unui program”.

Unele notifică:

- modificarea dimensiunii terminalului;
- expirarea unui temporizator;
- terminarea unui proces copil;
- continuarea execuției unui proces suspendat.

Bash poate reacționa la aceste evenimente prin mecanismul `trap`, pe care îl vom studia într-un capitol dedicat.

Greșeală din lumea reală

Mulți administratori folosesc direct:

```
kill -9
```

pentru orice proces blocat.

Aceasta ar trebui să fie **ultima soluție**, nu prima.

În majoritatea situațiilor este recomandat să se încerce mai întâi:

```
kill PID
```

care trimite **SIGTERM** și oferă aplicației șansa de a se închide controlat.

Administrator Notes

Reține regula practică:

1. încearcă **SIGTERM**;
2. verifică dacă procesul se închide;
3. utilizează **SIGKILL** doar dacă procesul nu mai răspunde.

Această abordare reduce riscul pierderii de date sau al coruperii fișierelor.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- ce este un semnal;
 - diferența dintre **SIGTERM** și **SIGKILL**;
 - efectul combinației **Ctrl+C**;
 - utilizarea comenzii **kill**.
-

Laborator 11.6

Nivel 1 – Înțelegere

Pornește un proces:

```
sleep 300
```

Întrerupe-l folosind:

```
ctrl+c
```

Observă revenirea la prompt.

Nivel 2 – Aplicare

Pornește procesul în fundal:

```
sleep 300 &
```

Identifică PID-ul și execută:

```
kill PID
```

Verifică dacă procesul s-a încheiat.

Nivel 3 – Administrator

Pornește un proces `sleep` în fundal și testează:

```
kill PID
```

iar apoi, dacă procesul încă există:

```
kill -9 PID
```

Compară comportamentul și explică diferența dintre cele două semnale.

Ce trebuie să reții

- Semnalele sunt mecanismul standard de comunicare între kernel și procese.
- **SIGINT** este transmis de obicei prin **Ctrl+C**.
- **SIGTERM** solicită terminarea controlată a unui proces.
- **SIGKILL** oprește imediat procesul și nu poate fi interceptat.
- În administrarea sistemelor, **SIGTERM** este alegerea recomandată, iar **SIGKILL** reprezintă soluția de ultimă instanță.

Capitolul 11

11.7 Controlul proceselor (Job Control)

„Job Control reprezintă mecanismul prin care Bash permite suspendarea, reluarea și administrarea proceselor pornite din shell-ul curent.”

Ce vei învăța

În acest capitol vei înțelege ce este un **job**, care este diferența dintre un proces și un job și cum permite Bash administrarea comenzilor aflate în execuție.

Introducere

Până acum am executat comenzi precum:

```
sleep 300
```

sau

```
find / -name "*.log"
```

Aceste comenzi rulează până la terminarea execuției.

Dar apare o întrebare firească.

Ce facem dacă dorim să întrerupem temporar execuția unei comenzi și să revenim ulterior la ea?

Acesta este rolul mecanismului **Job Control**.

Ce este un job?

În Bash, un **job** reprezintă una sau mai multe procese pe care shell-ul le administrează ca pe o singură unitate.

Un job poate fi format din:

- un singur proces;

sau

- o întreagă conductă (pipeline).

Exemplu:

```
ps aux | grep ssh
```

Deși sunt două procese, Bash le tratează ca un singur job.

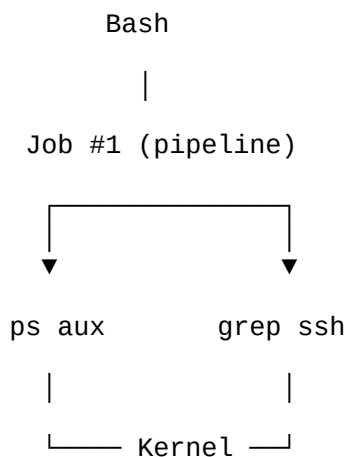
Job și proces

Este important să nu confundăm cele două concepte.

Un **proces** este o entitate gestionată de kernel.

Un **job** este o abstracție introdusă de Bash pentru a administra unul sau mai multe procese pornite din același shell.

Modelul mental



Kernelul cunoaște procesele.

Bash cunoaște joburile.

Pornirea unui job în fundal

Executăm:

```
sleep 300 &
```

Rezultat posibil:

```
[1] 25431
```

Bash afișează două informații importante:

- **[1]** – numărul jobului;
- **25431** – PID-ul ultimului proces din acel job.

Aceste două valori nu trebuie confundate.

Tabela joburilor

Bash păstrează o evidență a joburilor active.

Aceasta poate fi afișată cu:

```
jobs
```

Rezultat posibil:

```
[1]+  Running      sleep 300 &
```

Comanda `jobs` afișează informațiile cunoscute de Bash despre procesele pe care le administrează.

Suspendarea unui proces

Dacă o comandă rulează în prim-plan, apăsarea:

```
Ctrl+Z
```

nu o încheie.

Ea transmite semnalul de suspendare, iar controlul revine shell-ului.

Rezultat posibil:

```
[1]+  Stopped      sleep 300
```

Procesul există în continuare, dar execuția lui este suspendată.

Continuarea execuției

Un job suspendat poate fi reluat în fundal:

```
bg
```

sau readus în prim-plan:

```
fg
```

Aceste comenzi acționează asupra joburilor gestionate de Bash.

Caseta „În spatele cortinei”

Bash asociază fiecărui pipeline un job și păstrează o tabelă internă a acestora.

Kernelul nu știe nimic despre „joburi”; el gestionează doar procese și grupuri de procese. Conceptul de **job** aparține shell-ului.

Greșeală din lumea reală

Administratorul încearcă:

```
kill %1
```

într-un alt terminal și este surprins că nu funcționează.

Explicația este simplă.

Notația:

```
%1
```

reprezintă un **jobspec** și este cunoscută doar de shell-ul care a creat acel job. În alt terminal, Bash are o altă tabelă de joburi.

Administrator Notes

Folosește:

- **jobs** pentru administrarea joburilor din shell-ul curent;
- **ps** pentru vizualizarea proceselor din întregul sistem.

Cele două comenzi răspund la întrebări diferite.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- diferența dintre proces și job;
 - rolul comenzilor **jobs**, **bg** și **fg**;
 - efectul combinației **Ctrl+Z**;
 - diferența dintre PID și numărul unui job.
-

Laborator 11.7

Nivel 1 – Înțelegere

Execută:

```
sleep 300
```

Apasă:

`Ctrl+Z`

Observă mesajul afișat de Bash.

Nivel 2 – Aplicare

Reia execuția folosind:

`bg`

Afișează apoi lista joburilor:

`jobs`

Readu procesul în prim-plan:

`fg`

Nivel 3 – Administrator

Pornește două procese în fundal:

```
sleep 300 &  
sleep 600 &
```

Folosește:

`jobs`

Identifică:

- numărul fiecărui job;
- PID-ul afișat;
- starea fiecărui job.

Explică diferența dintre informațiile afișate de `jobs` și cele afișate de `ps`.

Ce trebuie să reții

- Un **job** este o abstracție introdusă de Bash pentru administrarea unuia sau mai multor procese.
- Kernelul gestionează procese; Bash gestionează joburi.
- `jobs` afișează joburile shell-ului curent.
- `Ctrl+Z` suspendă execuția unui job.

- `bg` continuă execuția în fundal, iar `fg` readuce jobul în prim-plan.

Capitolul 11

11.8 Identificarea joburilor (Job Specifications)

„Bash permite referirea la joburile pe care le administrează folosind identificatori speciali numiți job specifications (jobspec). Aceștia oferă o modalitate simplă de a controla procesele fără a utiliza PID-uri.”

Ce vei învăța

În acest capitol vei înțelege ce este un **jobspec**, cum identifică Bash un job și cum poate fi utilizat împreună cu comenzile `fg`, `bg` și `kill`.

Introducere

În capitolul anterior am văzut că Bash afișează:

```
sleep 300 &
```

Rezultat:

```
[1] 25431
```

Acum știm că:

- **1** este numărul jobului;
- **25431** este PID-ul ultimului proces din acel job.

Dar cum îi spunem lui Bash:

„Lucrează cu jobul 1.”

Răspunsul este:

jobspec.

Sintaxa generală

Un jobspec începe întotdeauna cu caracterul:

%

Exemple:

%1

%2

%3

Acestea identifică joburile după numărul lor.

Primul exemplu

Pornim două procese:

```
sleep 300 &  
sleep 600 &
```

Afișăm lista:

```
jobs
```

Rezultat posibil:

```
[1] - Running    sleep 300 &  
[2]+ Running    sleep 600 &
```

Pentru a readuce în prim-plan al doilea job:

```
fg %2
```

Identificarea după nume

Bash permite identificarea unui job și prin începutul comenzii care l-a creat.

Exemplu:

```
%sleep
```

Dacă există un singur job al cărui nume începe cu:

```
sleep
```

Bash îl identifică automat.

Identificarea după un fragment

Dacă nu cunoaștem începutul exact al comenzii, putem utiliza:

```
%?text
```

Exemplu:

```
%?ssh
```

Bash caută un job al cărui text conține șirul:

```
ssh
```

Dacă există o singură potrivire, acel job este selectat.

Jobul curent

Bash păstrează permanent noțiunea de **job curent**.

Acesta poate fi referit prin:

```
%%
```

sau

```
%+
```

Ambele desemnează același job.

Jobul anterior

Jobul utilizat înaintea celui curent poate fi referit prin:

```
%-
```

Acest mecanism este util atunci când alternăm între două joburi suspendate.

Modelul mental

Tabela joburilor

```
[1] sleep 300
```

```
[2] vim
```

```
[3] tail -f log
```

|



Jobspec

%1

%vim

;%tail

%%

%-

Jobspec-ul este un identificator înțeles de Bash.

Kernelul nu utilizează această sintaxă.

Utilizare practică

Reluarea în prim-plan:

`fg %1`

Reluarea în fundal:

`bg %1`

Terminarea unui job:

`kill %1`

Observă că `kill` poate primi atât un PID, cât și un jobspec atunci când este executat din Bash.

Caseta „În spatele cortinei”

Jobspec-urile există numai în shell-ul care a creat joburile.

Dacă deschizi un alt terminal, acesta are propria tabelă de joburi și nu poate utiliza jobspec-urile din primul terminal.

Greșeală din lumea reală

Administratorul execută:

`fg %apache`

și primește un mesaj de eroare.

Motivul este că există două joburi care încep cu același prefix.

În această situație, prefixul nu mai identifică un job în mod unic și Bash solicită o referință mai precisă.

Administrator Notes

În scripturi, utilizează de regulă PID-uri.

Jobspec-urile sunt destinate în principal utilizării interactive, direct în shell.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- sintaxa `%n`;
 - semnificația lui `%%`, `%+` și `%-`;
 - identificarea unui job după prefix (`%text`);
 - identificarea după un fragment (`??text`);
 - utilizarea jobspec-urilor împreună cu `fg`, `bg` și `kill`.
-

Laborator 11.8

Nivel 1 – Înțelegere

Pornește două procese în fundal:

```
sleep 300 &  
sleep 600 &
```

Afișează:

```
jobs
```

Identifică numărul fiecărui job.

Nivel 2 – Aplicare

Readu în prim-plan primul job:

```
fg %1
```

Suspendă-l folosind:

```
Ctrl+Z
```

Continuă execuția în fundal:

```
bg %1
```

Nivel 3 – Administrator

Pornește trei procese diferite.

Folosește:

```
jobs
```

Apoi testează următoarele forme de identificare:

```
fg %1  
fg %%  
kill %1
```

Explică ce reprezintă fiecare identificator și în ce situații este util.

Ce trebuie să reții

- Un **jobspec** este modul în care Bash identifică un job.
- **%n** identifică jobul cu numărul **n**.
- **%%** și **%+** reprezintă jobul curent.
- **%-** reprezintă jobul anterior.
- **%prefix** și **%?text** permit identificarea după numele comenzii.
- Jobspec-urile sunt valabile numai în shell-ul care gestionează acele joburi.

Capitolul 11

11.9 Notificarea modificării stării unui job

„Bash monitorizează permanent joburile pe care le administrează. Atunci când starea unui job se modifică, shell-ul poate informa utilizatorul imediat sau poate amâna afișarea mesajului până la afișarea următorului prompt.”

Ce vei învăța

În acest capitol vei înțelege când și cum informează Bash utilizatorul despre modificarea stării unui job și cum poate fi schimbat acest comportament.

Introducere

Pornim un proces în fundal:

```
sleep 10 &
```

După câteva secunde procesul se termină.

Totuși, de multe ori Bash **nu afișează imediat** mesajul:

```
[1]+ Done      sleep 10
```

Mesajul apare abia atunci când shell-ul este gata să afișeze din nou promptul. Acest comportament este cel implicit al lui Bash.

Comportamentul implicit

În mod normal, Bash așteaptă până când este pe punctul de a afișa promptul următor.

Abia atunci verifică dacă vreun job și-a schimbat starea și afișează mesajele corespunzătoare.

Acest mecanism evită întreruperea ieșirii produse de alte comenzi.

Modelul mental

```
sleep 10 &
```

|



Procesul se termină

|



Bash memorează schimbarea

|



Prompt nou

|



Mesajul este afișat

Notificare imediată

Acest comportament poate fi modificat folosind opțiunea:

```
set -b
```

sau echivalent:

```
set -o notify
```

În acest mod, Bash afișează imediat modificările de stare ale joburilor, fără să mai aștepte afișarea următorului prompt.

Revenirea la comportamentul implicit

Pentru dezactivare:

```
set +b
```

sau

```
set +o notify
```

Shell-ul revine la comportamentul standard.

De ce există acest mecanism?

Imaginează-ți că rulezi:

```
cat raport.txt
```

iar în timpul afișării textului un job din fundal se termină.

Dacă Bash ar afișa imediat notificarea, ieșirea comenzii ar putea fi întreruptă și mai greu de urmărit.

Din acest motiv, comportamentul implicit este să amâne notificarea până la următorul prompt.

Caseta „În spatele cortinei”

Bash detectează schimbările de stare ale proceselor prin mecanismele oferite de sistemul de operare.

Dacă există un `trap` definit pentru semnalul `SIGCHLD`, acesta este executat pentru fiecare proces copil care își schimbă starea.

Greșeală din lumea reală

Administratorul lansează mai multe procese în fundal și crede că unul dintre ele încă rulează deoarece nu a primit mesajul:

```
Done
```

În realitate, procesul s-a terminat deja.

Bash doar nu a afișat încă notificarea deoarece utilizatorul nu a revenit la prompt.

Administrator Notes

Dacă lucrezi interactiv, comportamentul implicit este de regulă cel mai confortabil.

Activează `set -b` doar atunci când ai nevoie să fii informat imediat despre terminarea proceselor din fundal.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- comportamentul implicit al notificărilor pentru joburi;

- rolul opțiunii `set -b (set -o notify)`;
 - faptul că Bash poate afișa imediat sau amânat modificările de stare ale joburilor.
-

Laborator 11.9

Nivel 1 – Înțelegere

Pornește:

```
sleep 5 &
```

Așteaptă terminarea procesului fără să introduci alte comenzi.

Observă momentul în care Bash afișează mesajul:

Done

Nivel 2 – Aplicare

Activează:

```
set -b
```

Pornește din nou:

```
sleep 5 &
```

Compară momentul afișării notificării.

Nivel 3 – Administrator

Testează ambele moduri de funcționare (`set +b` și `set -b`) lansând mai multe procese în fundal.

Explică avantajele și dezavantajele fiecărui comportament.

Ce trebuie să reții

- Bash monitorizează permanent joburile pe care le administrează.
- În mod implicit, notificările sunt afișate la următorul prompt.
- `set -b (set -o notify)` determină afișarea imediată a schimbărilor de stare ale joburilor.

- Acest mecanism există pentru a evita întreruperea ieşirii produse de comenzile aflate în execuţie.

Capitolul 11

11.10 Ieşirea din Bash atunci când există joburi active

„Pentru a preveni pierderea accidentală a proceselor aflate în execuţie, Bash verifică existenţa joburilor oprite înainte de închiderea shell-ului şi poate avertiza utilizatorul.”

Ce vei învăţa

În acest capitol vei înţelege ce verifică Bash înainte de terminarea unei sesiuni interactive şi de ce uneori comanda `exit` nu închide imediat shell-ul.

Introducere

Presupunem că executăm:

```
sleep 300
```

Suspendăm procesul folosind:

```
Ctrl+Z
```

Rezultat posibil:

```
[1]+  Stopped      sleep 300
```

Acum tastăm:

```
exit
```

Mulţi utilizatori se aşteaptă ca terminalul să se închidă imediat.

În realitate, Bash verifică dacă există joburi suspendate înainte de a încheia sesiunea.

Comportamentul implicit

Dacă există unul sau mai multe joburi oprite, Bash afișează un mesaj de avertizare și nu închide imediat shell-ul.

Scopul acestui mecanism este protejarea utilizatorului împotriva închiderii accidentale a unor procese importante.

De ce face Bash acest lucru?

Imaginează-ți că editezi un fișier cu un editor text.

Suspendezi editorul folosind:

`Ctrl+Z`

și, din neatenție, tastezi:

`exit`

Fără acest mecanism de protecție, editorul ar putea fi terminat înainte să salvezi modificările.

De aceea Bash preferă să avertizeze utilizatorul.

Modelul mental

Proces suspendat

|

▼

Utilizatorul tastează

`exit`

|

▼

Bash verifică tabela joburilor

|

├─ Nu există joburi

↓

Shell-ul se închide

└─ Există joburi suspendate

|

O a doua încercare

Conform comportamentului descris în man `bash`, dacă utilizatorul execută din nou:

```
exit
```

fără să intervină asupra joburilor existente, Bash nu mai afișează încă o dată avertizarea și continuă procesul de închidere a shell-ului.

Opțiunea `checkjobs`

Bash oferă și opțiunea:

```
shopt -s checkjobs
```

Atunci când este activată, shell-ul verifică nu doar existența joburilor suspendate, ci și a celor aflate în execuție și poate afișa lista acestora înainte de închidere.

Pentru revenirea la comportamentul implicit:

```
shopt -u checkjobs
```

Caseta „În spatele cortinei”

Acest mecanism funcționează doar pentru joburile pe care Bash le administrează.

Procesele pornite din alte terminale sau din alte sesiuni nu fac parte din tabela de joburi a shell-ului curent și nu sunt verificate la ieșire.

Greșeală din lumea reală

Un administrator pornește un proces în fundal, îl suspendă și încearcă să închidă terminalul.

Primește un mesaj de avertizare și crede că Bash a întâmpinat o eroare.

În realitate, shell-ul încearcă doar să prevină pierderea unui job pe care încă îl administrează.

Administrator Notes

Înainte de a închide o sesiune, verifică întotdeauna:

`jobs`

Dacă există joburi suspendate, decide dacă:

- le reiei cu `fg`;
- le continui în fundal cu `bg`;
- le închei folosind `kill`.

Astfel eviți procese rămase în stare suspendată sau închise accidental.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- faptul că Bash avertizează la ieșire dacă există joburi suspendate;
 - rolul opțiunii `checkjobs`;
 - utilizarea comenzii `jobs` înainte de închiderea unei sesiuni.
-

Laborator 11.10

Nivel 1 – Înțelegere

Execută:

`sleep 300`

Suspendă procesul:

`Ctrl+Z`

Încearcă:

`exit`

Observă comportamentul shell-ului.

Nivel 2 – Aplicare

Afișează:

`jobs`

Reia procesul:

`fg`

Înterupe-l cu:

`ctrl+c`

Execută din nou:

`exit`

Compară diferența.

Nivel 3 – Administrator

Activează:

`shopt -s checkjobs`

Pornește unul sau mai multe procese în fundal și încearcă să închizi shell-ul.

Observă informațiile suplimentare afișate și explică rolul opțiunii `checkjobs`.

Ce trebuie să reții

- Bash verifică existența joburilor înainte de închiderea unei sesiuni interactive.
- Dacă există joburi suspendate, shell-ul avertizează utilizatorul.
- Opțiunea `checkjobs` extinde această verificare și asupra joburilor aflate în execuție.
- Verificarea se aplică doar joburilor administrate de shell-ul curent.

Capitolul 12

Prompt-ul Bash (PROMPTING)

„Atunci când rulează interactiv, Bash afișează unul sau mai multe prompt-uri care indică faptul că shell-ul este pregătit să primească o comandă sau că așteaptă continuarea unei comenzi deja începute.”

Ce vei învăța

În acest capitol vei înțelege ce este un prompt Bash, care sunt principalele tipuri de prompt și în ce momente sunt afișate.

Introducere

Când deschizi un terminal, primul lucru pe care îl observi este prompt-ul.

De exemplu:

```
daniel@server:~$
```

Acesta nu este doar un element vizual.

El reprezintă semnalul că Bash a terminat executarea comenzii anterioare și este pregătit să citească o comandă nouă.

Prompt-ul principal – PS1

În modul interactiv, Bash afișează prompt-ul principal folosind variabila:

```
PS1
```

Acesta este prompt-ul pe care îl vedem înainte de fiecare comandă.

Exemplu:

```
echo "$PS1"
```

Rezultatul poate conține secvențe speciale care vor fi interpretate de Bash înainte de afișare.

Prompt-ul secundar – PS2

Uneori o comandă nu este completă.

Exemplu:

```
echo "
```

Utilizatorul apasă Enter.

Bash observă că ghilimelele nu au fost închise și nu execută încă comanda.

În locul prompt-ului obișnuit apare:

>

Acesta este prompt-ul secundar, controlat de variabila:

PS2

El indică faptul că Bash așteaptă continuarea comenzii.

Prompt-ul PS0

Mai puțin cunoscut este:

PS0

Conform documentației Bash, acesta este afișat **după citirea unei comenzi, dar înainte de executarea ei**.

Este utilizat rar, însă poate fi util în anumite scenarii de depanare sau personalizare a shell-ului.

Prompt-ul PS4

Variabila:

PS4

este utilizată atunci când este activat modul de depanare:

`set -x`

În acest caz, fiecare comandă executată este precedată de conținutul variabilei PS4.

Acest mecanism este foarte util pentru analiza scripturilor Bash.

Modelul mental

Pornire shell

|

▼

PS1

|

Utilizator introduce o comandă

|



Comanda este incompletă?



Da → PS2



Nu



Executare



PS1

Caseta „În spatele cortinei”

Prompt-ul nu este afișat de emulatorul de terminal.

El este generat de Bash.

Terminalul afișează doar textul primit de la shell.

Greșeală din lumea reală

Un utilizator vede prompt-ul:

>

și crede că terminalul s-a blocat.

În realitate, Bash așteaptă completarea unei comenzi neterminate, de exemplu:

- o ghilimeă deschisă;
 - o paranteză neînchisă;
 - un operator de continuare.
-

Administrator Notes

Dacă apare prompt-ul:

>

verifică mai întâi dacă ai închis toate ghilimelele, parantezele și construcțiile începute.

De cele mai multe ori, problema nu este Bash, ci faptul că linia de comandă este incompletă.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul variabilelor PS1, PS2, PS0 și PS4;
 - diferența dintre prompt-ul principal și cel secundar;
 - utilizarea lui PS4 împreună cu `set -x`.
-

Laborator 12.1

Nivel 1 – Înțelegere

Afișează valorile:

```
echo "$PS1"
echo "$PS2"
echo "$PS4"
```

Observă diferențele.

Nivel 2 – Aplicare

Introdu intenționat o comandă incompletă:

```
echo "
```

Observă apariția prompt-ului PS2.

Completează apoi comanda și execut-o.

Nivel 3 – Administrator

Activează depanarea:

```
set -x
```

Execută câteva comenzi simple și observă modul în care este utilizată variabila PS4.

Dezactivează depanarea:

```
set -x
```

Ce trebuie să reții

- PS1 este prompt-ul principal.
- PS2 apare atunci când Bash așteaptă continuarea unei comenzi.
- PS0 este afișat înainte de executarea unei comenzi.
- PS4 este utilizat în modul de depanare (`set -x`).
- Prompt-urile sunt generate de Bash, nu de emulatorul de terminal.

Capitolul 12

12.2 Secvențele speciale utilizate în prompt (Prompt Escape Sequences)

„Variabilele PS1, PS2, PS3 și PS4 pot conține secvențe speciale precedate de caracterul «\». Înainte de afișarea promptului, Bash înlocuiește aceste secvențe cu informațiile corespunzătoare.”

Ce vei învăța

În acest capitol vei înțelege cele mai utilizate secvențe speciale din prompt și modul în care Bash le transformă înainte de afișare.

Introducere

Până acum am aflat că promptul este controlat de variabila:

```
PS1
```

Dar dacă executăm:

```
echo "$PS1"
```

rezultatul poate fi asemănător cu:

```
\u@\h:\w\$\
```


Aceste caractere nu sunt afișate literalmente.

Înainte de afișarea promptului, Bash le interpretează și le înlocuiește cu informații despre sesiunea curentă.

Informații despre utilizator

\u

Reprezintă utilizatorul curent.

Exemplu:

```
PS1="\u$ "
```

Rezultat posibil:

```
student$
```

\h

Afișează numele sistemului până la primul punct.

Exemplu:

```
server
```

\H

Afișează numele complet al sistemului.

Exemplu:

```
server.example.com
```

Informații despre director

\w

Afișează directorul curent complet.

Exemplu:

~/Documente

\w

Afișează doar ultimul element al căii.

Dacă directorul curent este:

/home/student/Documente

rezultatul va fi:

Documente

Informații despre timp

Bash poate afișa ora în mai multe formate.

\t

14:35:42

(format 24 de ore)

\T

02:35:42

(format 12 ore)

\@

02:35 PM

(format 12 ore cu AM/PM)

\A

14:35

(format 24 de ore fără secunde)

Informații despre shell

\s

Numele shell-ului.

Rezultat:

bash

\v

Versiunea Bash.

Exemplu:

5.2

\V

Versiunea completă (versiune + patch level).

Informații despre joburi

\j

Afișează numărul joburilor administrate de shell în acel moment.

Dacă există două procese în fundal:

2

Alte secvențe utile

Secvență	Semnificație
\n	Linie nouă
\r	Carriage Return
\a	Caracter BEL (beep)

Secvență	Semnificație
\e	Caracter ESC
\\	Caracterul \
\\$	Afișează # pentru root și \$ pentru ceilalți utilizatori

Modelul mental

PS1

|

▼

\u@\h:\w\\$

|

▼

Bash interpretează secvențele

|

▼

daniel@server:~/Documente\$

Caseta „În spatele cortinei”

Secvențele speciale sunt expandate înainte ca promptul să fie afișat.

Ulterior, dacă opțiunea `promptvars` este activă (și este activă în mod implicit), Bash aplică și expansiunile obișnuite asupra rezultatului, precum expansiunea parametrilor, substituția de comenzi și expansiunea aritmetică.

Greșeală din lumea reală

Un utilizator modifică:

```
PS1="\u@\h:\w$ "
```

și observă că promptul se termină întotdeauna cu:

\$

chiar și atunci când lucrează ca `root`.

Forma recomandată este:

```
PS1="\u@\h:\w\$ "
```

Secvența `\$` permite lui Bash să afișeze automat:

- `$` pentru utilizatorii obișnuiți;
 - `#` pentru utilizatorul `root`.
-

Administrator Notes

Nu încerca să memorezi toate secvențele.

În practică sunt utilizate frecvent doar:

- `\u`
- `\h`
- `\w`
- `\W`
- `\A`
- `\$`

Acestea sunt suficiente pentru construirea majorității prompturilor utilizate în administrarea sistemelor Linux.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul variabilei `PS1`;
 - cele mai utilizate secvențe speciale;
 - diferența dintre `\w` și `\W`;
 - diferența dintre `\h` și `\H`;
 - rolul secvenței `\$`.
-

Laborator 12.2

Nivel 1 – Înțelegere

Afișează valoarea curentă:

```
echo "$PS1"
```

Identifică secvențele speciale utilizate.

Nivel 2 – Aplicare

Configurează temporar promptul:

```
PS1="\u@\h:\w\$ "
```

Navighează între mai multe directoare și observă modificările.

Nivel 3 – Administrator

Construiește un prompt care să afișeze:

- utilizatorul;
- hostname-ul;
- directorul curent;
- ora în format 24 de ore;
- simbolul corespunzător utilizatorului (\$ sau #).

Verifică modul în care promptul se actualizează după schimbarea directorului de lucru.

Ce trebuie să reții

- Promptul Bash poate conține secvențe speciale introduse prin caracterul \.
- Bash le înlocuiește cu informații despre utilizator, sistem și sesiunea curentă înainte de afișare.
- \u, \h, \w, \W, \A și \\$ sunt cele mai utilizate secvențe în practică.
- După expandarea acestor secvențe, dacă opțiunea `promptvars` este activă, promptul este supus și expansiunilor obișnuite Bash.

Capitolul 12

12.3 Expansiunile aplicate asupra promptului (promptvars)

„După decodificarea secvențelor speciale din prompt, Bash poate aplica asupra rezultatului expansiunea parametrilor, substituția de comenzi, expansiunea aritmetică și eliminarea ghilimelelor. Acest comportament este controlat de opțiunea `promptvars`.”

Ce vei învăța

În acest capitol vei înțelege de ce un prompt Bash poate afișa informații dinamice și care este rolul opțiunii `promptvars`.

Introducere

Până acum am utilizat secvențe precum:

```
PS1="\u@\h:\w\$ "
```

Acestea sunt interpretate de Bash înainte de afișarea promptului.

Dar Bash poate merge și mai departe.

Promptul poate conține:

- variabile;
- substituții de comenzi;
- expresii aritmetice.

Acestea sunt evaluate înainte ca promptul să fie afișat.

Un prim exemplu

Configurăm promptul astfel:

```
PS1='$(date +%H:%M:%S) \$ '
```

Rezultat posibil:

```
15:42:18 $
```

Observă că:

```
$(date +%H:%M:%S)
```

nu este afișat ca text.

Bash execută comanda și înlocuiește expresia cu rezultatul acesteia înainte de afișarea promptului.

Utilizarea variabilelor

Promptul poate utiliza și variabile.

Exemplu:

```
NUME="LPIC"
```

```
PS1="$NUME \$ "
```

Rezultat:

```
LPIC $
```

Valoarea variabilei este expandată înainte ca promptul să fie afișat.

Expresii aritmetice

Pot fi utilizate și expresii aritmetice.

Exemplu:

```
PS1='$( (10+5) ) \$ '
```

Rezultat:

```
15 $
```

Și această expresie este evaluată înainte de afișarea promptului.

Modelul mental

PS1

|

▼

Secvențe speciale

\u

\h

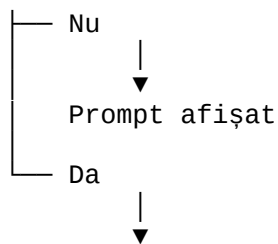
\w

|

▼

promptvars activ?

|



Parameter Expansion

Command Substitution

Arithmetic Expansion

Quote Removal

|



Prompt final

Opțiunea `promptvars`

Comportamentul descris este controlat de opțiunea:

`shopt promptvars`

În mod implicit, aceasta este activată.

Rezultat posibil:

`promptvars` `on`

Dezactivarea

Poate fi dezactivată folosind:

`shopt -u promptvars`

și reactivată cu:

`shopt -s promptvars`

După dezactivare, Bash nu mai aplică aceste expansiuni asupra promptului.

Caseta „În spatele cortinei”

Există două etape distincte:

1. Bash interpretează secvențele speciale (`\u`, `\w`, `\A` etc.).
2. Dacă `promptvars` este activă, rezultatul este supus mecanismelor obișnuite de expansiune ale shell-ului.

Această ordine este descrisă explicit în documentația oficială.

Greșeală din lumea reală

Un administrator observă că promptul execută o comandă la fiecare apăsare a tastei **Enter** și presupune că este un comportament al terminalului.

În realitate, Bash reevaluează promptul înainte de fiecare afișare, iar orice substituție de comandă din PS1 este executată din nou.

Din acest motiv, este recomandat ca comenzile incluse în prompt să fie rapide.

Administrator Notes

Evită introducerea în PS1 a unor comenzi costisitoare (de exemplu, căutări recursive sau interogări de rețea).

Promptul este evaluat înaintea fiecărei comenzi. Un prompt complex poate încetini vizibil lucrul în terminal.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- că promptul poate conține variabile și substituții de comenzi;
 - rolul opțiunii `promptvars`;
 - faptul că `promptvars` este activată implicit.
-

Laborator 12.3

Nivel 1 – Înțelegere

Verifică starea opțiunii:

```
shopt promptvars
```

Nivel 2 – Aplicare

Configurează temporar promptul:

```
PS1='$(date +%H:%M:%S) \u@\h:\w\$ '
```

Execută câteva comenzi și observă actualizarea orei.

Nivel 3 – Administrator

Dezactivează:

```
shopt -u promptvars
```

Testează același prompt și compară rezultatul.

Reactivează opțiunea:

```
shopt -s promptvars
```

Explică diferențele observate.

Ce trebuie să reții

- După interpretarea secvențelor speciale, Bash poate aplica și expansiunile obișnuite asupra promptului.
- Acest comportament este controlat de opțiunea `promptvars`.
- `promptvars` este activată implicit.
- Promptul este reevaluat înainte de fiecare afișare, motiv pentru care comenzile incluse în `PS1` sunt executate repetat.

Capitolul 13

13.1 Built-in-urile `:`, `true` și `false`

„Bash include câteva comenzi interne extrem de simple, utilizate frecvent în scripting pentru controlul fluxului de execuție și al codurilor de ieșire.”

Ce vei învăța

În acest capitol vei înțelege rolul comenzilor `:`, `true` și `false`, precum și situațiile în care acestea sunt utilizate în scripturile Bash.

Introducere

La prima vedere, comenzile:

`:`

`true`

`false`

par aproape inutile.

Primele două nu afișează nimic.

Ultima nu produce nici ea un mesaj de eroare.

Și totuși, ele apar foarte des în scripturile Bash.

Motivul este simplu.

Valoarea lor nu constă în ceea ce afișează, ci în **codul de ieșire** pe care îl întorc.

Comanda `:`

Caracterul:

`:`

reprezintă o comandă internă Bash.

Ea nu execută nicio operație și se încheie cu succes.

Exemplu:

```
:  
echo $?
```

Rezultat:

```
0
```

La ce este utilă?

Poate fi utilizată atunci când sintaxa Bash cere existența unei comenzi, dar nu dorim efectuarea niciunei operații.

Exemplu:

```
if grep "root" /etc/passwd >/dev/null  
then  
    :  
fi
```

Instrucțiunea este validă, iar blocul `then` nu execută nicio acțiune.

Comanda `true`

Comanda:

```
true
```

nu produce ieșire și returnează întotdeauna:

```
0
```

Exemplu:

```
true  
echo $?
```

Rezultat:

```
0
```

Comanda `false`

Comanda:

```
false
```

nu produce ieșire, dar returnează întotdeauna un cod diferit de zero (în practică, 1).

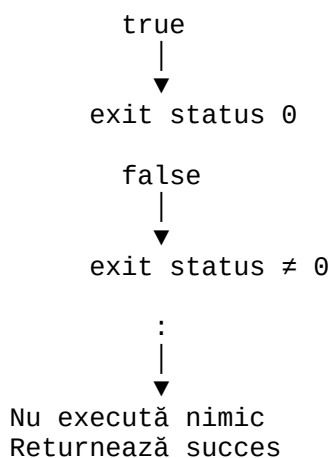
Exemplu:

```
false  
echo $?
```

Rezultat:

1

Modelul mental



De ce există trei comenzi?

La prima vedere, `:` și `true` par identice.

În multe situații chiar sunt.

Totuși, `:` are și un rol sintactic în Bash și apare frecvent în expresii de expansiune și în construcții specifice shell-ului, motiv pentru care există ca built-in separat.

Caseta „În spatele cortinei”

Aceste comenzi sunt implementate direct în Bash.

Nu există un proces extern care să fie lansat pentru executarea lor.

Acesta este unul dintre motivele pentru care sunt foarte rapide și utilizate frecvent în scripting.

Greșeală din lumea reală

Un administrator vede într-un script:

:

și presupune că este o greșeală de tipar.

În realitate, aceasta este o comandă validă Bash și este folosită intenționat pentru a nu executa nicio acțiune, păstrând în același timp sintaxa corectă.

Administrator Notes

Nu utiliza : doar pentru că ai văzut-o într-un exemplu.

Folosește-o atunci când vrei să exprimi explicit faptul că **nu este necesară nicio acțiune**, dar structura scriptului cere existența unei comenzi.

Acest lucru face codul mai clar pentru cel care îl va citi ulterior.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul comenzilor :, true și false;
 - codurile lor de ieșire;
 - faptul că sunt built-in-uri Bash.
-

Laborator 13.1

Nivel 1 – Înțelegere

Execută:

:

echo \$?

Execută apoi:

true

echo \$?

și:

```
false  
echo $?
```

Compară rezultatele.

Nivel 2 – Aplicare

Construiește un exemplu simplu în care blocul `then` conține doar:

:

Explică de ce scriptul rămâne valid.

Nivel 3 – Administrator

Analizează un script existent și identifică utilizările comenzilor:

- :
- `true`
- `false`

Explică rolul fiecăreia în contextul respectiv.

Ce trebuie să reții

- : este o comandă internă care nu execută nicio acțiune și returnează succes.
- `true` returnează întotdeauna codul de ieșire 0.
- `false` returnează întotdeauna un cod diferit de 0.
- Aceste comenzi sunt utilizate frecvent în scripting pentru controlul fluxului de execuție și pentru exprimarea explicită a intenției programatorului.

Capitolul 13

13.2 Încărcarea și executarea codului Bash (`.`, `source`, `eval`, `exec`)

„Bash oferă mai multe mecanisme pentru executarea comenzilor și a scripturilor. Deși pot părea asemănătoare, ele au efecte diferite asupra shell-ului curent și asupra proceselor create.”

Ce vei învăța

În acest capitol vei înțelege diferențele dintre comenzile `.` (dot), `source`, `eval` și `exec` și vei ști când trebuie utilizată fiecare.

Introducere

Să presupunem că există fișierul:

```
config.sh
```

care conține:

```
NUME="LPIC"
```

Există mai multe moduri prin care acest fișier poate fi utilizat.

La prima vedere par similare.

În realitate, comportamentul lor este complet diferit.

Comanda `.` (dot)

Sintaxa:

```
. fisier
```

Comanda citește și execută conținutul fișierului **în shell-ul curent**, fără a crea un proces nou.

Exemplu:

```
. config.sh
```

După execuție:

```
echo "$NUME"
```

afișează:

```
LPIC
```

Variabila rămâne disponibilă deoarece fișierul a fost executat în aceeași instanță Bash.

Comanda `source`

Sintaxa:

```
source config.sh
```

`source` este echivalentul comenzii `.` în Bash. Ambele execută fișierul în shell-ul curent.

În multe scripturi moderne vei întâlni mai frecvent forma `source`, deoarece este mai ușor de citit.

Comanda `eval`

Sintaxa:

```
eval comanda
```

`eval` nu citește un fișier.

Ea primește un șir de caractere, îl reconstruiește ca linie de comandă și îl transmite din nou interpretării Bash.

Exemplu:

```
CMD="echo Salut"
```

```
eval "$CMD"
```

Rezultat:

```
Salut
```

Comanda `exec`

Sintaxa:

```
exec comanda
```

Spre deosebire de celelalte comenzi, `exec` **nu creează un proces copil**.

Ea înlocuiește procesul Bash curent cu programul indicat.

Exemplu:

```
exec bash
```

Instanța curentă de Bash este înlocuită cu o nouă instanță.

Dacă executăm:

```
exec ls
```

după terminarea comenzii `ls`, shell-ul nu mai revine la prompt, deoarece procesul Bash inițial a fost înlocuit.

Tabel comparativ

Comandă	Creează proces nou	Modifică shell-ul curent
.	Nu	Da
source	Nu	Da
eval	Nu	Execută în shell-ul curent
exec	Nu (înlocuiește procesul curent)	Da

Caseta „În spatele cortinei”

Aceste patru comenzi sunt toate **built-in-uri Bash**.

Ele nu pot fi implementate eficient ca programe externe, deoarece trebuie să poată modifica starea shell-ului curent sau modul în care acesta execută comenzile.

Greșeală din lumea reală

Un administrator execută:

```
./config.sh
```

și se întreabă de ce variabilele definite în script dispar după terminarea acestuia.

Explicația este că scriptul rulează într-un proces separat.

Dacă scopul este modificarea mediului shell-ului curent, trebuie utilizată una dintre comenzile:

```
. config.sh
```

sau

```
source config.sh
```

Administrator Notes

Regula practică este simplă:

- folosește **source** (sau **.**) pentru a încărca configurații în shell-ul curent;
 - folosește **eval** doar atunci când este cu adevărat necesar și ai control asupra șirului executat;
 - folosește **exec** atunci când dorești înlocuirea procesului curent, de exemplu în scripturi de inițializare sau lansatoare de aplicații.
-

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- diferența dintre `.` și `source`;
 - rolul comenzii `eval`;
 - comportamentul particular al comenzii `exec`;
 - faptul că toate sunt comenzi built-in Bash.
-

Laborator 13.2

Nivel 1 – Înțelegere

Creează fișierul:

```
config.sh
```

cu conținutul:

```
NUME="LPIC"
```

Execută:

```
source config.sh
```

Verifică:

```
echo "$NUME"
```

Nivel 2 – Aplicare

Execută același fișier folosind:

```
./config.sh
```

Compară rezultatul și explică de ce variabila nu mai este disponibilă după terminarea scriptului.

Nivel 3 – Administrator

Experimentează cu:

```
exec bash
```

și explică de ce procesul Bash inițial nu mai continuă execuția după apelul `exec`.

Capitolul 13

13.3 Comanda `builtin`

„Comanda `builtin` execută o comandă internă Bash, chiar dacă există o funcție cu același nume.”

Ce vei învăța

În acest capitol vei înțelege rolul comenzii `builtin`, când este necesară și cum permite apelarea explicită a comenzilor interne Bash.

Introducere

Am învățat că Bash caută comenzile în următoarea ordine:

1. alias;
2. funcții;
3. built-in-uri;
4. executabile din PATH.

Dar apare o întrebare interesantă.

Ce se întâmplă dacă definim o funcție care are același nume ca un builtin?

Un exemplu

Definim:

```
pwd() {  
    echo "Funcția mea"  
}
```

Acum executăm:

```
pwd
```

Rezultat:

```
Funcția mea
```

Builtin-ul original nu mai este apelat.

Cum apelăm builtin-ul?

Pentru aceasta există comanda:

```
builtin
```

Exemplu:

```
builtin pwd
```

Rezultat:

```
/home/student
```

Bash ignoră funcția și execută direct builtin-ul `pwd`.

De ce este utilă?

Uneori dorim să extindem comportamentul unei comenzi interne fără să îl înlocuim complet.

Exemplu:

```
cd() {  
    echo "Schimb directorul..."  
    builtin cd "$@"  
}
```

Acum:

```
cd /tmp
```

va afișa:

```
Schimb directorul...
```

după care va executa builtin-ul `cd`.

Modelul mental

Utilizator

|

▼

cd

|

▼

Există funcție?

|



Da

|



builtin cd

|



Builtin-ul original Bash

Caseta „În spatele cortinei”

`builtin` poate executa **numai comenzi interne Bash**.

Nu poate executa:

- funcții;
- aliasuri;
- programe externe.

Scopul său este exclusiv accesul direct la implementările interne ale shell-ului.

Greșeală din lumea reală

Un administrator redefinește:

```
cd() {  
    echo "Intru în director..."  
    cd "$@"  
}
```

Rezultatul este o recursivitate infinită.

Funcția se apelează pe ea însăși.

Forma corectă este:

```
cd() {  
    echo "Intru în director..."  
    builtin cd "$@"  
}
```

Administrator Notes

Comanda `builtin` este utilă atunci când personalizezi comportamentul unor built-in-uri Bash.

Ea permite adăugarea de funcționalități fără a pierde implementarea originală.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul comenzii `builtin`;
 - faptul că execută explicit un builtin Bash;
 - utilizarea ei în funcții care suprascriu comenzi interne.
-

Laborator 13.3

Nivel 1 – Înțelegere

Definește:

```
pwd() {  
    echo "Funcție personalizată"  
}
```

Execută:

```
pwd
```

Observă rezultatul.

Nivel 2 – Aplicare

Execută:

```
builtin pwd
```

Compară rezultatul cu cel obținut anterior.

Nivel 3 – Administrator

Creează o funcție:

```
cd() {  
    echo "Mutare în: $1"  
    builtin cd "$@"  
}
```

Testează schimbarea directorului și verifică faptul că funcția adaugă comportament nou fără a modifica funcționarea builtin-ului.

Ce trebuie să reții

- `builtin` execută explicit o comandă internă Bash.
- Este util atunci când o funcție are același nume ca un builtin.
- Evită recursivitatea accidentală în funcțiile care suprascriu built-in-uri.
- Nu poate executa programe externe sau funcții, ci doar built-in-uri Bash.

Capitolul 13

13.4 Comanda `caller`

„Comanda `caller` afișează informații despre apelul curent al unei funcții Bash, inclusiv numărul liniei și fișierul din care aceasta a fost apelată.”

Ce vei învăța

În acest capitol vei înțelege rolul comenzii `caller`, când poate fi utilizată și cum ajută la depanarea funcțiilor Bash.

Introducere

Atunci când un script conține mai multe funcții, apare uneori întrebarea:

Cine a apelat funcția curentă?

În limbajele de programare există noțiunea de *call stack* (stiva apelurilor).

Bash oferă un mecanism simplificat pentru accesarea acestor informații prin comanda:

caller

Aceasta este o comandă internă Bash destinată în principal depanării și diagnosticării scripturilor.

Primul exemplu

Script:

```
functie() {  
    caller  
}
```

functie

Rezultat posibil:

```
12 main script.sh
```

Semnificația câmpurilor este:

- **12** – linia din care a fost apelată funcția;
- **main** – contextul apelului (nivelul principal al scriptului);
- **script.sh** – fișierul în care a avut loc apelul.

Valorile exacte depind de structura scriptului.

Nivelurile apelurilor

Comanda acceptă opțional un nivel numeric.

Exemplu:

```
caller 0
```

afișează apelul curent.

Dacă există mai multe funcții apelate una din alta, pot fi inspectate și nivelurile superioare folosind:

```
caller 1  
caller 2
```

și așa mai departe, atât timp cât există apeluri în stivă.

Modelul mental

main

```
|  
▼  
functie_A()  
|  
▼  
functie_B()  
|  
▼  
caller
```

`caller` permite funcției să afle cine a apelat-o și din ce locație.

Când este util?

În scripturile mici, aproape niciodată.

În schimb, în scripturi mari, cu zeci de funcții, `caller` poate fi folosit pentru:

- diagnosticarea erorilor;
 - construirea mesajelor de depanare;
 - jurnalizarea apelurilor.
-

Caseta „În spatele cortinei”

`caller` funcționează doar atunci când există un context de apel al unei funcții sau al unui script.

Dacă nu există informații disponibile pentru nivelul solicitat, comanda nu produce rezultat și întoarce un cod de eroare.

Greșeală din lumea reală

Un administrator execută:

```
caller
```

direct în terminal și se așteaptă să obțină informații.

Într-un shell interactiv, fără o funcție sau un script care să creeze un context de apel, `caller` nu are informații de afișat.

Administrator Notes

`caller` nu este o comandă utilizată zilnic.

Totuși, atunci când depanezi un script complex sau construiești o bibliotecă de funcții Bash, poate economisi mult timp prin identificarea rapidă a locului din care a fost făcut un apel.

Legătura cu LPIC

Pentru examen este suficient să cunoști:

- existența comenzii `caller`;
 - faptul că este un built-in Bash;
 - rolul său în depanarea funcțiilor și a scripturilor.
-

Laborator 13.4

Nivel 1 – Înțelegere

Creează o funcție:

```
functie() {  
    caller  
}
```

`functie`

Observă informațiile afișate.

Nivel 2 – Aplicare

Creează două funcții:

```
functie_A() {  
    functie_B  
}
```

```
functie_B() {  
    caller  
}
```

`functie_A`

Observă de unde a fost apelată `functie_B`.

Nivel 3 – Administrator

Construiește un script cu trei funcții apelate în lanț și experimentează cu:

```
caller 0  
caller 1
```

Analizează diferențele dintre informațiile afișate.

Ce trebuie să reții

- `caller` este un built-in Bash destinat depanării.
- Afișează informații despre apelul unei funcții.
- Poate inspecta mai multe niveluri ale stivei de apeluri.
- Este util în scripturi complexe și în biblioteci de funcții.

Capitolul 13

13.5 Comanda `enable`

„Comanda `enable` permite activarea sau dezactivarea comenzilor interne (built-in) ale shell-ului și, în anumite situații, încărcarea unor built-in-uri dinamice.”

Ce vei învăța

În acest capitol vei înțelege rolul comenzii `enable`, modul în care poate controla comenzile interne ale Bash și în ce situații este utilă.

Introducere

Până acum am aflat că Bash conține numeroase comenzi interne:

- `cd`
- `pwd`
- `echo`

- `read`
- `export`

Acestea sunt implementate direct în shell și nu sunt executabile externe.

Dar apare o întrebare interesantă.

Poate fi dezactivată temporar o comandă internă?

Răspunsul este:

Da, folosind comanda `enable`.

Afișarea comenzilor interne

Execută:

```
enable
```

Bash afișează lista built-in-urilor disponibile.

Aceasta este o modalitate rapidă de a vedea comenzile implementate direct în shell.

Dezactivarea unui built-in

Sintaxa generală este:

```
enable -n nume_builtin
```

Exemplu:

```
enable -n echo
```

După această comandă, Bash nu va mai utiliza built-in-ul `echo`.

Dacă în sistem există un executabil extern cu același nume, acesta va fi utilizat în schimb.

Reactivarea

Pentru reactivare:

```
enable echo
```

Builtin-ul devine din nou disponibil.

Un exemplu practic

Verificăm mai întâi:

```
type echo
```

Rezultat:

```
echo is a shell builtin
```

Dezactivăm builtin-ul:

```
enable -n echo
```

Verificăm din nou:

```
type echo
```

Dacă există un executabil extern, rezultatul poate deveni:

```
echo is /usr/bin/echo
```

Bash va utiliza acum programul extern.

Modelul mental

Comanda: echo

|

▼

Builtin activ?

|

Da → builtin echo

|

Nu

▼

Caută în PATH

|

▼

/usr/bin/echo

De ce este utilă?

În activitatea zilnică, foarte rar.

În schimb, este extrem de utilă pentru:

- depanare;
 - testarea diferențelor dintre un builtin și executabilul extern;
 - dezvoltarea și verificarea comportamentului shell-ului.
-

Built-in-uri dinamice

Documentația Bash menționează și posibilitatea încărcării sau eliminării unor built-in-uri dinamice folosind opțiunile `-f` și `-d`. Această facilitate există doar în versiunile Bash compilate cu suport pentru încărcarea dinamică și este utilizată foarte rar în administrarea obișnuită a sistemelor.

Pentru pregătirea LPIC este suficient să cunoști existența acestui mecanism.

Caseta „În spatele cortinei”

Atunci când un builtin este dezactivat, el nu este șters din Bash.

Shell-ul doar îl exclude din procesul de rezolvare a comenzilor, permițând utilizarea unui executabil extern cu același nume.

Greșeală din lumea reală

Un administrator dezactivează:

```
enable -n test
```

și observă că unele scripturi se comportă diferit.

Explicația este că Bash utilizează acum executabilul extern `test`, iar în anumite situații comportamentul poate diferi de implementarea internă.

Administrator Notes

Nu dezactiva built-in-uri într-un mediu de producție decât dacă ai un motiv bine justificat.

Comanda `enable` este destinată în principal testării, depanării și studiului funcționării shell-ului.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul comenzii `enable`;
 - diferența dintre `enable` și `enable -n`;
 - faptul că Bash poate dezactiva temporar un builtin și poate utiliza executabilul extern corespunzător, dacă acesta există.
-

Laborator 13.5

Nivel 1 – Înțelegere

Execută:

```
enable
```

Observă lista built-in-urilor disponibile.

Nivel 2 – Aplicare

Verifică:

```
type echo
```

Dezactivează builtin-ul:

```
enable -n echo
```

Execută din nou:

```
type echo
```

Compară rezultatele.

Nivel 3 – Administrator

Testează același mecanism cu un alt builtin care are și implementare externă (de exemplu `test`, dacă există în sistemul tău).

Explică modul în care Bash își schimbă algoritmul de rezolvare a comenzilor după dezactivarea builtin-ului.

Ce trebuie să reții

- `enable` afișează și controlează built-in-urile Bash.
- `enable -n` dezactivează temporar un builtin.
- `enable nume` reactivează builtin-ul.
- Dacă există un executabil extern cu același nume, Bash îl poate utiliza după dezactivarea builtin-ului intern.
- Suportul pentru built-in-uri dinamice există, dar este o funcționalitate avansată și rar întâlnită în administrarea curentă.

Capitolul 13

13.6 Comanda `help`

„Comanda `help` afișează informații despre comenzile interne (built-in) ale shell-ului și reprezintă mecanismul principal de documentare integrat în Bash.”

Ce vei învăța

În acest capitol vei înțelege rolul comenzii `help`, diferența dintre `help` și `man` și modul în care poate fi utilizată pentru consultarea rapidă a documentației comenzilor interne Bash.

Introducere

Atunci când dorim informații despre o comandă, primul reflex este adesea:

```
man comanda
```

Acest lucru funcționează foarte bine pentru programele externe, precum:

```
man ls
man grep
man find
```

Dar multe dintre comenzile pe care le-am studiat până acum nu sunt programe externe.

Ele sunt **built-in-uri Bash**.

Pentru acestea, Bash oferă propria comandă de documentare:

```
help
```

Sintaxa

Forma generală este:

```
help nume_builtin
```

Exemplu:

```
help cd
```

Bash afișează descrierea, sintaxa și opțiunile comenzii `cd`.

Documentația integrată

Spre deosebire de `man`, comanda `help` nu citește pagini de manual din sistemul de fișiere.

Informațiile sunt integrate în Bash și sunt disponibile imediat.

Acesta este motivul pentru care răspunsul este foarte rapid.

Un exemplu

Executăm:

```
help read
```

Rezultatul descrie:

- sintaxa;
- opțiunile disponibile;
- comportamentul comenzii `read`.

Este aceeași informație pe care Bash o folosește intern pentru comenzile sale.

`help` sau `man`?

Aceasta este o întrebare foarte frecventă.

Regula practică este simplă.

Comandă	Când se utilizează
<code>help</code>	pentru built-in-urile Bash

Comandă	Când se utilizează
<code>man</code>	pentru programe externe și documentația generală

Exemple:

```
help cd
help read
help export
```

și:

```
man ls
man grep
man tar
```

Verificarea tipului unei comenzi

Dacă nu știi dacă o comandă este builtin sau executabil extern, poți verifica:

```
type read
```

Rezultat:

```
read is a shell builtin
```

În acest caz, alegerea corectă este:

```
help read
```

Căutarea după subiect

Comanda `help` poate afișa și o listă de built-in-uri care corespund unui model.

Exemplu:

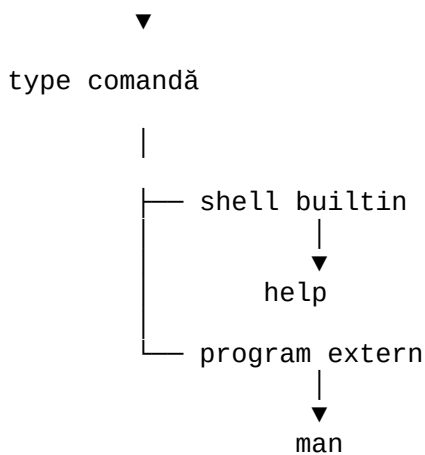
```
help read
```

sau, folosind un model, poți identifica rapid comenzi înrudite. Funcționalitatea de căutare este mai limitată decât cea oferită de `man`, dar este suficientă pentru consultarea rapidă a documentației integrate.

Modelul mental

Ai o comandă

|



Caseta „În spatele cortinei”

`help` poate documenta doar comenzile cunoscute de Bash.

Nu poate afișa documentația unui program extern precum `ls` sau `tar`, deoarece acestea nu fac parte din shell.

Greșeală din lumea reală

Un utilizator execută:

```
man cd
```

și este surprins că informațiile sunt limitate sau diferă de cele așteptate.

Explicația este că `cd` este un built-in Bash.

Documentația cea mai relevantă este:

```
help cd
```

Administrator Notes

Atunci când lucrezi într-un terminal și uiți sintaxa unei comenzi interne, încearcă mai întâi:

```
help nume_builtin
```

Este mai rapid decât deschiderea paginii `man` și conține exact informațiile necesare pentru acel builtin.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul comenzii `help`;
 - diferența dintre `help` și `man`;
 - faptul că `help` documentează built-in-urile Bash.
-

Laborator 13.6

Nivel 1 – Înțelegere

Execută:

```
help pwd
```

Apoi:

```
help export
```

Observă structura informațiilor afișate.

Nivel 2 – Aplicare

Pentru fiecare dintre comenzile de mai jos, stabilește dacă trebuie utilizat `help` sau `man`:

```
cd  
read  
export  
ls  
grep  
find
```

Verifică răspunsurile folosind și comanda:

```
type nume_comandă
```

Nivel 3 – Administrator

Alege trei built-in-uri pe care le-ai utilizat în timpul laboratorului și consultă documentația lor folosind `help`.

Identifică pentru fiecare:

- sintaxa;

- două opțiuni importante;
 - codul de ieșire (dacă este menționat).
-

Ce trebuie să reții

- `help` este sistemul de documentație integrat în Bash pentru comenzile interne.
- `man` este utilizat în principal pentru programele externe și documentația generală.
- Dacă nu știi ce tip de comandă ai în față, folosește mai întâi `type`, apoi alege între `help` și `man`.
- Consultarea rapidă a documentației direct din shell este o abilitate practică importantă pentru orice administrator Linux.

Capitolul 13

13.7 Comanda `compgen`

„Comanda `compgen` generează liste de completări folosind același mecanism intern utilizat de Bash pentru completarea comenzilor prin tasta Tab.”

Ce vei învăța

În acest capitol vei înțelege rolul comenzii `compgen`, modul în care generează liste de completări și situațiile în care poate fi utilizată pentru explorarea mediului Bash.

Introducere

Atunci când apăsăm tasta:

Tab

Bash încearcă să completeze automat ceea ce tastăm.

De exemplu:

```
ls /et<Tab>
```

poate deveni:

```
ls /etc
```

Acest comportament este implementat de mecanismul de **command completion**.

Comanda:

```
compgen
```

permite accesarea aceluiasi mecanism direct din linia de comandă.

Sintaxa

Forma generală este:

```
compgen [opțiuni]
```

În funcție de opțiuni, Bash generează diferite categorii de informații.

Listarea comenzilor

Execută:

```
compgen -c
```

Rezultatul este lista comenzilor pe care Bash le poate executa.

Lista poate include:

- aliasuri;
 - funcții;
 - built-in-uri;
 - executabile găsite în PATH.
-

Listarea aliasurilor

```
compgen -a
```

Afișează toate aliasurile definite.

Listarea funcțiilor

```
compgen -A function
```

sau, în funcție de versiunea Bash:

`compgen -A function`

Rezultatul conține toate funcțiile definite în shell-ul curent.

Listarea built-in-urilor

`compgen -b`

Afișează comenzile interne Bash.

Listarea variabilelor

`compgen -A variable`

Rezultatul conține numele tuturor variabilelor cunoscute de shell.

Listarea utilizatorilor

`compgen -u`

Generează lista utilizatorilor cunoscuți de sistem.

Listarea grupurilor

`compgen -g`

Generează lista grupurilor existente.

Modelul mental

Tab

|

▼

Motorul de completare Bash

▲

|
compgen

compgen nu implementează un mecanism diferit.

El utilizează exact motorul intern folosit de Bash pentru completarea comenzilor.

Caseta „În spatele cortinei”

compgen nu modifică nimic în sistem.

El doar generează liste de completări pe baza informațiilor disponibile în shell și în sistemul de operare.

Greșeală din lumea reală

Un administrator execută:

```
compgen -c
```

și se așteaptă să vadă doar executabilele din PATH.

În realitate, rezultatul poate include și:

- aliasuri;
- funcții;
- built-in-uri.

Acest lucru reflectă modul în care Bash rezolvă comenzile.

Administrator Notes

compgen este foarte util pentru depanare.

Dacă vrei să afli rapid:

- ce aliasuri există;
- ce funcții sunt încărcate;
- ce built-in-uri sunt disponibile;

poți genera aceste informații fără să inspecțezi manual configurația shell-ului.

Legătura cu LPIC

Pentru examen este suficient să cunoști:

- rolul comenzii `compgen`;
 - faptul că utilizează mecanismul intern de completare al Bash;
 - câteva dintre opțiunile cele mai frecvent utilizate (`-c`, `-a`, `-b`, `-u`, `-g`, `-A variable`).
-

Laborator 13.7

Nivel 1 – Înțelegere

Execută:

```
compgen -c
```

Observă că lista conține atât comenzi interne, cât și programe externe.

Nivel 2 – Aplicare

Execută pe rând:

```
compgen -a  
compgen -b  
compgen -u  
compgen -g
```

Compară tipurile de informații afișate.

Nivel 3 – Administrator

Execută:

```
compgen -A variable
```

Identifică:

- trei variabile de mediu;
- trei variabile interne Bash.

Explică diferența dintre ele.

Ce trebuie să reții

- `comp`gen utilizează mecanismul intern de completare al Bash.
- Poate genera liste de comenzi, aliasuri, funcții, built-in-uri, variabile, utilizatori și grupuri.
- Este un instrument util pentru explorarea și depanarea mediului Bash.
- Nu modifică sistemul, ci doar afișează informații disponibile în momentul execuției.

Capitolul 13

13.8 Comanda `complete`

„Comanda `complete` definește modul în care Bash realizează completarea comenzilor și a argumentelor atunci când utilizatorul apasă tasta `Tab`.”

Ce vei învăța

În acest capitol vei înțelege rolul comenzii `complete`, modul în care Bash asociază reguli de completare unei comenzi și relația dintre `complete` și `comp`gen.

Introducere

În capitolul anterior am învățat că:

`comp`gen

poate genera liste de completări utilizând mecanismul intern al Bash.

Dar apare o întrebare firească.

Cum știe Bash ce trebuie să completeze atunci când apăsăm tasta `Tab`?

Răspunsul este:

prin regulile definite cu ajutorul comenzii:

`complete`

Rolul comenzii `complete`

`complete` nu generează completări.

Ea definește regulile după care Bash va genera completările pentru o anumită comandă.

Cu alte cuvinte:

- `compgen` produce rezultatele;
 - `complete` stabilește cum vor fi obținute acele rezultate.
-

Afișarea regulilor existente

Execută:

```
complete -p
```

Bash afișează toate regulile de completare definite în sesiunea curentă.

Pe multe distribuții Linux vei observa că lista este destul de lungă, deoarece este încărcată automat de pachetul **bash-completion**.

Definirea unei reguli simple

O regulă poate fi asociată unei comenzi.

Exemplu:

```
complete -w "start stop restart status" serviciu
```

Acum, după tastarea:

```
serviciu <Tab>
```

Bash va propune una dintre valorile:

- `start`
- `stop`
- `restart`
- `status`

Acest exemplu ilustrează mecanismul descris în `man bash`: o listă de cuvinte este utilizată pentru completare.

Eliminarea unei reguli

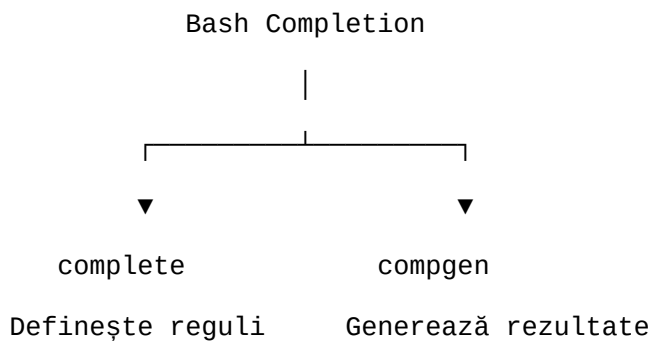
O regulă poate fi eliminată folosind:

```
complete -r serviciu
```

După această comandă, Bash nu mai utilizează regula definită anterior pentru comanda `serviciu`.

Relația cu `compgen`

Cele două comenzi sunt complementare.



În practică, multe definiții de completare utilizează intern `compgen` pentru a produce lista de sugestii.

Caseta „În spatele cortinei”

Comanda `complete` modifică doar comportamentul completării automate din shell-ul curent.

Ea nu schimbă funcționarea comenzilor și nu modifică programele externe.

Greșeală din lumea reală

Un administrator definește o regulă cu `complete` și observă că aceasta dispare după închiderea terminalului.

Acest comportament este normal.

Regulile create interactiv există doar pe durata sesiunii Bash. Pentru a le păstra permanent, ele trebuie încărcate dintr-un fișier de inițializare, cum ar fi `~/ .bashrc` sau prin infrastructura `bash-completion`.

Administrator Notes

În administrarea curentă vei utiliza rar `complete` direct.

Cel mai des vei beneficia de regulile deja furnizate de distribuția Linux prin pachetul `bash-completion`.

Totuși, atunci când scrii utilitare proprii sau scripturi complexe, definirea unei completări personalizate poate îmbunătăți considerabil experiența utilizatorului.

Legătura cu LPIC

Pentru examen este suficient să cunoști:

- rolul comenzii `complete`;
 - diferența dintre `complete` și `compgen`;
 - faptul că Bash poate defini reguli personalizate pentru completarea comenzilor.
-

Laborator 13.8

Nivel 1 – Înțelegere

Afișează regulile existente:

```
complete -p
```

Observă numărul și tipul regulilor definite.

Nivel 2 – Aplicare

Definește o regulă simplă:

```
complete -w "start stop restart status" serviciu
```

Tastează:

```
serviciu
```

urmat de **Tab** și observă sugestiile oferite de Bash.

Nivel 3 – Administrator

Elimină regula:

```
complete -r serviciu
```

Verifică faptul că sugestiile nu mai sunt disponibile.

Dacă distribuția ta utilizează `bash-completion`, compară regulile existente înainte și după definirea propriei reguli.

Ce trebuie să reții

- `complete` definește regulile de completare automată pentru Bash.
- `compgen` generează efectiv rezultatele utilizate la completare.
- Regulile definite interactiv sunt valabile doar în sesiunea curentă.
- `complete -p` afișează regulile existente, iar `complete -r` elimină o regulă.

Capitolul 13

13.9 Comanda `compopt`

„Comanda `compopt` modifică opțiunile unei specificații de completare existente sau ale funcției de completare aflate în execuție.”

Ce vei învăța

În acest capitol vei înțelege rolul comenzii `compopt`, relația sa cu `complete` și situațiile în care este utilizată pentru personalizarea comportamentului completării automate.

Introducere

În capitolul anterior am învățat că:

`complete`

definește regulile după care Bash realizează completarea automată.

Dar uneori apare o nouă cerință.

Regula există deja, însă dorim să schimbăm modul în care Bash se comportă în timpul completării.

Pentru această situație există comanda:

`compopt`

Rolul comenzii compopt

Spre deosebire de:

complete

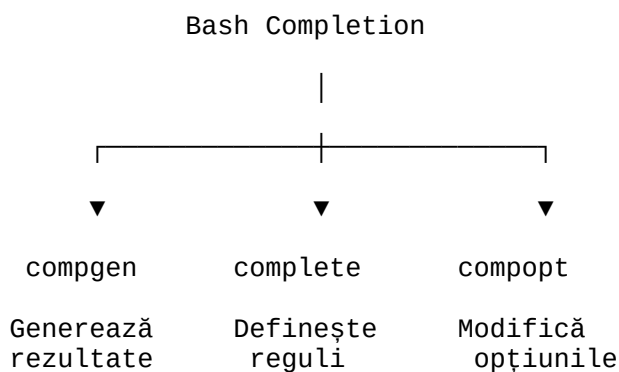
care definește o regulă nouă,

compopt

modifică opțiunile unei reguli existente.

Relația dintre cele trei comenzi

Acum putem vedea imaginea completă.



Fiecare comandă are un rol diferit.

Utilizarea în practică

În majoritatea cazurilor, administratorii folosesc:

complete

pentru definirea unei completări.

compopt apare mai ales în funcțiile complexe de completare, unde comportamentul trebuie ajustat dinamic în funcție de context.

De ce este utilă?

Să presupunem că o funcție de completare detectează două situații diferite:

- utilizatorul completează un nume de fișier;

- utilizatorul completează un nume de serviciu.

În timpul execuției, funcția poate modifica opțiunile completării folosind:

`compopt`

astfel încât Bash să adapteze comportamentul fără redefinirea întregii reguli.

Modelul mental

Apăsare Tab

|



Regulă definită?

|



complete

|



Este nevoie de ajustări?

|



`compopt`

|



Sugestiile sunt afișate

Caseta „În spatele cortinei”

`compopt` este utilizată aproape exclusiv împreună cu funcții de completare personalizate.

Ea nu generează sugestii și nu creează reguli noi.

Rolul ei este exclusiv modificarea comportamentului unei completări deja existente.

Greșeală din lumea reală

Un administrator încearcă să creeze o completare automată folosind doar:

```
compopt
```

și observă că nu se întâmplă nimic.

Explicația este simplă.

`compopt` presupune existența unei reguli de completare. Dacă aceasta nu există, nu are ce modifica.

Administrator Notes

În administrarea obișnuită a sistemelor Linux este puțin probabil să utilizezi direct `compopt`.

Totuși, dacă dezvolti scripturi complexe sau personalizezi mecanismul de completare Bash, este important să știi că această comandă există și că completează funcționalitatea oferită de `complete`.

Legătura cu LPIC

Pentru examen este suficient să cunoști:

- rolul comenzii `compopt`;
 - diferența dintre `compgen`, `complete` și `compopt`;
 - faptul că `compopt` modifică opțiunile unei completări existente.
-

Laborator 13.9

Nivel 1 – Înțelegere

Consultă documentația integrată:

```
help compopt
```

Identifică scopul principal al comenzii.

Nivel 2 – Aplicare

Examinează un script de completare din sistemul tău (dacă este instalat pachetul `bash-completion`) și identifică eventualele apeluri ale comenzii:

`compopt`

Observă în ce context sunt utilizate.

Nivel 3 – Administrator

Compară rolurile comenzilor:

`compgen`
`complete`
`compopt`

Explică, pentru fiecare, dacă:

- generează rezultate;
 - definește reguli;
 - modifică opțiuni.
-

Ce trebuie să reții

- `compgen` generează liste de completări.
- `complete` definește regulile de completare.
- `compopt` modifică opțiunile unei completări existente.
- Cele trei comenzi formează împreună infrastructura de completare automată a Bash.

Capitolul 13

13.10 Comanda `fc` (Fix Commands)

„Comanda `fc` permite listarea, editarea și reexecutarea comenzilor din istoricul Bash.”

Ce vei învăța

În acest capitol vei înțelege rolul comenzii `fc`, diferența dintre `history` și `fc` și modul în care poți corecta și reexecuta rapid comenzile din istoricul shell-ului.

Introducere

Până acum am utilizat istoricul comenzilor prin:

- săgețile ↑ și ↓;
- comanda:

`history`

Aceste mecanisme sunt suficiente pentru a regăsi comenzile executate anterior.

Dar apare o situație frecventă.

Ai executat o comandă foarte lungă și ai observat că ai greșit un singur argument.

Să o rescrii de la zero este inefficient.

Pentru astfel de situații există comanda:

`fc`

Rolul comenzii `fc`

`fc` lucrează cu istoricul comenzilor.

Ea poate:

- lista comenzile din istoric;
 - deschide comenzile într-un editor;
 - executa din nou comenzile după modificare.
-

Listarea comenzilor

Forma:

`fc -l`

afișează comenzile din istoric.

Exemplu:

```
102  ls -l
103  cd /etc
104  cat passwd
105  grep root /etc/passwd
```

Opțiunea - l transformă fc într-un instrument de listare.

Selectarea unui interval

Se poate afișa un interval de comenzi.

Exemplu:

```
fc -l -10
```

afișează ultimele zece comenzi din istoric.

De asemenea, pot fi specificate două limite:

```
fc -l 100 110
```

pentru afișarea comenzilor cuprinse în acel interval.

Editarea unei comenzi

Dacă opțiunea - l nu este utilizată, fc deschide comenzile selectate într-un editor de text.

După salvarea și închiderea editorului, comenzile modificate sunt executate automat.

Editorul utilizat este ales în următoarea ordine:

1. valoarea variabilei FCEDIT;
 2. dacă aceasta nu este definită, valoarea variabilei EDITOR;
 3. dacă nici aceasta nu este definită, editorul implicit este vi.
-

Reexecutarea rapidă

O formă foarte utilă este:

```
fc -s
```

Aceasta execută din nou ultima comandă din istoric.

Poate fi utilizată și pentru înlocuirea rapidă a unui text.

Exemplu:

```
fc -s vechi=nou
```

Bash înlocuiește prima apariție a textului **vechi** cu **nou** și execută comanda rezultată.

Un exemplu practic

Ai executat:

```
grep rooot /etc/passwd
```

Observi că ai scris greșit `root`.

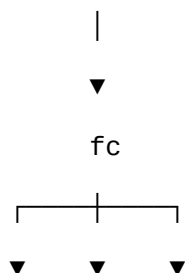
În loc să rescrii întreaga comandă, poți utiliza:

```
fc -s rooot=root
```

Bash reconstruiește comanda cu textul corectat și o execută imediat.

Modelul mental

Istoric Bash



Listare Editare Reexecutare

Caseta „În spatele cortinei”

Comanda `fc` operează asupra aceluiași istoric utilizat de mecanismul `history`.

Ea nu păstrează un istoric separat, ci oferă funcții suplimentare pentru editarea și reutilizarea comenzilor existente.

Greșeală din lumea reală

Un administrator execută:

```
fc
```

și este surprins că se deschide editorul `vi`.

Acesta nu este un comportament accidental.

Dacă variabilele FCEDIT și EDITOR nu sunt definite, Bash utilizează implicit vi.

Administrator Notes

fc este deosebit de utilă atunci când lucrezi cu:

- comenzi foarte lungi;
- expresii regulate complexe;
- comenzi SQL;
- apeluri find, tar, rsync sau ssh cu numeroase opțiuni.

Corectarea unei singure litere este adesea mult mai rapidă decât rescrierea întregii comenzi.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul comenzii fc;
 - diferența dintre history și fc;
 - utilizarea opțiunilor -l și -s;
 - rolul variabilei FCEDIT.
-

Laborator 13.10

Nivel 1 – Înțelegere

Execută câteva comenzi simple:

```
pwd
ls
date
whoami
```

Apoi afișează istoricul folosind:

```
fc -l
```

Compară rezultatul cu:

```
history
```

Nivel 2 – Aplicare

Execută intenționat o comandă cu o greșeală minoră:

```
grep rooot /etc/passwd
```

Corectează-o folosind:

```
fc -s rooot=root
```

Observă că Bash execută automat varianta corectată.

Nivel 3 – Administrator

Configurează temporar editorul preferat:

```
export FCEDIT=nano
```

(sau alt editor disponibil în sistem)

Execută:

```
fc
```

Modifică ultima comandă în editor, salvează fișierul și observă execuția automată după închiderea editorului.

Ce trebuie să reții

- `fc` este un built-in destinat lucrului cu istoricul comenzilor.
- Poate lista, edita și reexecuta comenzile din istoric.
- `fc -l` listează comenzile, iar `fc -s` permite reexecutarea rapidă cu sau fără înlocuirea unui text.
- Editorul utilizat este determinat de `FCEDIT`, apoi de `EDITOR`, iar în lipsa acestora de `vi`.

Capitolul 13

13.11 Comanda `getopts`

„Comanda `getopts` este utilizată de scripturile Bash pentru interpretarea opțiunilor și argumentelor transmise în linia de comandă.”

Ce vei învăța

În acest capitol vei înțelege rolul comenzii `getopts`, modul în care interpretează opțiunile unui script și avantajele sale față de analiza manuală a argumentelor.

Introducere

Să presupunem că scriem un script:

```
backup.sh
```

și dorim să îl apelăm astfel:

```
backup.sh -v -f arhiva.tar
```

unde:

- `-v` activează modul verbose;
- `-f` primește numele fișierului de arhivă.

Cum poate scriptul să înțeleagă aceste opțiuni?

Răspunsul este:

```
getopts
```

Rolul comenzii `getopts`

`getopts` analizează argumentele primite de un script și identifică opțiunile definite de programator.

Spre deosebire de verificarea manuală a variabilelor `$1`, `$2`, `$3`, `getopts` tratează automat opțiunile și argumentele asociate acestora.

Sintaxa generală

Forma de bază este:

```
getopts optstring variabila
```

unde:

- `optstring` definește opțiunile acceptate;
- `variabila` primește opțiunea identificată la fiecare apel.

Ce este `optstring`?

`optstring` este un șir de caractere.

Fiecare caracter reprezintă o opțiune acceptată.

Exemplu:

`vf:`

Semnificație:

- `v` este o opțiune simplă;
- `f` : este o opțiune care trebuie să primească un argument.

Caracterul `:` după o literă indică faptul că acea opțiune necesită o valoare.

Un exemplu complet

```
while getopts "vf:" opt
do
    case "$opt" in
        v)
            echo "Mod verbose activ"
            ;;
        f)
            echo "Fișier: $OPTARG"
            ;;
    esac
done
```

Dacă scriptul este executat astfel:

```
./backup.sh -v -f arhiva.tar
```

rezultatul va fi:

```
Mod verbose activ
Fișier: arhiva.tar
```

Variabila `OPTARG`

Atunci când o opțiune are un argument, acesta este disponibil în variabila:

`OPTARG`

În exemplul anterior:

OPTARG = arhiva.tar

Această variabilă este gestionată automat de `getopts`.

Variabila OPTIND

Bash menține și variabila:

OPTIND

Ea indică poziția următorului argument care urmează să fie analizat.

După terminarea buclei `getopts`, această variabilă este folosită frecvent împreună cu:

```
shift $((OPTIND - 1))
```

pentru a elimina opțiunile deja procesate și a lăsa doar argumentele rămase.

Modelul mental

Argumente

```
-v -f arhiva.tar documente
```

|

▼

getopts

|

```
| -v  
| -f  
| arhiva.tar
```

|

▼

Argument rămas

documente

Caseta „În spatele cortinei”

`getopts` nu modifică argumentele scriptului.

El doar le parcurge și actualizează intern variabile precum `OPTIND` și `OPTARG`.

Greșeală din lumea reală

Un administrator utilizează:

```
while getopts "vf:" opt
```

dar uită să execute ulterior:

```
shift $((OPTIND - 1))
```

Rezultatul este că argumentele neprocesate rămân deplasate și scriptul interpretează incorect parametrii rămași.

Administrator Notes

Pentru scripturile care acceptă opțiuni în stil Unix (`-a`, `-v`, `-f`, `-o`), `getopts` este metoda recomandată.

El produce cod mai clar, mai robust și mai ușor de întreținut decât verificarea manuală a variabilelor `$1`, `$2`, `$3`.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul comenzii `getopts`;
 - semnificația șirului `optstring`;
 - utilizarea variabilelor `OPTARG` și `OPTIND`;
 - faptul că caracterul `:` indică o opțiune care necesită un argument.
-

Laborator 13.11

Nivel 1 – Înțelegere

Creează un script care acceptă:

- `-v`
- `-f nume_fișier`

și afișează valorile primite.

Nivel 2 – Aplicare

Execută scriptul folosind:

```
./script.sh -v -f test.txt
```

și apoi:

```
./script.sh -f backup.tar
```

Observă modul în care `getopts` identifică opțiunile și argumentele lor.

Nivel 3 – Administrator

Extinde scriptul astfel încât să accepte și opțiunea:

```
-h
```

pentru afișarea unui mesaj de ajutor.

După terminarea buclei `getopts`, folosește:

```
shift $((OPTIND - 1))
```

și afișează argumentele rămase, dacă există.

Ce trebuie să reții

- `getopts` este mecanismul standard Bash pentru interpretarea opțiunilor din linia de comandă.
- `optstring` definește opțiunile acceptate.
- `OPTARG` conține argumentul asociat unei opțiuni.
- `OPTIND` indică următorul argument care urmează să fie procesat.
- Pentru scripturile profesionale, `getopts` este preferabil analizei manuale a parametrilor.

Capitolul 13

13.12 Comenzile `mapfile` și `readarray`

„Comenzile `mapfile` și `readarray` citesc linii de intrare și le încarcă într-un tablou indexat (*indexed array*).”

Ce vei învăța

În acest capitol vei înțelege rolul comenzilor `mapfile` și `readarray`, avantajele lor față de citirea linie cu linie folosind `read` și principalele opțiuni disponibile.

Introducere

Până acum am utilizat frecvent:

```
read
```

pentru a citi o singură linie de la tastatură sau dintr-un fișier.

Dar apare o situație diferită.

Cum citim rapid toate liniile unui fișier într-un tablou Bash?

În locul unei bucle:

```
while read linie
do
    ...
done
```

Bash oferă o soluție directă:

```
mapfile
```

sau, echivalent:

```
readarray
```

Cele două denumiri sunt sinonime.

Rolul comenzilor

`mapfile` citește toate liniile disponibile și le salvează într-un tablou indexat.

Dacă nu este specificat numele tabloului, Bash utilizează implicit variabila:

```
MAPFILE
```

Aceasta este creată automat atunci când este necesar.

Primul exemplu

Presupunem că fișierul:

```
utilizatori.txt
```

conține:

```
ana  
mihai  
elena
```

Executăm:

```
mapfile utilizatori < utilizatori.txt
```

Rezultatul este:

```
echo "${utilizatori[0]}"
```

```
ana
```

și:

```
echo "${utilizatori[2]}"
```

```
elena
```

Fiecare linie a devenit un element al tabloului.

Variabila implicită MAPFILE

Dacă nu indicăm un nume de tablou:

```
mapfile < utilizatori.txt
```

Bash utilizează automat:

```
MAPFILE
```

Exemplu:

```
echo "${MAPFILE[1]}"
```

va afișa a doua linie din fișier.

Opțiunea -t

În mod implicit, fiecare element păstrează delimitatorul de sfârșit de linie.

Pentru eliminarea acestuia se utilizează:

```
mapfile -t utilizatori < utilizatori.txt
```

Opțiunea `-t` elimină delimitatorul (implicit caracterul newline) din fiecare element citit.

Aceasta este și opțiunea utilizată cel mai frecvent în practică.

Citirea unui număr limitat de linii

Pentru a citi doar primele n linii:

```
mapfile -n 5 utilizatori < utilizatori.txt
```

Vor fi încărcate cel mult cinci elemente în tablou.

Ignorarea primelor linii

Pentru a omite un număr de linii de la începutul fișierului:

```
mapfile -s 1 -t utilizatori < utilizatori.txt
```

În acest exemplu, prima linie este ignorată, iar citirea începe cu a doua.

Citirea dintr-un descriptor de fișier

În mod normal, `mapfile` citește de la intrarea standard.

Cu opțiunea:

```
-u fd
```

citirea se face din descriptorul de fișier indicat.

Această facilitate este utilă în scripturi mai complexe care utilizează mai multe fluxuri de intrare.

Modelul mental

Fișier text

```
ana  
mihai  
elena
```

|



```
mapfile -t utilizatori
```

|



```
utilizatori[0]="ana"
```

```
utilizatori[1]="mihai"
```

```
utilizatori[2]="elena"
```

Caseta „În spatele cortinei”

Spre deosebire de:

```
while read
```

care procesează câte o linie la fiecare iterație, `mapfile` citește rapid toate liniile și construiește direct un tablou Bash.

Acest lucru simplifică scripturile atunci când întregul conținut trebuie păstrat în memorie.

Greșeală din lumea reală

Un administrator execută:

```
mapfile utilizatori < utilizatori.txt
```

și observă că fiecare element pare să conțină un caracter suplimentar la sfârșit.

Motivul este că delimitatorul de sfârșit de linie este păstrat.

În majoritatea cazurilor, soluția este:

```
mapfile -t utilizatori < utilizatori.txt
```

Administrator Notes

Folosește `mapfile -t` atunci când ai nevoie să încarci rapid conținutul unui fișier într-un tablou.

Dacă fișierul este foarte mare și nu este necesar să păstrezi toate liniile simultan în memorie, o buclă `while read` poate fi o alegere mai potrivită.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- că `mapfile` și `readarray` sunt sinonime;
 - rolul variabilei implicite `MAPFILE`;
 - utilizarea opțiunilor:
 - `-t`
 - `-n`
 - `-s`
 - `-u`
-

Laborator 13.12

Nivel 1 – Înțelegere

Creează fișierul:

`servere.txt`

cu conținutul:

```
web01
db01
mail01
```

Încarcă-l folosind:

```
mapfile -t servere < servere.txt
```

Afișează apoi:

```
printf "%s\n" "${servere[@]}"
```

Nivel 2 – Aplicare

Repetă exercițiul fără opțiunea `-t`.

Compară conținutul elementelor și explică diferența observată.

Nivel 3 – Administrator

Testează:

```
mapfile -n 2 -t servere < servere.txt
```

și apoi:

```
mapfile -s 1 -t servere < servere.txt
```

Explică modul în care opțiunile `-n` și `-s` influențează rezultatul.

Ce trebuie să reții

- `mapfile` și `readarray` reprezintă aceeași comandă.
- Ele citesc linii și le încarcă într-un tablou indexat.
- Dacă nu este specificat un nume de tablou, Bash utilizează variabila `MAPFILE`.
- Opțiunea `-t` elimină delimitatorul de sfârșit de linie și este cea mai utilizată în practică.
- `-n`, `-s` și `-u` permit controlul modului în care sunt citite datele.

Capitolul 13

13.13 Coprocesele (coproc)

„Un coprocess este o comandă a shell-ului precedată de cuvântul rezervat `coproc`. Ea este executată asincron într-un subshell, iar între shell și coprocess este creat automat un canal de comunicație bidirecțional.”

Ce vei învăța

În acest capitol vei înțelege ce este un coprocess, cum îl creează Bash și cum permite schimbul de date în ambele sensuri între shell și procesul lansat.

Introducere

Până acum am studiat:

- procesele;
- procesele în fundal (`&`);

- conductele (|);
- redirectionările.

Toate acestea permit executarea și comunicarea între procese.

Dar Bash oferă și un mecanism special:

`coproc`

Acesta creează un proces care rulează în fundal și stabilește automat **două canale de comunicație**:

- unul pentru trimiterea datelor către proces;
- unul pentru citirea răspunsurilor generate de acesta.

Sintaxa

Forma generală este:

`coproc [NUME] comandă [redirectionări]`

Dacă nu este specificat un nume, Bash utilizează implicit:

`COPROC`

ca nume al coprocesului.

Primul exemplu

Pornim:

`coproc cat`

Bash creează un coprocess care execută comanda:

`cat`

și stabilește automat două fluxuri de comunicație.

Nu este nevoie să construim manual conducte sau FIFO-uri.

Variabila asociată

La crearea unui coprocess, Bash definește un tablou cu același nume.

Dacă folosim:

`coproc COP {`

```
} cat
```

Bash creează tabloul:

`COP`

în care:

- `COP[0]` reprezintă descriptorul de fișier din care shell-ul citește ieșirea coprocesului;
 - `COP[1]` reprezintă descriptorul către care shell-ul trimite date către coproces.
-

PID-ul coprocesului

Bash pune la dispoziție și identificatorul procesului.

Pentru un coprocess numit:

`COP`

PID-ul este disponibil în variabila:

`COP_PID`

Această variabilă este creată automat de Bash.

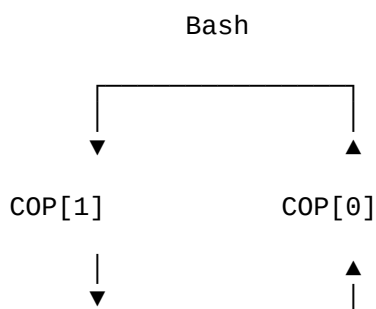
Așteptarea terminării

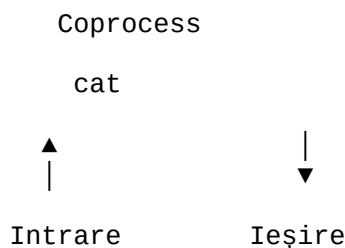
După lansarea coprocesului, putem utiliza:

`wait`

pentru a aștepta terminarea execuției acestuia. Documentația menționează explicit că `wait` poate fi folosit în acest scop.

Modelul mental





Spre deosebire de o conductă simplă (|), comunicarea este **bidirecțională**.

De ce există coproc?

Într-o conductă clasică:

```
echo test | cat
```

datele circulă într-o singură direcție.

În schimb, un coprocess poate primi date și poate răspunde în timpul execuției.

Acest model este util atunci când un proces trebuie să rămână activ și să proceseze mai multe cereri succesive.

Caseta „În spatele cortinei”

Conform documentației, coprocess-ul este executat **asincron**, într-un **subshell**, ca și cum comanda ar fi fost lansată folosind operatorul &. În plus, Bash creează automat cele două conducte necesare comunicării bidirecționale și le asociază descriptorilor de fișier din tabloul coprocesului.

Greșeală din lumea reală

Un administrator presupune că:

```
coproc
```

este doar o variantă mai complicată a operatorului:

```
&
```

În realitate, diferența este esențială.

Operatorul & doar execută procesul în fundal.

`coproc` execută procesul în fundal și creează automat infrastructura necesară schimbului bidirecțional de date.

Administrator Notes

Majoritatea administratorilor Linux nu folosesc zilnic `coproc`.

Totuși, atunci când trebuie să comunici repetat cu un proces extern (de exemplu un interpret, un utilitar interactiv sau un serviciu care citește de la intrarea standard), `coproc` poate simplifica semnificativ scriptul față de gestionarea manuală a descriptorilor de fișier.

Legătura cu LPIC

Pentru examen este suficient să cunoști:

- rolul cuvântului rezervat `coproc`;
 - faptul că execută comanda asincron într-un subshell;
 - existența tabloului cu descriptorii de fișier;
 - existența variabilei `NAME_PID`;
 - utilizarea comenzii `wait` pentru a aștepta terminarea coprocesului.
-

Laborator 13.13

Nivel 1 – Înțelegere

Lansează un coprocess simplu:

```
coproc cat
```

Verifică existența variabilelor:

```
declare -p COPROC  
echo "$COPROC_PID"
```

(dacă ai folosit numele implicit).

Nivel 2 – Aplicare

Creează un coprocess cu un nume explicit:

```
coproc CANAL {
```



```
} cat
```

Identifică:

- descriptorul de citire (CANAL[0]);
 - descriptorul de scriere (CANAL[1]);
 - PID-ul (CANAL_PID).
-

Nivel 3 – Administrator

Folosește `wait` pentru a sincroniza terminarea coprocesului și explică de ce acesta rulează într-un subshell și de ce comunicarea este bidirecțională.

Ce trebuie să reții

- `coproc` este un **cuvânt rezervat Bash**, nu un program extern.
- El execută o comandă asincron într-un subshell.
- Bash creează automat două canale de comunicație între shell și coprocess.
- Descriptorii de fișier sunt disponibili într-un tablou asociat coprocesului.
- PID-ul este disponibil în variabila `NAME_PID`.
- `wait` poate fi utilizat pentru sincronizarea cu terminarea coprocesului.

Capitolul 14

14.1 Definirea funcțiilor Bash (Shell Function Definitions)

„O funcție Bash este un obiect care poate fi apelat asemenea unei comenzi simple și care execută un bloc de comenzi folosind un nou set de parametri poziționali.”

Ce vei învăța

În acest capitol vei înțelege ce este o funcție Bash, cum se definește corect și prin ce se deosebește de un script.

Introducere

Pe măsură ce scripturile cresc, apare inevitabil aceeași problemă.

Același grup de comenzi trebuie executat în mai multe locuri.

Prima soluție este copierea codului.

Aceasta funcționează la început, dar devine greu de întreținut.

Bash oferă o soluție mai elegantă:

funcțiile.

O funcție permite gruparea mai multor comenzi sub un singur nume, astfel încât acestea să poată fi apelate ori de câte ori este nevoie.

Ce este o funcție?

Conform documentației Bash, o funcție este un obiect care:

- este apelat asemenea unei comenzi simple;
- execută un bloc de comenzi;
- primește propriul set de parametri poziționali (\$1, \$2, ...).

Acest ultim aspect este foarte important.

Funcția nu modifică parametrii poziționali ai scriptului apelant, ci lucrează cu propriul set de argumente pe durata execuției.

Prima formă de definire

Sintaxa clasică este:

```
nume_functie() {  
    comenzi  
}
```

Exemplu:

```
salut() {  
    echo "Salut!"  
}
```

Apelul funcției se face exact ca în cazul unei comenzi:

```
salut
```

A doua formă de definire

Bash acceptă și sintaxa:

```
function nume_functie {  
    comenzi  
}
```

sau:

```
function nume_functie() {  
    comenzi  
}
```

Cuvântul rezervat:

```
function
```

este **opțional**.

Parantezele și acoladele

Există un detaliu subtil.

Dacă este utilizat cuvântul rezervat:

```
function
```

parantezele () devin opționale.

În schimb, dacă acestea lipsesc, corpul funcției trebuie delimitat prin acolade {}. Acest comportament este descris explicit în `man bash`.

Funcția este o comandă

După definire:

```
backup() {  
    echo "Backup pornit"  
}
```

Bash tratează:

```
backup
```

ca pe orice altă comandă.

Funcția participă la mecanismul de rezolvare a comenzilor al shell-ului, alături de aliasuri, built-in-uri și executabile.

Parametrii funcției

O funcție poate primi argumente:

```
salut() {  
    echo "Salut, $1!"  
}
```

Apel:

```
salut Daniel
```

Rezultat:

```
Salut, Daniel!
```

În interiorul funcției:

- \$1 este Daniel;
- \$# este 1;
- \$@ conține toate argumentele funcției.

Acești parametri există doar pe durata execuției funcției.

Redirecționări la definire

Documentația Bash precizează că o funcție poate avea redirecționări atașate definiției sale:

```
functie() {  
    echo "Mesaj"  
} > jurnal.log
```

Aceste redirecționări **nu sunt executate în momentul definirii**, ci atunci când funcția este apelată.

Acesta este un detaliu pe care mulți utilizatori nu îl cunosc.

Codul de ieșire

Definirea unei funcții se încheie cu succes (0), dacă sintaxa este corectă și nu există o funcție protejată (readonly) cu același nume.

La executare, funcția întoarce codul de ieșire al **ultimei comenzi executate** din corpul ei.

Exemplu:

```
verifica() {  
    ls /tmp
```

```
}
```

Codul întors de funcție este exact codul întors de comanda `ls`.

Modelul mental

Apel

`backup documente`

|



Bash găsește funcția

|



Se creează un nou set
de parametri poziționali

|



Se execută corpul funcției

|



Se întoarce codul
ultimei comenzi

Caseta „În spatele cortinei”

O funcție Bash **nu creează automat un proces nou**.

Ea este executată în contextul shell-ului curent, motiv pentru care poate modifica variabilele, directorul curent și alte elemente ale mediului shell-ului. Aceasta este una dintre diferențele fundamentale dintre o funcție și executarea unui script ca proces separat.

Greșeală din lumea reală

Un administrator mută un grup de comenzi într-o funcție și presupune că variabilele modificate în interior vor dispărea după terminarea funcției.

În realitate, funcția rulează în același proces Bash.

Dacă nu sunt declarate locale, variabilele modificate în funcție rămân vizibile și după încheierea acesteia.

Vom studia mecanismul `local` într-un capitol dedicat.

Administrator Notes

În scripturile mici este tentant să scrii tot codul liniar.

Începând cu 30-40 de linii, funcțiile devin aproape obligatorii.

Ele îmbunătățesc:

- lizibilitatea;
 - reutilizarea codului;
 - depanarea;
 - testarea.
-

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- cele două forme de definire a unei funcții;
 - faptul că `function` este opțional;
 - că funcțiile folosesc un nou set de parametri poziționali;
 - că redirectionările asociate definiției sunt aplicate la execuție;
 - că funcția întoarce codul ultimei comenzi executate.
-

Laborator 14.1

Nivel 1 – Înțelegere

Definește funcția:

```
salut() {  
    echo "Salut!"  
}
```

Execut-o de mai multe ori și observă că poate fi apelată asemenea unei comenzi.

Nivel 2 – Aplicare

Modifică funcția pentru a primi un argument:

```
salut() {  
    echo "Salut, $1!"  
}
```

Testează:

```
salut Maria  
salut Andrei
```

Observă modul în care se modifică valoarea lui \$1.

Nivel 3 – Administrator

Creează funcția:

```
backup() {  
    tar -czf "$1.tar.gz" "$1"  
}
```

Execut-o pentru două directoare diferite și verifică faptul că aceeași funcție poate fi reutilizată fără modificări.

Ce trebuie să reții

- O funcție Bash este apelată asemenea unei comenzi simple.
- Ea execută un bloc de comenzi folosind propriul set de parametri poziționali.
- `function` este un cuvânt rezervat opțional.
- Funcțiile rulează în shell-ul curent și pot modifica starea acestuia.
- Codul de ieșire al funcției este codul ultimei comenzi executate în corpul ei.

Capitolul 14

14.2 Variabile locale – comanda `local`

„Comanda `local` creează variabile a căror vizibilitate este limitată la funcția curentă și la funcțiile apelate de aceasta.”

Ce vei învăța

În acest capitol vei înțelege diferența dintre variabilele globale și cele locale, modul de funcționare al comenzii `local` și de ce aceasta este esențială pentru scrierea unor funcții robuste.

Introducere

În capitolul anterior am aflat că o funcție Bash este executată în shell-ul curent.

Acest lucru are un avantaj important:

- funcția poate modifica variabilele existente.

Dar are și un dezavantaj.

O funcție poate modifica accidental variabile utilizate în alte părți ale scriptului.

Pentru a evita această problemă, Bash oferă comanda:

```
local
```

Variabile globale

Să presupunem:

```
NUME="Daniel"

functie() {
    NUME="Maria"
}

functie

echo "$NUME"
```

Rezultatul este:

```
Maria
```

Funcția a modificat variabila globală deoarece rulează în același shell.

Variabile locale

Același exemplu poate fi scris astfel:

```
NUME="Daniel"
```



```
functie() {  
    local NUME="Maria"  
}
```

```
functie
```

```
echo "$NUME"
```

Rezultatul este:

Daniel

Variabila locală există doar în interiorul funcției.

Domeniul de vizibilitate (Scope)

Conform documentației Bash:

atunci când `local` este utilizată într-o funcție, variabila este vizibilă doar în acea funcție și în funcțiile apelate de aceasta.

Acest concept poartă numele de **scope** (domeniu de vizibilitate).

Modelul este:

```
functie_A
```

```
|
```

```
▼
```

```
local X
```

```
|
```

```
▼
```

```
functie_B
```

```
|
```

```
▼
```

```
X este vizibilă
```

După întoarcerea
din functie_A

X dispare

Inițializarea

O variabilă locală poate fi creată fără valoare:

```
local nume
```

sau poate fi inițializată imediat:

```
local nume="Daniel"
```

Ambele forme sunt acceptate.

local și declare

Documentația precizează că `local` acceptă aceleași opțiuni ca și comanda:

```
declare
```

De exemplu:

```
local -i contor=10
```

crează o variabilă locală cu atributul întreg (`integer`).

Un detaliu foarte puțin cunoscut

`local` poate primi și argumentul:

```
local -
```

În acest caz, Bash face **locale opțiunile shell-ului** modificate prin `set`.

La ieșirea din funcție, acestea sunt restaurate automat la valorile anterioare.

Acesta este un detaliu avansat din `man bash`, rar întâlnit în documentația introductivă.

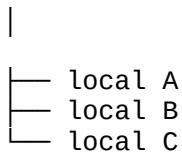
Modelul mental

Script

|

└─ Variabilă globală

└─ Funcție



La ieșirea din funcție

A, B și C dispar

Caseta „În spatele cortinei”

Variabilele locale nu sunt stocate într-o zonă separată de memorie.

Bash păstrează pentru fiecare apel de funcție un context propriu al variabilelor locale și îl elimină automat la revenirea din funcție. Acesta este mecanismul care face posibilă reutilizarea aceluiași nume de variabile în funcții diferite fără conflicte.

Greșeală din lumea reală

Un administrator scrie:

```
backup() {
    FISIER="$1"
}
```

Scriptul funcționează.

După câteva luni apare o a doua funcție care utilizează aceeași variabilă:

```
FISIER
```

Cele două funcții încep să își suprascrie reciproc valorile.

Forma recomandată este:

```
backup() {
    local FISIER="$1"
}
```

Astfel, fiecare funcție își gestionează propriile date.

Administrator Notes

O regulă simplă folosită în multe proiecte Bash este:

Toate variabilele din interiorul unei funcții sunt declarate `local`, cu excepția celor care trebuie în mod intenționat să fie vizibile în exterior.

Această disciplină reduce drastic erorile cauzate de modificarea accidentală a variabilelor globale.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul comenzii `local`;
 - diferența dintre variabile globale și locale;
 - faptul că `local` poate fi utilizată numai în interiorul unei funcții;
 - că acceptă aceleași opțiuni ca `declare`;
 - existența formei speciale `local -`, care localizează modificările opțiunilor shell-ului.
-

Laborator 14.2

Nivel 1 – Înțelegere

Scrie funcția:

```
NUME="Daniel"
```

```
test_local() {  
    local NUME="Maria"  
    echo "$NUME"  
}
```

```
test_local  
echo "$NUME"
```

Explică de ce cele două afișări sunt diferite.

Nivel 2 – Aplicare

Creează două funcții care utilizează aceeași variabilă:

```
local CONTOR
```

Verifică faptul că valorile nu interferează între ele.

Nivel 3 – Administrator

Scrie o funcție care:

- declară toate variabilele cu `local`;
- folosește `local -i` pentru un contor numeric;
- modifică temporar o opțiune a shell-ului folosind `set` și `local -`.

Verifică faptul că, după ieșirea din funcție, opțiunile shell-ului revin la valorile inițiale.

Ce trebuie să reții

- `local` creează variabile vizibile doar în funcția curentă și în funcțiile apelate de aceasta.
- Variabilele locale dispar automat la terminarea funcției.
- `local` acceptă aceleași opțiuni ca `declare`.
- Forma `local -` permite restaurarea automată a opțiunilor shell-ului modificate în interiorul funcției.
- Utilizarea consecventă a variabilelor locale face scripturile mai robuste și mai ușor de întreținut.

Capitolul 14

14.3 Comanda `return`

„Comanda `return` determină încheierea execuției unei funcții shell și întoarcerea unui cod de stare către apelant.”

Ce vei învăța

În acest capitol vei înțelege rolul comenzii `return`, diferența dintre `return` și `exit`, precum și modul în care funcțiile transmit coduri de stare către apelant.

Introducere

În orice limbaj de programare apare aceeași întrebare.

Cum se termină execuția unei funcții?

În Bash, răspunsul este:

return

Această comandă încheie imediat funcția și întoarce controlul către locul din care funcția a fost apelată.

Sintaxa

Forma generală este:

```
return [n]
```

unde:

- `n` reprezintă codul de stare întors apelantului;
 - dacă este omis, Bash utilizează codul ultimei comenzi executate în funcție.
-

Primul exemplu

```
verifica() {  
    echo "Verific..."  
    return 0  
}
```

```
verifica
```

```
echo $?
```

Rezultat:

```
Verific...  
0
```

Funcția s-a încheiat cu succes.

Întoarcerea unui cod de eroare

```
verifica() {  
    return 1  
}
```

```
verifica
```

```
echo $?
```

Rezultat:

```
1
```

Acest cod poate fi utilizat imediat într-o structură:

```
if verifica
then
    echo "Succes"
else
    echo "Eroare"
fi
```

return fără argument

Exemplu:

```
test() {
    ls /etc
    return
}
```

În acest caz, funcția întoarce codul produs de:

```
ls /etc
```

Acesta este comportamentul implicit descris în documentația Bash.

return și exit

Aceasta este cea mai importantă diferență.

Comandă	Efect
return	încheie funcția
exit	încheie întregul shell sau script

Exemplu:

```
functie() {
    return
}
```

```
echo "Înainte"
```

```
functie
```

```
echo "După"
```

Rezultat:

```
Înainte
După
```

În schimb:

```
functie() {
```

```
}    exit
```

determină terminarea întregului script în momentul apelului funcției.

Domeniul de utilizare

`return` poate fi utilizată:

- într-o funcție;
- într-un script executat prin `source (.)`.

Dacă este utilizată în afara acestor contexte, Bash raportează o eroare și întoarce un cod diferit de zero.

Modelul mental

Script

|

▼

functie()

|

├─ comenzi

├─ return 0

▼

Scriptul continuă

Codurile de stare

Ca orice comandă Bash, `return` utilizează coduri de ieșire.

Prin convenție:

Cod	Semnificație

0	succes

≠0	eroare sau condiție specială

Aceste valori sunt interpretate de:

- `if`
- `while`
- `until`
- operatorii `&&` și `||`

exact ca în cazul oricărei alte comenzi.

Caseta „În spatele cortinei”

`return` nu întoarce valori, ci coduri de stare.

Acesta este un aspect esențial.

În Bash, funcțiile nu returnează șiruri de caractere sau numere în sensul limbajelor precum C, Python sau Java.

Ele întorc un **exit status** cu valori cuprinse între 0 și 255. Dacă este nevoie să transmitem date, acestea se realizează prin:

- ieșirea standard (`stdout`);
 - variabile;
 - tablouri;
 - variabile globale sau locale.
-

Greșeală din lumea reală

Un administrator scrie:

```
functie() {  
    return 500  
}
```

și se așteaptă ca funcția să întoarcă valoarea 500.

În realitate, codurile de stare Bash sunt limitate la 8 biți.

Rezultatul observat va fi:

244

deoarece valoarea este redusă modulo 256.

Administrator Notes

Folosește `return` doar pentru a indica:

- succes;
- eșec;
- tipul rezultatului.

Nu utiliza `return` pentru transmiterea datelor.

Dacă funcția trebuie să "întoarcă" un text, afișează-l pe ieșirea standard și capturează-l folosind substituția de comandă:

```
rezultat=$(functie)
```

Aceasta este convenția folosită în majoritatea scripturilor Bash profesionale.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul comenzii `return`;
 - diferența dintre `return` și `exit`;
 - faptul că lipsa argumentului determină întoarcerea codului ultimei comenzi executate;
 - că `return` este validă doar în funcții și în scripturi executate cu `source`.
-

Laborator 14.3

Nivel 1 – Înțelegere

Scrie funcția:

```
verifica() {  
    return 0  
}
```

Execut-o și verifică:

```
echo $?
```

Nivel 2 – Aplicare

Modifică funcția astfel încât să întoarcă:

```
return 1
```

Folosește-o într-o instrucțiune:

```
if verifica
then
    echo "OK"
else
    echo "EROARE"
fi
```

Explică rezultatul.

Nivel 3 – Administrator

Creează două funcții:

```
functie_return()
```

și

```
functie_exit()
```

Prima va utiliza `return`, iar a doua `exit`.

Execută-le într-un script și observă diferența dintre terminarea unei funcții și terminarea întregului proces Bash.

Ce trebuie să reții

- `return` încheie execuția unei funcții și întoarce un cod de stare.
- Dacă nu este specificat un argument, este întors codul ultimei comenzi executate.
- `return` și `exit` au roluri complet diferite.
- Codurile întoarse de `return` sunt coduri de stare (0–255), nu valori de date.

Capitolul 14

14.4 Contextul de execuție al funcțiilor Bash

„Atunci când este executată o funcție, argumentele acesteia devin parametrii poziționali ai funcției. Parametrul special `$0` nu este modificat, iar primul element al variabilei `FUNCNAME` este numele funcției aflate în execuție.”

Ce vei învăța

În acest capitol vei înțelege ce se întâmplă în interiorul Bash atunci când este apelată o funcție și de ce funcțiile se comportă diferit față de scripturile executate ca procese separate.

Introducere

Să presupunem că avem funcția:

```
salut() {  
    echo "Salut, $1!"  
}
```

și o apelăm:

```
salut Daniel
```

Pare evident că:

```
$1 = Daniel
```

Dar apare imediat o întrebare.

Ce se întâmplă cu parametrii scriptului?

Sau cu:

```
$0
```

Ori cu variabilele speciale utilizate de Bash?

Documentația oficială răspunde exact acestor întrebări.

Parametrii poziționali sunt înlocuiți temporar

În momentul apelului:

```
salut Ana Maria
```

funcția primește propriul set de parametri:

```
$1 = Ana  
$2 = Maria  
$# = 2  
$@ = Ana Maria
```

Acești parametri există **numai pe durata execuției funcției**.

La terminarea acesteia, Bash restaurează automat parametrii anteriori.

Variabila \$0

Un detaliu foarte important este că:

\$0

nu se modifică.

Exemplu:

```
echo "$0"
```

poate afișa:

```
backup.sh
```

iar în interiorul funcției:

```
functie() {  
    echo "$0"  
}
```

rezultatul rămâne același:

```
backup.sh
```

Documentația precizează explicit că parametrul special \$0 rămâne neschimbat pe durata execuției unei funcții.

Variabila FUNCNAME

În timpul execuției unei funcții, Bash actualizează automat variabila:

FUNCNAME

Primul element al tabloului conține numele funcției aflate în execuție.

Exemplu:

```
salut() {  
    echo "${FUNCNAME[0]}"  
}
```

```
salut
```

Rezultat:

```
salut
```

Această variabilă este foarte utilă pentru:

- depanare;
 - jurnalizare;
 - construirea mesajelor de eroare.
-

Funcțiile rulează în același shell

Acesta este probabil cel mai important lucru de reținut.

Funcția:

```
functie() {  
    cd /tmp  
}
```

modifică directorul shell-ului apelant.

După apel:

```
functie  
pwd
```

rezultatul este:

```
/tmp
```

Nu a fost creat un proces nou.

Funcția a rulat în contextul shell-ului curent.

Modelul mental

Script

```
$0 = backup.sh
```

```
$1 = arhiva
```

```
$2 = /home
```

|



```
functie test()
```

|



```
$1 = local funcției
```

```
$2 = local funcției  
$0 = backup.sh  
FUNCNAME[0] = test
```

|



Funcția se termină

|



Parametrii originali
sunt restaurați

Caseta „În spatele cortinei”

Bash construiește pentru fiecare apel de funcție un nou context de execuție.

Acesta conține:

- parametrii poziționali ai funcției;
- variabilele locale;
- informațiile despre funcția activă (FUNCNAME).

La revenirea din funcție, acest context este eliminat automat, iar shell-ul continuă cu starea anterioară.

Greșeală din lumea reală

Un administrator presupune că:

```
functie() {  
    cd /var/log  
}
```

va modifica doar directorul funcției.

După apel observă că întregul script rulează acum din:

/var/log

Motivul este simplu.

Funcția nu rulează într-un proces separat, ci în shell-ul curent.

Administrator Notes

Dacă dorești izolarea modificărilor asupra directorului curent sau a variabilelor globale, ai la dispoziție mai multe soluții:

- utilizarea unui **subshell**:

```
(  
    cd /tmp  
    # alte comenzi  
)
```

- utilizarea variabilelor `local`;
- salvarea și restaurarea explicită a stării shell-ului.

Nu presupune niciodată că o funcție izolează automat modificările pe care le face.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- că argumentele funcției devin parametrii poziționali ai acesteia;
 - că `$0` nu este modificat în timpul execuției funcției;
 - rolul variabilei `FUNCNAME`;
 - faptul că funcțiile rulează în shell-ul curent și pot modifica starea acestuia.
-

Laborator 14.4

Nivel 1 – Înțelegere

Creează funcția:

```
test() {  
    echo "FUNCNAME=${FUNCNAME[0]}"  
    echo "\$0=$0"  
    echo "\$1=$1"  
}
```

test exemplu

Observă valorile afișate.

Nivel 2 – Aplicare

Scrie un script care afișează valorile lui:

```
$0  
$#  
$@
```

Înainte și după apelul unei funcții care primește alți parametri.

Verifică faptul că parametrii originali sunt restaurați după terminarea funcției.

Nivel 3 – Administrator

Construiește două funcții:

- una care schimbă directorul folosind `cd`;
- una care execută aceeași operație într-un subshell:

```
(  
    cd /tmp  
    pwd  
)
```

Compară rezultatul și explică diferența dintre cele două abordări.

Ce trebuie să reții

- La apelul unei funcții, Bash creează un nou set de parametri poziționali.
- Parametrul special `$0` rămâne neschimbat.
- `FUNCNAME[0]` conține numele funcției aflate în execuție.
- Funcțiile rulează în shell-ul curent și pot modifica directorul, variabilele și alte elemente ale mediului shell.
- La terminarea funcției, Bash restaurează automat parametrii poziționali ai apelantului.

Capitolul 14

14.5 Stiva apelurilor: `FUNCNAME`, `BASH_SOURCE` și `BASH_LINENO`

„Variabilele `FUNCNAME`, `BASH_SOURCE` și `BASH_LINENO` descriu stiva apelurilor funcțiilor shell și permit identificarea funcției curente, a fișierului în care este definită și a liniei din care a fost apelată.”

Ce vei învăța

În acest capitol vei înțelege cum urmărește Bash apelurile dintre funcții și cum poți identifica rapid locul din care provine o eroare.

Introducere

În capitolul anterior am aflat că:

`FUNCNAME`

conține numele funcției aflate în execuție.

Dar aceasta este doar o parte din informație.

Bash păstrează, de fapt, întreaga **stivă a apelurilor** (*call stack*).

Aceasta este alcătuită din trei tablouri care funcționează împreună:

- `FUNCNAME`
 - `BASH_SOURCE`
 - `BASH_LINENO`
-

Variabila `FUNCNAME`

`FUNCNAME` este un tablou.

Elementul:

`${FUNCNAME[0]}`

conține funcția executată în acel moment.

Exemplu:

```
afisare() {  
    echo "${FUNCNAME[0]}"  
}
```

`afisare`

Rezultat:

`afisare`

Dacă funcțiile sunt apelate una din alta, tabloul conține întreaga succesiune a apelurilor.

Variabila **BASH_SOURCE**

BASH_SOURCE este tot un tablou.

El indică fișierul în care este definită fiecare funcție din **FUNCNAME**.

Astfel:

`${BASH_SOURCE[0]}`

reprezintă fișierul în care este definită funcția curentă.

Variabila **BASH_LINENO**

BASH_LINENO completează informația.

Ea conține numărul liniei din care a fost apelată funcția corespunzătoare.

Împreună:

FUNCNAME

↓

BASH_SOURCE

↓

BASH_LINENO

descriu traseul complet al execuției.

Relația dintre cele trei tablouri

Documentația Bash precizează explicit că elementele au aceeași poziție logică.

Pentru indicele **i**:

FUNCNAME[i]

este funcția,

BASH_SOURCE[i]

este fișierul în care este definită,

iar

BASH_LINENO[i]

reprezintă linia din care a fost apelată.

Modelul mental

main

|

▼

backup()

|

▼

copiere()

|

▼

arhivare()

Tablourile pot conține, de exemplu:

FUNCNAME

```
[0] arhivare
[1] copiere
[2] backup
[3] main
```

și informațiile corespunzătoare în BASH_SOURCE și BASH_LINENO.

Relația cu caller

În capitolul dedicat comenzii:

caller

am văzut că aceasta afișează informații despre apelul curent.

Conform documentației, caller utilizează exact informațiile disponibile în:

- FUNCNAME;
- BASH_SOURCE;
- BASH_LINENO.

Cu alte cuvinte, `caller` este o interfață simplificată peste aceste variabile.

Caseta „În spatele cortinei”

Aceste variabile există numai atunci când Bash execută funcții shell.

La fiecare apel de funcție, Bash actualizează stiva internă.

La revenirea din funcție, informațiile sunt eliminate automat.

Acesta este mecanismul care permite reconstruirea traseului complet al execuției.

Greșeală din lumea reală

Un administrator utilizează:

```
echo "$LINENO"
```

și se așteaptă să afle din ce funcție provine apelul.

În realitate:

- `LINENO` indică linia curentă;
- `FUNCNAME`, `BASH_SOURCE` și `BASH_LINENO` descriu relațiile dintre funcțiile apelate.

Sunt mecanisme diferite.

Administrator Notes

În scripturile mici nu vei utiliza aproape niciodată aceste variabile.

În schimb, în biblioteci Bash sau în aplicații de câteva sute de linii, ele sunt extrem de utile pentru:

- jurnalizarea erorilor;
 - construirea mesajelor de diagnostic;
 - depanare;
 - implementarea unui mecanism propriu de logging.
-

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul variabilei `FUNCNAME`;
 - existența variabilelor `BASH_SOURCE` și `BASH_LINENO`;
 - faptul că împreună descriu stiva apelurilor funcțiilor Bash;
 - relația lor cu built-in-ul `caller`.
-

Laborator 14.5

Nivel 1 – Înțelegere

Creează două funcții:

```
functie_A() {  
    functie_B  
}  
  
functie_B() {  
    declare -p FUNCNAME  
}
```

Observă ordinea funcțiilor în tabloul `FUNCNAME`.

Nivel 2 – Aplicare

Extinde exemplul astfel încât funcția `functie_B` să afișeze:

```
printf '%s\n' "${BASH_SOURCE[@]}"  
printf '%s\n' "${BASH_LINENO[@]}"
```

Compară rezultatele cu valorile din `FUNCNAME`.

Nivel 3 – Administrator

Construiește un mic mecanism de jurnalizare care afișează, la apariția unei erori:

- numele funcției (`FUNCNAME[0]`);
- fișierul (`BASH_SOURCE[0]`);
- linia apelului (`BASH_LINENO[0]`).

Explică de ce aceste informații sunt mai utile decât afișarea unui simplu mesaj de eroare.

Ce trebuie să reții

- FUNCNAME este un tablou care conține funcțiile din stiva apelurilor.
- BASH_SOURCE indică fișierul în care este definită fiecare funcție.
- BASH_LINENO indică linia din care funcția a fost apelată.
- caller utilizează aceste informații pentru a afișa contextul apelurilor.

Capitolul 14

14.6 Tratarea semnalelor cu trap

„Comanda trap permite specificarea comenzilor care vor fi executate atunci când shell-ul primește anumite semnale sau când apar evenimente speciale precum ieșirea din shell ori execuția unei comenzi.”

Ce vei învăța

În acest capitol vei înțelege ce sunt semnalele Unix, cum funcționează comanda trap și cum poate fi utilizată pentru curățarea resurselor, jurnalizare și depanare.

Introducere

Un script Bash nu rulează izolat.

În timpul execuției poate apărea oricare dintre următoarele situații:

- utilizatorul apasă **Ctrl+C**;
- terminalul este închis;
- un proces părinte trimite un semnal;
- scriptul se termină normal;
- apare o eroare.

Cum poate reacționa scriptul?

Răspunsul este:

trap

Ce este un semnal?

Un **semnal** este un mesaj scurt trimis unui proces de către kernel sau de către un alt proces.

Exemple cunoscute:

Semnal	Situație obișnuită
SIGINT	Ctrl+C
SIGTERM	solicitare de terminare
SIGHUP	închiderea terminalului
SIGQUIT	Ctrl+\

Documentația Bash descrie modul în care shell-ul tratează aceste semnale atât în modul interactiv, cât și în timpul execuției scripturilor.

Sintaxa

Forma generală este:

```
trap 'comenzi' semnal
```

Exemplu:

```
trap 'echo "Script întrerupt"' SIGINT
```

Dacă utilizatorul apasă:

Ctrl+C

Bash execută mai întâi:

```
echo "Script întrerupt"
```

și abia apoi continuă comportamentul asociat semnalului, dacă acesta nu a fost modificat.

Evenimentul EXIT

Una dintre cele mai utile utilizări este:

```
trap 'rm -f /tmp/fisier.tmp' EXIT
```

Indiferent dacă scriptul se termină cu succes sau printr-o instrucțiune `exit`, Bash execută comanda asociată evenimentului EXIT.

Acesta este echivalentul unui bloc **cleanup** din alte limbaje de programare.

Restaurarea comportamentului implicit

Pentru eliminarea unui `trap`:

```
trap - SIGINT
```

Bash revine la comportamentul implicit pentru semnalul respectiv.

Afișarea trap-urilor active

Execută:

```
trap -p
```

Vor fi afișate toate acțiunile definite în sesiunea curentă.

`trap` și funcțiile

Foarte des, `trap` este utilizată împreună cu funcțiile.

Exemplu:

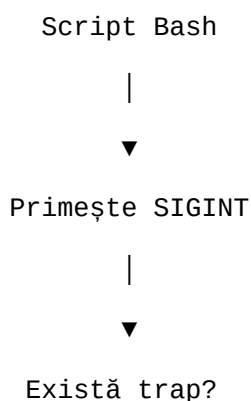
```
cleanup() {  
    rm -f "$TEMPFILE"  
}
```

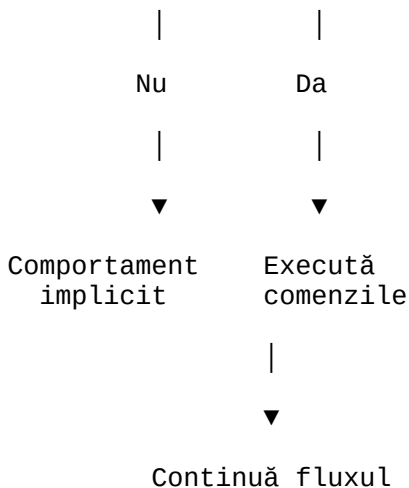
```
trap cleanup EXIT
```

La terminarea scriptului, Bash va apela funcția `cleanup`.

Această abordare este mai lizibilă decât plasarea directă a comenzilor în instrucțiunea `trap`.

Modelul mental





Relația cu **DEBUG**, **RETURN** și **ERR**

Pe lângă semnalele Unix, Bash permite definirea unor trap-uri pentru evenimente interne, precum:

- **DEBUG**
- **RETURN**
- **ERR**
- **EXIT**

Aceste mecanisme sunt folosite frecvent pentru depanare și jurnalizare. Documentația explică și modul în care acestea sunt moștenite de funcții și de fișiere încărcate cu **SOURCE**, în funcție de opțiunile shell-ului (de exemplu **extdebug**).

Vom studia fiecare dintre aceste evenimente într-un capitol dedicat.

Caseta „În spatele cortinei”

Dacă Bash așteaptă terminarea unei comenzi și primește un semnal pentru care există un **trap**, acesta nu este executat imediat. Documentația precizează că execuția trap-ului este amânată până la terminarea comenzii curente. Există însă un comportament special atunci când shell-ul așteaptă prin built-in-ul **wait**.

Acest detaliu explică de ce uneori reacția la un semnal pare întârziată.

Greșeală din lumea reală

Un administrator creează un fișier temporar:

```
TEMP=$(mktemp)
```

și îl șterge doar la sfârșitul scriptului.

Dacă utilizatorul apasă **Ctrl+C**, scriptul se oprește înainte de ștergerea fișierului, iar acesta rămâne pe disc.

Soluția este:

```
trap 'rm -f "$TEMP"' EXIT
```

Astfel, fișierul este eliminat indiferent de modul în care se încheie scriptul.

Administrator Notes

O regulă foarte bună în scripturile de producție este:

- creezi o resursă temporară;
- definești imediat un `trap EXIT` pentru eliberarea acesteia.

Nu aștepta sfârșitul scriptului pentru operațiile de curățare. În acest fel, scriptul rămâne robust chiar dacă este întrerupt înainte de final.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul comenzii `trap`;
 - utilizarea cu semnale precum `SIGINT` și `SIGTERM`;
 - utilizarea evenimentului `EXIT`;
 - `trap -p` pentru afișarea trap-urilor active;
 - `trap -` pentru restaurarea comportamentului implicit.
-

Laborator 14.6

Nivel 1 – Înțelegere

Scrie un script:

```
trap 'echo "Ctrl+C detectat"' SIGINT
```

```
while true
do
    sleep 1
```

done

Rulează-l și întrerupe-l folosind **Ctrl+C**.

Nivel 2 – Aplicare

Creează un fișier temporar:

```
TEMP=$(mktemp)
```

și definește:

```
trap 'rm -f "$TEMP"' EXIT
```

Verifică faptul că fișierul este eliminat indiferent de modul în care se termină scriptul.

Nivel 3 – Administrator

Construiește un script care:

- creează un director temporar;
- scrie date în el;
- înregistrează într-un jurnal momentul în care primește **SIGTERM**;
- elimină directorul folosind un `trap EXIT`.

Testează comportamentul trimițând semnalul:

```
kill -TERM <PID>
```

și verifică faptul că resursele sunt eliberate corect.

Ce trebuie să reții

- `trap` permite asocierea unor acțiuni cu semnale Unix și cu evenimente interne Bash.
- `EXIT` este cel mai utilizat eveniment pentru operații de curățare.
- `trap -p` afișează trap-urile active, iar `trap` - restaurează comportamentul implicit.
- Trap-urile sunt un instrument esențial pentru scrierea unor scripturi robuste și ușor de depanat.

Capitolul 14

14.7 Evenimentul ERR și tratarea erorilor

„Dacă un trap este asociat pseudo-semnalului ERR, acesta este executat ori de câte ori o comandă simplă se termină cu un cod de ieșire diferit de zero, cu anumite excepții definite de Bash.”

Ce vei învăța

În acest capitol vei înțelege cum funcționează evenimentul ERR, când este declanșat și cum poate fi utilizat pentru construirea unui mecanism centralizat de tratare a erorilor.

Introducere

În majoritatea scripturilor întâlnim verificări repetate:

```
if ! cp fisier1 fisier2
then
    echo "Copiere eșuată"
    exit 1
fi
```

sau:

```
cp fisier1 fisier2

if [ $? -ne 0 ]
then
    echo "Eroare"
fi
```

Acest stil funcționează, dar conduce rapid la scripturi greu de citit.

Bash oferă o alternativă:

```
trap '...' ERR
```

Astfel, toate erorile pot fi tratate într-un singur loc.

Sintaxa

Forma generală este:

```
trap 'comenzi' ERR
```

Exemplu:

```
trap 'echo "A apărut o eroare."' ERR
```

Dacă o comandă se termină cu un cod diferit de zero, Bash execută comenzile asociate evenimentului ERR.

Un exemplu simplu

```
trap 'echo "Comandă eșuată."' ERR
```

```
ls /director_inexistent
```

```
echo "Scriptul continuă"
```

Rezultat posibil:

```
ls: cannot access '/director_inexistent': No such file or directory
Comandă eșuată.
Scriptul continuă
```

Observă că `trap ERR` **nu oprește** automat execuția scriptului.

El doar execută acțiunea asociată.

Împreună cu `set -e`

Foarte frecvent întâlnim:

```
set -e
```

```
trap 'echo "Execuția s-a oprit din cauza unei erori."' ERR
```

În această combinație:

- `ERR` afișează informațiile de diagnostic;
- `set -e` întrerupe execuția scriptului.

Cele două mecanisme se completează reciproc.

Când **NU** este executat `ERR`

Acesta este unul dintre cele mai importante detalii din documentația Bash.

Trap-ul `ERR` **nu este declanșat** în toate situațiile.

Conform `man bash`, el nu este executat atunci când eroarea apare:

- într-o condiție `if`;
- într-o condiție `while` sau `until`;
- într-o comandă precedată de operatorul `!`;
- în comenzile intermediare dintr-o conductă (`|`), cu excepția ultimei, dacă nu este activată opțiunea `pipefail`;
- în alte contexte în care Bash testează intenționat codul de ieșire al unei comenzi.

Aceste excepții există deoarece, în aceste construcții, un cod diferit de zero poate face parte din logica normală a scriptului.

Un exemplu

```
trap 'echo "ERR"' ERR

if grep "utilizator" fisier.txt
then
    echo "Găsit"
fi
```

Dacă `grep` nu găsește șirul căutat și întoarce `1`, `trap`-ul `ERR` nu este executat.

Acesta este comportamentul normal al Bash și este descris explicit în documentația oficială.

Funcție de tratare a erorilor

În practică este recomandată utilizarea unei funcții:

```
eroare() {
    echo "Eroare în funcția ${FUNCNAME[1]}"
    echo "Linia: ${BASH_LINENO[0]}"
}

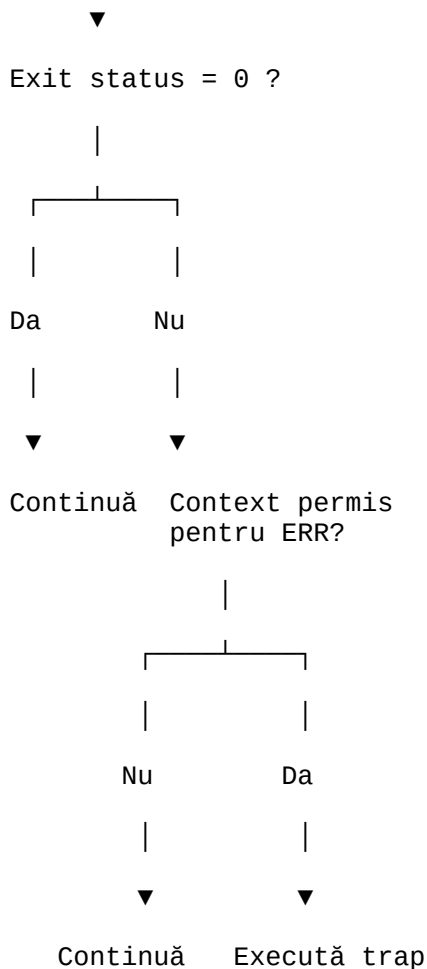
trap eroare ERR
```

Această abordare este mai ușor de extins și reutilizat decât includerea directă a comenzilor în instrucțiunea `trap`.

Modelul mental

Comandă

|



Caseta „În spatele cortinei”

Evenimentul ERR nu este un semnal Unix.

El este un **pseudo-semnal Bash**, generat intern de shell atunci când sunt îndeplinite condițiile descrise în documentație. Din acest motiv, comportamentul său depinde de regulile limbajului Bash și nu de nucleul sistemului de operare.

Greșeală din lumea reală

Un administrator presupune că:

```
trap 'echo "ERR"' ERR
```

va fi executat pentru **orice** comandă care întoarce un cod diferit de zero.

După testare observă că nu se execută în interiorul unei instrucțiuni:

```
if
```


Aceasta nu este o eroare a Bash.

Este comportamentul normal descris în documentația oficială.

Administrator Notes

În scripturile mari, este recomandat să existe **un singur punct de tratare a erorilor**.

O funcție asociată evenimentului ERR, împreună cu variabile precum:

- FUNCNAME
- BASH_SOURCE
- BASH_LINENO

permite construirea unui mecanism de diagnostic comparabil cu cel întâlnit în limbaje de programare mult mai complexe.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul pseudo-semnalului ERR;
 - faptul că este gestionat prin `trap`;
 - principalele situații în care ERR **nu** este declanșat;
 - relația dintre ERR și `set -e`.
-

Laborator 14.7

Nivel 1 – Înțelegere

Definește:

```
trap 'echo "Eroare detectată"' ERR  
false
```

Observă executarea trap-ului.

Nivel 2 – Aplicare

Testează aceeași comandă într-o instrucțiune:

```
if false
then
    echo "OK"
fi
```

Compară comportamentul și explică de ce ERR nu este declanșat.

Nivel 3 – Administrator

Construiește o funcție:

```
log_eroare() {
    echo "Funcție: ${FUNCNAME[1]}"
    echo "Fișier: ${BASH_SOURCE[1]}"
    echo "Linie: ${BASH_LINENO[0]}"
}

trap log_eroare ERR
```

Execută intenționat o comandă care eșuează și analizează informațiile afișate.

Ce trebuie să reții

- ERR este un pseudo-semnal Bash utilizat pentru tratarea centralizată a erorilor.
- El nu este declanșat în toate contextele; `if`, `while`, `until`, `!` și anumite conduite reprezintă excepții definite de Bash.
- ERR nu oprește automat execuția scriptului; pentru aceasta este utilizat frecvent împreună cu `set -e`.
- Combinarea ERR cu `FUNCNAME`, `BASH_SOURCE` și `BASH_LINENO` oferă un mecanism puternic de diagnostic și depanare.

Capitolul 14

14.8 Evenimentul DEBUG și depanarea execuției

*„Dacă un trap este asociat pseudo-semnalului **DEBUG**, comenzile asociate sunt executate înainte de fiecare comandă simplă, înaintea anumitor comenzi compuse și înainte de prima comandă executată într-o funcție shell.”*

Ce vei învăța

În acest capitol vei înțelege rolul evenimentului `DEBUG`, modul în care acesta permite urmărirea execuției unui script și diferența dintre `DEBUG` și `set -x`.

Introducere

În capitolul anterior am utilizat:

```
trap '...' ERR
```

Acesta reacționează **după** ce o comandă s-a terminat cu eroare.

Dar uneori dorim altceva.

Vrem să vedem fiecare comandă înainte ca ea să fie executată.

Pentru această situație Bash oferă:

```
trap '...' DEBUG
```

Sintaxa

Forma generală este:

```
trap 'comenzi' DEBUG
```

Exemplu:

```
trap 'echo "Urmează: $BASH_COMMAND"' DEBUG
```

Înainte de executarea fiecărei comenzi, Bash va afișa conținutul variabilei:

```
BASH_COMMAND
```

care conține comanda ce urmează să fie executată.

Un exemplu

Script:

```
trap 'echo "Execut: $BASH_COMMAND"' DEBUG
```

```
echo "Salut"
```

```
pwd
```

date

Rezultat posibil:

Execut: echo "Salut"
Salut

Execut: pwd
/home/student

Execut: date
Mon Aug 4 ...

Observă că mesajul este afișat **înainte** de fiecare comandă.

Variabila BASH_COMMAND

În timpul unui DEBUG trap, Bash pune la dispoziție variabila:

BASH_COMMAND

Ea conține comanda care urmează să fie executată.

Această variabilă este extrem de utilă pentru:

- jurnalizare;
 - depanare;
 - instrumente de analiză a execuției.
-

DEBUG și set -x

La prima vedere, cele două mecanisme par asemănătoare.

Există însă o diferență importantă.

Mecanism	Rol
set -x	Bash afișează comenzile într-un format standard
trap DEBUG	Programatorul decide ce acțiune execută Bash înaintea fiecărei comenzi

Cu DEBUG putem:

- scrie într-un jurnal;
 - cronometra execuția comenzilor;
 - afișa informații suplimentare;
 - implementa propriul sistem de trasare.
-

Relația cu FUNCNAME

Într-un `DEBUG trap` putem utiliza și variabilele studiate anterior:

`FUNCNAME`
`BASH_SOURCE`
`BASH_LINENO`

Astfel putem afla:

- funcția în care ne aflăm;
- fișierul sursă;
- locul apelului.

Aceste informații permit construirea unui mecanism simplu de trasare a execuției.

Opțiunea `extdebug`

Documentația Bash descrie un comportament special atunci când este activată opțiunea:

`shopt -s extdebug`

Printre efectele acesteia se numără:

- dacă o comandă executată de `DEBUG trap` întoarce un cod diferit de zero, următoarea comandă este omisă;
- dacă întoarce valoarea 2 în timpul execuției unei funcții sau a unui script încărcat cu `source`, Bash simulează o instrucțiune `return`;
- funcțiile, substituțiile de comandă și subshell-urile moștenesc trap-urile `DEBUG` și `RETURN`;
- este activată și moștenirea trap-ului `ERR`.

Aceste facilități sunt destinate în principal debuggerelor și instrumentelor de analiză.

Modelul mental

Comandă

|

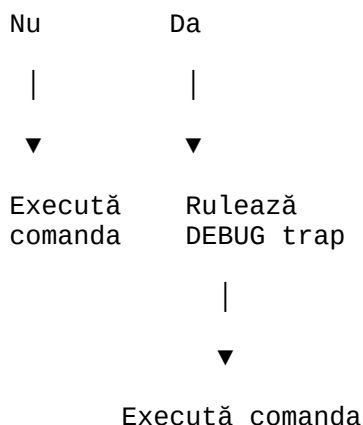
▼

Există `DEBUG trap`?

|

|

|



Caseta „În spatele cortinei”

DEBUG este un **pseudo-semnal Bash**, nu un semnal Unix.

El este generat de interpretul Bash înaintea execuției comenzilor. Din acest motiv poate observa structura internă a scriptului, lucru imposibil pentru semnalele furnizate de sistemul de operare.

Greșeală din lumea reală

Un administrator definește:

```
trap 'echo "$BASH_COMMAND"' DEBUG
```

și observă că sunt afișate foarte multe mesaje.

Acest comportament este normal.

DEBUG este executat înaintea fiecărei comenzi simple și poate produce un volum mare de informații în scripturile complexe.

Administrator Notes

În practică, `DEBUG trap` este folosit mai ales:

- temporar, în timpul dezvoltării;
- pentru construirea unor sisteme proprii de logging;
- în instrumente automate de depanare.

Nu este recomandat să rămână activ permanent în scripturile de producție, deoarece poate introduce un cost suplimentar și poate genera jurnale foarte voluminoase.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul pseudo-semnalului DEBUG;
 - faptul că este executat înaintea comenzilor;
 - diferența dintre DEBUG și ERR;
 - relația dintre DEBUG și opțiunea `extdebug`.
-

Laborator 14.8

Nivel 1 – Înțelegere

Definește:

```
trap 'echo "Execut: $BASH_COMMAND"' DEBUG

echo "Salut"
pwd
date
```

Observă ordinea mesajelor afișate.

Nivel 2 – Aplicare

Modifică trap-ul astfel încât să afișeze și:

```
echo "${FUNCNAME[0]}"
```

Execută un script care conține două funcții și urmărește modificarea valorii `FUNCNAME`.

Nivel 3 – Administrator

Construiește o funcție de jurnalizare:

```
log_debug() {
    printf "[%s] %s\n" "${FUNCNAME[1]}" "$BASH_COMMAND"
}

trap log_debug DEBUG
```

Execută un script cu mai multe funcții și analizează jurnalul produs. Încearcă apoi să activezi:

shopt -s extdebug

și observă modul în care se modifică comportamentul descris în documentația Bash.

Ce trebuie să reții

- DEBUG este un pseudo-semnal Bash executat înaintea comenzilor.
- Variabila BASH_COMMAND conține comanda care urmează să fie executată.
- DEBUG oferă un mecanism flexibil de trasare și depanare, diferit de set -x.
- Opțiunea extdebug extinde comportamentul DEBUG, inclusiv moștenirea trap-urilor și controlul fluxului de execuție.

Capitolul 15

15.1 Variabilele BASH_VERSION și BASH_VERSINFO

„Bash pune la dispoziția scripturilor informații despre propria versiune prin intermediul variabilelor speciale BASH_VERSION și BASH_VERSINFO.”

Ce vei învăța

În acest capitol vei înțelege cum poate un script să afle versiunea Bash în care rulează și de ce această informație este importantă pentru compatibilitate.

Introducere

Nu toate sistemele Linux utilizează aceeași versiune de Bash.

De exemplu:

- Debian Stable poate utiliza o versiune;
- Ubuntu o alta;
- Red Hat Enterprise Linux alta;
- macOS poate include o versiune considerabil mai veche.

Acest lucru contează deoarece unele funcționalități există doar în versiunile mai noi.

Pentru a afla versiunea interpreterului, Bash oferă două variabile speciale.

Variabila BASH_VERSION

Cea mai simplă este:

```
echo "$BASH_VERSION"
```

Rezultat posibil:

```
5.2.37(1)-release
```

Aceasta conține versiunea completă a interpreterului Bash, inclusiv informațiile despre nivelul de release.

La ce este utilă?

Un script poate afișa informații de diagnostic:

```
echo "Rulez pe Bash $BASH_VERSION"
```

sau poate înregistra versiunea în jurnalul aplicației.

Este și una dintre primele variabile pe care un administrator o verifică atunci când investighează un comportament diferit între două sisteme.

Variabila BASH_VERSINFO

Pentru comparații automate, Bash oferă și:

```
BASH_VERSINFO
```

Aceasta este **un tablou** (array), nu un șir de caractere.

Structura tabloului

Elementele principale sunt:

Element	Conținut
BASH_VERSINFO[0]	versiunea majoră
BASH_VERSINFO[1]	versiunea minoră
BASH_VERSINFO[2]	nivelul patch
BASH_VERSINFO[3]	build

Element	Conținut
BASH_VERSION[4]	starea release
BASH_VERSION[5]	arhitectura/platforma pentru care a fost compilat Bash

Exemplu

```
printf '%s\n' "${BASH_VERSION[@]}"
```

Rezultat posibil:

```
5
2
37
1
release
x86_64-pc-linux-gnu
```

Valorile exacte depind de versiunea Bash instalată.

De ce există două variabile?

Rolurile lor sunt diferite.

BASH_VERSION

↓

Afișare pentru oameni

BASH_VERSION

↓

Prelucrare în scripturi

Prima este convenabilă pentru afișare.

A doua este convenabilă pentru comparații și decizii automate.

Exemplu practic

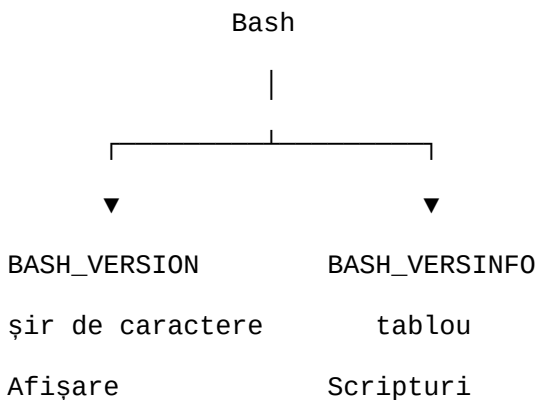
Putem verifica versiunea majoră:

```
if (( BASH_VERSION[0] >= 5 ))
then
    echo "Versiune acceptată"
else
```

```
    echo "Versiune prea veche"
fi
```

Observă că, fiind un tablou numeric, `BASH_VERSINFO` este mult mai potrivită pentru astfel de verificări decât analiza șirului din `BASH_VERSION`.

Modelul mental



Caseta „În spatele cortinei”

Aceste variabile sunt create automat de Bash la pornirea shell-ului.

Ele descriu interpretul aflat în execuție și nu pot fi utilizate pentru a afla versiunea altui shell instalat în sistem. Dacă un script este executat cu o altă implementare de shell (de exemplu `dash`), aceste variabile nu sunt disponibile.

Greșeală din lumea reală

Un administrator compară versiunile astfel:

```
if [ "$BASH_VERSION" \> "5.10" ]
```

Comparația este lexicografică, nu numerică, și poate produce rezultate neașteptate.

Pentru verificări de versiune este preferabil să folosești elementele numerice din `BASH_VERSINFO`.

Administrator Notes

Dacă un script utilizează facilități moderne (de exemplu anumite opțiuni `glob`, `coproc`, `mapfile` sau alte caracteristici introduse în versiunile recente), este recomandat să verifice versiunea Bash la începutul execuției și să afișeze un mesaj clar dacă interpreterul este prea vechi.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul variabilei `BASH_VERSION`;
 - faptul că `BASH_VERSINFO` este un tablou;
 - diferența dintre utilizarea pentru afișare și utilizarea pentru comparații automate.
-

Laborator 15.1

Nivel 1 – Înțelegere

Afișează:

```
echo "$BASH_VERSION"
```

și notează versiunea interpreterului instalat pe sistem.

Nivel 2 – Aplicare

Afișează toate elementele tabloului:

```
printf '%s\n' "${BASH_VERSINFO[@]}"
```

Identifică versiunea majoră, minoră și nivelul de patch.

Nivel 3 – Administrator

Scrie un script care verifică dacă versiunea majoră Bash este cel puțin 5:

```
if (( BASH_VERSINFO[0] >= 5 ))
then
    echo "Versiune compatibilă."
else
    echo "Este necesară o versiune mai nouă de Bash."
fi
```

Testează scriptul și explică de ce este preferată utilizarea `BASH_VERSION` în locul analizei textului din `BASH_VERSION`.

Ce trebuie să reții

- `BASH_VERSION` conține versiunea completă a interpreterului Bash.
- `BASH_VERSION` este un tablou care separă componentele versiunii.
- `BASH_VERSION` este potrivită pentru afișare, iar `BASH_VERSION` pentru verificări automate și comparații numerice.
- Aceste variabile descriu interpreterul Bash aflat în execuție și sunt create automat la pornirea shell-ului.

Capitolul 15

15.2 Identitatea procesului Bash – \$\$, BASHPID și PPID

„\$\$ se extinde la PID-ul shell-ului. Într-un subshell creat cu (), el continuă să indice PID-ul shell-ului curent, nu pe cel al subshell-ului. Variabila BASHPID conține PID-ul procesului Bash aflat efectiv în execuție.”

Ce vei învăța

În acest capitol vei înțelege diferența dintre \$\$ și BASHPID, rolul variabilei PPID și modul în care Bash gestionează procesele și subshell-urile.

Introducere

Foarte mulți administratori folosesc:

```
echo $$
```

și spun:

„Acesta este PID-ul shell-ului.”

În majoritatea situațiilor au dreptate.

Dar nu întotdeauna.

În anumite contexte, Bash face o diferență între:

`$$`

și

`BASHPID`

Această diferență apare în special atunci când folosim **subshell-uri**.

Parametrul special `$$`

Execută:

```
echo $$
```

Rezultat posibil:

24518

Acesta reprezintă PID-ul shell-ului.

Într-un script, valoarea este adesea utilizată pentru:

- generarea numelor unice de fișiere;
 - identificarea procesului;
 - jurnalizare.
-

Variabila `BASHPID`

Execută:

```
echo "$BASHPID"
```

Într-o sesiune obișnuită vei observa probabil aceeași valoare:

24518

La prima vedere pare identică cu `$$`.

Totuși, documentația Bash precizează că `BASHPID` reprezintă **PID-ul procesului Bash curent**, iar acest lucru diferă de `$$` în anumite situații, cum sunt subshell-urile care nu necesită reinițializarea Bash.

Primul experiment

Execută:

```
echo "Shell: $$ / $BASHPID"

(
  echo "Subshell: $$ / $BASHPID"
)
```

Pe multe sisteme vei observa un rezultat de forma:

```
Shell:      24518 / 24518
Subshell: 24518 / 24536
```

Observație:

- \$\$ rămâne neschimbat;
- BASHPID indică noul proces Bash creat pentru subshell.

Acesta este unul dintre cele mai frumoase experimente din Bash.

Variabila PPID

Fiecare proces are un părinte.

Bash pune la dispoziție:

```
echo "$PPID"
```

Rezultatul este PID-ul procesului care a lansat shell-ul curent.

De exemplu:

```
Terminal
  |
  ▼
bash
```

Terminalul este procesul părinte al shell-ului.

Relația dintre cele trei valori

Modelul poate fi imaginat astfel:

Proces părinte

PID = 1800

|



Bash

PPID = 1800

PID = 24518

\$\$ = 24518

BASHPID = 24518

La apariția unui subshell:

bash

PID 24518



Subshell

PID 24536

\$\$ = 24518

BASHPID = 24536

Acesta este exact motivul pentru care există BASHPID.

La ce sunt utile?

Aceste variabile sunt utilizate frecvent pentru:

- generarea fișierelor temporare;
 - jurnalizare;
 - depanare;
 - identificarea proceselor;
 - comunicarea între procese.
-

Caseta „În spatele cortinei”

Documentația Bash precizează că atribuirea unei valori variabilei BASHPID nu are niciun efect asupra procesului. Mai mult, dacă variabila este eliminată cu `unset`, ea își pierde proprietățile speciale, chiar dacă este creată din nou ulterior.

Acesta este comportamentul obișnuit al mai multor variabile speciale Bash.

Greșeală din lumea reală

Un administrator presupune că:

`$$`

reprezintă întotdeauna PID-ul procesului aflat în execuție.

După introducerea unui subshell observă că valoarea nu se schimbă.

Motivul este descris explicit în `man bash`:

`$$` continuă să indice shell-ul inițial, în timp ce `BASHPID` indică procesul Bash care rulează efectiv codul în acel moment.

Administrator Notes

Pentru scripturile obișnuite:

`$$`

este suficient.

Dacă însă lucrezi cu:

- subshell-uri;
- procese concurente;
- depanare avansată;

preferă:

`BASHPID`

deoarece descrie procesul Bash aflat efectiv în execuție.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul parametrului special `$$`;
 - rolul variabilei `BASHPID`;
 - diferența dintre cele două în subshell-uri;
 - rolul variabilei `PPID`.
-

Laborator 15.2

Nivel 1 – Înțelegere

Execută:

```
echo "PID shell: $$"  
echo "BASHPID: $BASHPID"  
echo "PPID: $PPID"
```

Compară valorile și identifică procesul părinte.

Nivel 2 – Aplicare

Execută:

```
echo "Principal: $$ / $BASHPID"  
  
(  
    echo "Subshell: $$ / $BASHPID"  
)
```

Explică de ce doar BASHPID se modifică.

Nivel 3 – Administrator

Folosește:

```
ps -fp $$  
  
ps -fp "$PPID"  
  
ps -fp "$BASHPID"
```

Compară procesele afișate și desenează arborele proceselor pentru shell-ul tău.

Ce trebuie să reții

- \$\$ este parametrul special care identifică shell-ul.
- BASHPID identifică procesul Bash aflat efectiv în execuție și poate diferi de \$\$ în subshell-uri.
- PPID conține identificatorul procesului părinte.
- Diferența dintre \$\$ și BASHPID devine vizibilă mai ales în scenariile care utilizează subshell-uri.

Capitolul 15

15.3 Variabila BASH_SUBSHELL

„Variabila BASH_SUBSHELL este incrementată cu unu în fiecare subshell sau mediu de subshell creat de Bash.”

Ce vei învăța

În acest capitol vei înțelege ce este un subshell, cum urmărește Bash nivelul de execuție și în ce situații variabila BASH_SUBSHELL este utilă pentru depanare și scripting.

Introducere

În capitolul anterior am observat că:

BASHPID

se modifică atunci când Bash creează un subshell.

Dar apare imediat o întrebare.

Cum știe Bash dacă rulează în shell-ul principal sau într-un subshell?

Răspunsul este:

BASH_SUBSHELL

Rolul variabilei

Execută:

```
echo "$BASH_SUBSHELL"
```

În shell-ul principal rezultatul este:

0

Acest lucru înseamnă că execuția are loc în shell-ul inițial.

Primul subshell

Execută:

```
echo "Nivel: $BASH_SUBSHELL"

(
    echo "Nivel: $BASH_SUBSHELL"
)
```

Rezultat posibil:

```
Nivel: 0
Nivel: 1
```

La intrarea într-un subshell, Bash incrementează automat valoarea variabilei.

Subshell-uri imbricate

Exemplu:

```
echo "$BASH_SUBSHELL"

(
    echo "$BASH_SUBSHELL"

    (
        echo "$BASH_SUBSHELL"
    )
)
```

Rezultat posibil:

```
0
1
2
```

Fiecare nivel nou de subshell mărește valoarea cu unu.

Ce creează un subshell?

Printre cele mai întâlnite situații sunt:

```
(
    comenzi
)
```

subshell explicit,

substituția de comandă:

`$(comandă)`

și alte contexte interne în care Bash execută comenzile într-un mediu separat. Numărul exact al subshell-urilor depinde de construcția utilizată și de implementarea Bash.

Relația cu \$\$ și BASHPID

Acum putem compara toate cele trei variabile.

Shell principal:

```
$$           = 25010
BASHPID      = 25010
BASH_SHELL   = 0
```

În subshell:

```
$$           = 25010
BASHPID      = 25034
BASH_SHELL   = 1
```

Observăm că:

- \$\$ rămâne asociat shell-ului inițial;
 - BASHPID identifică procesul Bash curent;
 - BASH_SHELL indică nivelul logic al subshell-ului.
-

Modelul mental

Shell principal

Nivel 0

```
|
▼
```

```
(
  ...
)
```

Nivel 1

```
|
▼
```

```
(
  ...
)
```

Nivel 2

|



```
(  
    ...  
)
```

Nivel 3

Un exemplu practic

Script:

```
afiseaza() {  
    echo "PID=$BASHPID Nivel=$BASH_SUBSHELL"  
}  
  
afiseaza  
  
(  
    afiseaza  
  
    (  
        afiseaza  
    )  
)
```

Acest exemplu evidențiază simultan schimbarea PID-ului și creșterea nivelului subshell-ului.

Caseta „În spatele cortinei”

BASH_SUBSHELL este administrată exclusiv de Bash.

Programatorul nu trebuie să o actualizeze și nici nu ar trebui să îi atribuie manual alte valori. Ea reflectă starea internă a interpretului și este actualizată automat la intrarea și ieșirea din mediile de subshell.

Greșeală din lumea reală

Un administrator scrie:

```
(  
    VAR=100  
)
```

```
echo "$VAR"
```

și este surprins că variabila este goală.

Motivul este simplu.

Comenzile din paranteze au fost executate într-un subshell (`BASH_SUBSHELL=1`), iar modificările asupra variabilelor nu se propagă în shell-ul părinte.

Administrator Notes

`BASH_SUBSHELL` este foarte utilă în timpul depanării.

Dacă observi comportamente diferite între:

- un bloc delimitat de `{ ... ; }`;
- un bloc delimitat de `( ... )`;

afișarea valorii:

```
echo "$BASH_SUBSHELL"
```

te ajută imediat să identifici dacă execuția are loc într-un subshell.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul variabilei `BASH_SUBSHELL`;
 - faptul că valoarea inițială este `0`;
 - că valoarea este incrementată automat pentru fiecare subshell;
 - relația dintre `BASH_SUBSHELL`, `$$` și `BASHPID`.
-

Laborator 15.3

Nivel 1 – Înțelegere

Execută:

```
echo "$BASH_SUBSHELL"
```

Verifică faptul că shell-ul principal are nivelul `0`.

Nivel 2 – Aplicare

Execută:

```
echo "Nivel: $BASH_SUBSHELL"

(
    echo "Nivel: $BASH_SUBSHELL"
)
```

Compară valorile și explică diferența.

Nivel 3 – Administrator

Construiește un script care afișează simultan:

```
echo "$$"
echo "$BASHPID"
echo "$PPID"
echo "$BASH_SUBSHELL"
```

În shell-ul principal și în două subshell-uri imbricate.

Realizează un tabel cu valorile obținute și explică ce se modifică și ce rămâne constant la fiecare nivel de execuție.

Ce trebuie să reții

- BASH_SUBSHELL indică nivelul subshell-ului curent.
- În shell-ul principal valoarea este 0.
- Fiecare subshell incrementează automat valoarea cu unu.
- Împreună cu \$\$, BASHPID și PPID, această variabilă oferă o imagine completă asupra contextului de execuție al Bash.

Capitolul 15

15.4 Variabilele de timp: SECONDS, EPOCHSECONDS și EPOCHREALTIME

„Bash pune la dispoziție mai multe variabile speciale pentru măsurarea timpului. Unele sunt raportate la momentul pornirii shell-ului, iar altele la Unix Epoch.”

Ce vei învăța

În acest capitol vei înțelege diferența dintre SECONDS, EPOCHSECONDS și EPOCHREALTIME, precum și situațiile în care fiecare dintre ele este cea mai potrivită alegere.

Introducere

Foarte multe scripturi au nevoie de informații despre timp.

De exemplu:

- cât durează un backup;
- când a început execuția;
- cât a trecut între două operații;
- generarea unui identificator unic.

Bash oferă trei variabile speciale pentru astfel de situații.

Variabila SECONDS

Execută:

```
echo "$SECONDS"
```

Rezultat posibil:

```
18
```

Valoarea reprezintă **numărul de secunde scurse de la pornirea shell-ului**. De fiecare dată când variabila este referită, Bash calculează valoarea curentă.

Un exemplu

```
echo "$SECONDS"
```

```
sleep 5
```

```
echo "$SECONDS"
```

Rezultat posibil:

```
12  
17
```

Observă că valoarea crește automat.

O caracteristică interesantă

Spre deosebire de multe alte variabile speciale, SECONDS poate primi o valoare.

Exemplu:

```
SECONDS=0
```

```
sleep 3
```

```
echo "$SECONDS"
```

Rezultat:

```
3
```

Conform documentației, după o atribuire, valoarea afișată ulterior este egală cu **numărul de secunde scurse de la atribuire plus valoarea atribuită**.

Acest comportament transformă SECONDS într-un cronometru foarte simplu.

Variabila EPOCHSECONDS

Execută:

```
echo "$EPOCHSECONDS"
```

Rezultat posibil:

```
1785857342
```

Valoarea reprezintă **numărul de secunde scurse de la Unix Epoch (1 ianuarie 1970, 00:00:00 UTC)**. Fiecare referire produce valoarea curentă, iar atribuirea unei valori este ignorată.

Variabila EPOCHREALTIME

Execută:

```
echo "$EPOCHREALTIME"
```

Rezultat posibil:

```
1785857342.481927
```

Diferența este importantă.

EPOCHREALTIME oferă aceeași informație raportată la Unix Epoch, dar cu **rezoluție de microsecunde**, sub forma unui număr în virgulă mobilă. Atribuirile sunt ignorate și pentru această variabilă.

Comparație

Variabilă	Ce măsoară	Precizie
SECONDS	timpul scurs de la pornirea shell-ului	secunde
EPOCHSECONDS	timpul scurs de la Unix Epoch	secunde
EPOCHREALTIME	timpul scurs de la Unix Epoch	microsecunde

Când o folosim?

Pentru cronometrarea unui script

```
SECONDS=0  
  
# operații  
  
echo "Durata: ${SECONDS} secunde"
```

Pentru marcaje temporale

```
echo "$EPOCHSECONDS"
```

Este utilă pentru:

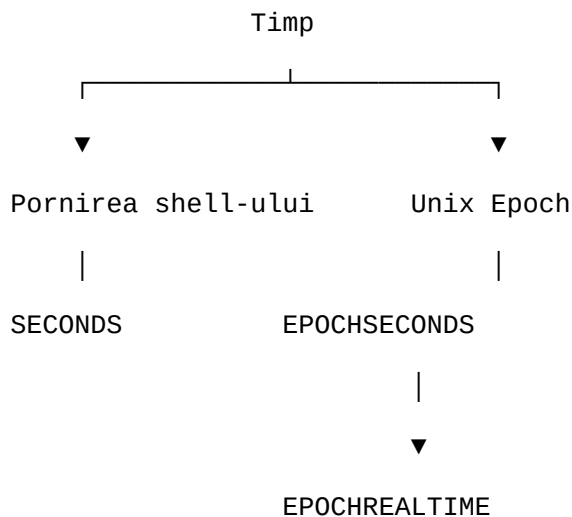
- jurnale;
 - identificatori;
 - compararea momentelor în timp.
-

Pentru măsurători fine

```
echo "$EPOCHREALTIME"
```

Este preferabilă atunci când diferențele de ordinul milisecundelor sau microsecundelor sunt importante.

Modelul mental



Caseta „În spatele cortinei”

SECONDS, EPOCHSECONDS și EPOCHREALTIME nu sunt actualizate de un cronometru care rulează în fundal.

Documentația precizează că Bash determină valoarea lor atunci când sunt evaluate. În plus, EPOCHSECONDS și EPOCHREALTIME ignoră orice atribuire, iar dacă sunt eliminate cu `unset`, își pierd proprietățile speciale.

Greșeală din lumea reală

Un administrator încearcă:

```
EPOCHSECONDS=0
```

și se așteaptă ca timpul să fie resetat.

Acest lucru nu se întâmplă.

Conform documentației Bash, atribuirea unei valori variabilei EPOCHSECONDS este ignorată.

Dacă dorești un cronometru resetabil, folosește SECONDS.

Administrator Notes

O regulă practică este:

- folosește SECONDS pentru măsurarea duratei unui script;
- folosește EPOCHSECONDS pentru marcaje temporale și comparații;

- folosește `EPOCHREALTIME` atunci când ai nevoie de o precizie mai mare.

Astfel alegi variabila potrivită fără a introduce apeluri externe inutile către comenzi precum `date`.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul variabilei `SECONDS`;
 - diferența dintre `SECONDS` și `EPOCHSECONDS`;
 - faptul că `EPOCHREALTIME` oferă precizie la nivel de microsecunde;
 - că atribuirea unei valori este ignorată pentru `EPOCHSECONDS` și `EPOCHREALTIME`, în timp ce `SECONDS` poate fi inițializată.
-

Laborator 15.4

Nivel 1 – Înțelegere

Execută:

```
echo "$SECONDS"
```

```
sleep 3
```

```
echo "$SECONDS"
```

Observă modificarea valorii.

Nivel 2 – Aplicare

Resetează cronometrul:

```
SECONDS=0
```

```
sleep 5
```

```
echo "$SECONDS"
```

Explică de ce rezultatul este aproximativ 5.

Nivel 3 – Administrator

Scrie un script care:

1. memorează valoarea lui:

```
START=$EPOCHSECONDS
```

2. execută câteva operații (de exemplu `sleep 2`);

3. memorează:

```
END=$EPOCHSECONDS
```

4. afișează durata folosind:

```
echo "$((END - START)) secunde"
```

Repetă experimentul utilizând `EPOCHREALTIME` și observă diferența de precizie.

Ce trebuie să reții

- `SECONDS` măsoară timpul scurs de la pornirea shell-ului și poate fi resetată prin atribuire.
- `EPOCHSECONDS` reprezintă numărul de secunde de la Unix Epoch și ignoră atribuirea unei valori.
- `EPOCHREALTIME` oferă aceeași informație cu precizie de microsecunde.
- Alegerea variabilei depinde de scop: cronometrare, marcaj temporal sau măsurători de mare precizie.

Capitolul 15

15.5 Variabilele `RANDOM` și `SRANDOM`

„Bash pune la dispoziție două variabile speciale pentru generarea valorilor pseudoaleatoare: `RANDOM` și `SRANDOM`.”

Ce vei învăța

În acest capitol vei înțelege modul de funcționare al variabilelor `RANDOM` și `SRANDOM`, diferențele dintre ele și situațiile în care este recomandată utilizarea fiecăreia.

Introducere

Numeroase scripturi au nevoie de valori aparent aleatoare.

De exemplu:

- alegerea unui server dintr-o listă;
- selectarea unui test;
- simulări;
- generarea unui identificator temporar.

Bash oferă două variabile speciale pentru astfel de situații:

RANDOM

și

SRANDOM

Variabila RANDOM

Execută:

```
echo "$RANDOM"
```

Rezultat posibil:

21874

Execută din nou:

```
echo "$RANDOM"
```

Rezultat posibil:

1352

Fiecare referire produce o nouă valoare pseudoaleatoare. Conform documentației Bash, intervalul valorilor este cuprins între **0 și 32767**.

Inițializarea generatorului

RANDOM are o proprietate interesantă.

Poate primi o valoare:

```
RANDOM=100
```

Aceasta **inițializează** (seed) generatorul intern.

După aceea, Bash produce aceeași succesiune de valori pentru aceeași valoare inițială.

Acest comportament este foarte util pentru teste reproductibile.

Un exemplu

RANDOM=1

```
echo "$RANDOM"
echo "$RANDOM"
echo "$RANDOM"
```

Dacă repeți experimentul folosind aceeași valoare inițială, vei obține aceeași succesiune de rezultate.

Variabila SRANDOM

Versiunile recente de Bash introduc și:

SRANDOM

Spre deosebire de RANDOM, aceasta utilizează un generator diferit și produce un număr întreg pseudoaleator pe 32 de biți. Documentația precizează că generatorul nu poate fi inițializat prin atribuirea unei valori, iar atribuirea este ignorată.

Exemplu:

```
echo "$SRANDOM"
```

Rezultat posibil:

2845031274

Valorile exacte diferă la fiecare execuție.

Diferența principală

Modelul poate fi rezumat astfel:

Variabilă	Caracteristici
RANDOM	valori între 0 și 32767, poate fi inițializată
SRANDOM	valori pe 32 de biți, nu poate fi inițializată prin atribuire

Obținerea unui interval

Foarte frecvent întâlnim:


```
echo $((RANDOM % 10))
```

Rezultat:

```
0  
...  
9
```

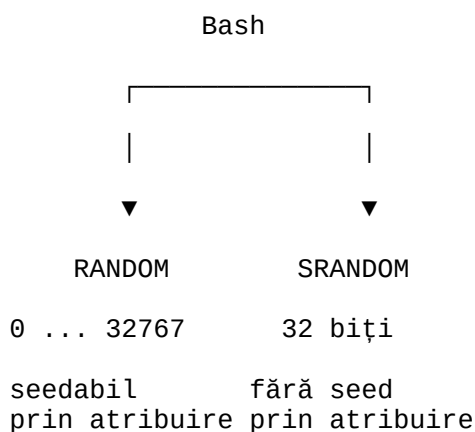
Același principiu poate fi utilizat și pentru alte intervale.

Exemplu:

```
echo $((RANDOM % 6 + 1))
```

simulează aruncarea unui zar.

Modelul mental



Caseta „În spatele cortinei”

Conform documentației, fiecare evaluare a variabilelor `RANDOM` și `SRANDOM` produce automat o nouă valoare.

Dacă una dintre aceste variabile este eliminată folosind:

```
unset RANDOM
```

sau

```
unset SRANDOM
```

iar apoi este creată din nou printr-o atribuire obișnuită, ea își pierde proprietățile speciale și devine o variabilă Bash obișnuită.

Greșeală din lumea reală

Un administrator încearcă:

```
SRANDOM=100
```

și se așteaptă ca următoarele valori generate să fie reproductibile.

Acest lucru nu se întâmplă.

Documentația Bash precizează că atribuirea unei valori lui `SRANDOM` este ignorată.

Dacă ai nevoie de o secvență reproductibilă, utilizează `RANDOM`.

Administrator Notes

În majoritatea scripturilor administrative, `RANDOM` este suficient.

Avantajul său este că poate fi inițializat și produce rezultate reproductibile, utile pentru testare.

`SRANDOM` este potrivit atunci când dorești valori pseudoaleatoare independente de un seed stabilit prin script și când Bash-ul disponibil suportă această variabilă.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul variabilei `RANDOM`;
 - intervalul valorilor generate de `RANDOM`;
 - faptul că `RANDOM` poate fi inițializată prin atribuire;
 - rolul variabilei `SRANDOM`;
 - faptul că atribuirea unei valori lui `SRANDOM` este ignorată.
-

Laborator 15.5

Nivel 1 – Înțelegere

Execută de mai multe ori:

```
echo "$RANDOM"
```

Observă că valoarea se schimbă la fiecare evaluare.

Nivel 2 – Aplicare

Execută:

```
RANDOM=1234
```

```
echo "$RANDOM"  
echo "$RANDOM"  
echo "$RANDOM"
```

Repetă experimentul și verifică dacă obții aceeași succesiune.

Nivel 3 – Administrator

Compară:

```
echo "$RANDOM"  
  
echo "$SRANDOM"
```

Execută comenzile de mai multe ori și observă:

- diferența de ordin de mărime a valorilor;
- faptul că `RANDOM` poate fi inițializată, în timp ce `SRANDOM` nu.

Încearcă apoi:

```
unset RANDOM  
unset SRANDOM
```

și verifică modul în care aceste variabile își pierd comportamentul special după recrearea lor printr-o atribuire obișnuită.

Ce trebuie să reții

- `RANDOM` generează valori pseudoaleatoare între 0 și 32767.
- Atribuirea unei valori lui `RANDOM` inițializează generatorul intern și permite obținerea unor secvențe reproductibile.
- `SRANDOM` generează valori pseudoaleatoare pe 32 de biți și nu poate fi inițializată prin atribuire.
- Dacă sunt eliminate cu `unset`, ambele variabile își pierd proprietățile speciale dacă sunt recreate ulterior ca variabile obișnuite.

Capitolul 15

15.6 Variabilele BASHOPTS și SHELLOPTS

„Variabilele *BASHOPTS* și *SHELLOPTS* reflectă opțiunile active ale shell-ului. *BASHOPTS* este asociată opțiunilor gestionate prin *shopt*, iar *SHELLOPTS* celor controlate prin *set -o*.”

Ce vei învăța

În acest capitol vei înțelege diferența dintre opțiunile gestionate prin *shopt* și cele gestionate prin *set -o*, precum și rolul variabilelor *BASHOPTS* și *SHELLOPTS*.

Introducere

Până acum am utilizat două comenzi importante:

set

și

shopt

Ambele modifică comportamentul Bash.

Dar apare o întrebare firească.

Cum putem afla rapid ce opțiuni sunt active în shell-ul curent?

Bash răspunde prin două variabile speciale:

BASHOPTS

și

SHELLOPTS

Variabila BASHOPTS

Execută:

```
echo "$BASHOPTS"
```

Rezultat posibil:

```
checkwinsize:cmdhist:extglob:histappend:progcomp
```

Variabila conține o listă de opțiuni **shopt** active, separate prin caracterul **:**. Ea este creată automat de Bash și este actualizată pe măsură ce opțiunile sunt activate sau dezactivate.

Relația cu shopt

Execută:

```
shopt -s extglob
```

Apoi verifică:

```
echo "$BASHOPTS"
```

Vei observa apariția opțiunii:

```
extglob
```

în lista afișată.

Variabila SHELLOPTS

Execută:

```
echo "$SHELLOPTS"
```

Rezultat posibil:

```
braceexpand:hashall:interactive-comments
```

Această variabilă conține opțiunile activate prin:

```
set -o
```

și este administrată automat de Bash.

Relația cu set -o

Execută:

```
set -o errexit
```

sau forma echivalentă:

```
set -e
```

Apoi verifică:

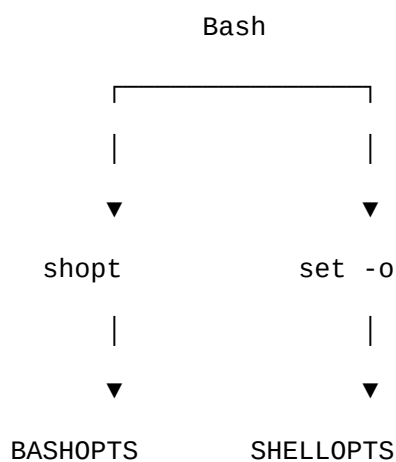
```
echo "$SHELLOPTS"
```

Vei observa că apare opțiunea:

```
errexit
```

Diferența dintre ele

Modelul poate fi reprezentat astfel:



Cele două variabile nu conțin aceleași informații.

Fiecare reflectă propria categorie de opțiuni.

Pot fi modificate?

Nu.

Documentația Bash precizează că aceste variabile sunt speciale și administrate de interpreter. Modificarea directă a conținutului lor nu reprezintă metoda de configurare a shell-ului. Pentru schimbarea opțiunilor trebuie utilizate comenzile:

```
shopt
```

și

```
set -o
```

Modelul mental

Administrator

|



set -o

sau

shopt

|



Bash modifică opțiunile

|



BASHOPTS / SHELLOPTS
se actualizează automat

Caseta „În spatele cortinei”

La pornirea shell-ului, Bash construiește aceste variabile pe baza opțiunilor active. Ele reprezintă o imagine a configurației curente și sunt actualizate automat atunci când opțiunile sunt modificate prin mecanismele oficiale.

Greșeală din lumea reală

Un administrator încearcă:

```
BASHOPTS=extglob
```

și se așteaptă ca opțiunea să fie activată.

Acest lucru nu funcționează.

Forma corectă este:

```
shopt -s extglob
```

În mod similar:

```
SHELLOPTS=errexit
```

nu înlocuiește:

```
set -e
```

Administrator Notes

Atunci când depanezi un script primit de la altcineva, verificarea rapidă a variabilelor:

```
echo "$BASHOPTS"
```

```
echo "$SHELLOPTS"
```

Îți oferă imediat informații despre comportamentul shell-ului fără să parcurgi întregul script în căutarea comenzilor `set` și `shopt`.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul variabilei `BASHOPTS`;
 - rolul variabilei `SHELLOPTS`;
 - relația dintre `BASHOPTS` și `shopt`;
 - relația dintre `SHELLOPTS` și `set -o`;
 - faptul că aceste variabile reflectă starea shell-ului și nu reprezintă mecanismul prin care aceasta este configurată.
-

Laborator 15.6

Nivel 1 – Înțelegere

Afișează:

```
echo "$BASHOPTS"
```

```
echo "$SHELLOPTS"
```

Identifică opțiunile active în shell-ul curent.

Nivel 2 – Aplicare

Activează:


```
shopt -s extglob
```

și observă modificarea variabilei:

```
echo "$BASHOPTS"
```

Apoi activează:

```
set -e
```

și verifică:

```
echo "$SHELLOPTS"
```

Nivel 3 – Administrator

Scrie un script care verifică dacă:

- `extglob` este activă în `BASHOPTS`;
- `errexit` este activă în `SHELLOPTS`;

și afișează un mesaj corespunzător pentru fiecare situație.

Ce trebuie să reții

- `BASHOPTS` conține opțiunile active gestionate prin `shopt`.
- `SHELLOPTS` conține opțiunile active gestionate prin `set -o`.
- Ambele variabile sunt actualizate automat de Bash.
- Pentru modificarea comportamentului shell-ului se utilizează `shopt` și `set -o`, nu atribuirea directă a unor valori acestor variabile.

Capitolul 15

15.7 Variabilele `BASH_ALIASES` și `BASH_CMDS`

„`BASH_ALIASES` este un tablou asociativ ale cărui elemente corespund listei interne de aliasuri gestionate de built-in-ul `alias`. `BASH_CMDS` este un tablou asociativ ale cărui elemente corespund tabelii interne de comenzi gestionate de built-in-ul `hash`.”

Ce vei învăța

În acest capitol vei înțelege cum păstrează Bash aliasurile și comenzile memorate în tabela de hash și cum pot fi inspectate folosind variabile speciale.

Introducere

Până acum am folosit comenzile:

```
alias
```

și

```
hash
```

Ambele creează informații pe care Bash le păstrează în memorie.

Dar unde sunt acestea stocate?

Documentația oficială arată că Bash le expune prin două tablouri asociative:

```
BASH_ALIASES
```

și

```
BASH_CMDS
```

Variabila BASH_ALIASES

BASH_ALIASES este un **tablou asociativ**.

Cheia reprezintă numele aliasului, iar valoarea reprezintă comanda asociată.

Exemplu:

```
alias ll='ls -aLF'
```

După definire, Bash actualizează automat și tabloul:

```
declare -p BASH_ALIASES
```

Rezultatul poate avea forma:

```
declare -A BASH_ALIASES=(  
  [ll]="ls -aLF"  
)
```

Conform documentației, elementele adăugate în acest tablou apar și în lista de aliasuri administrată de built-in-ul `alias`.

Relația cu alias

Putem verifica:

```
alias ll='ls -al'
```

```
alias
```

și:

```
declare -p BASH_ALIASES
```

Ambele reflectă aceeași informație.

Diferența este doar modul de prezentare.

Variabila BASH_CMDS

A doua structură este:

```
BASH_CMDS
```

Ea este tot un tablou asociativ.

Elementele sale corespund **tabelei interne de hash** utilizate de Bash pentru localizarea rapidă a comenzilor executate anterior.

Această tabelă este administrată și prin built-in-ul:

```
hash
```

Relația cu hash

Execută:

```
hash
```

Vei observa comenzile memorate de Bash.

Aceleași informații sunt disponibile și prin:

```
declare -p BASH_CMDS
```

De ce există această tabelă?

Atunci când execuți pentru prima dată:

`grep`

Bash caută executabilul în directoarele din variabila:

`PATH`

După ce îl găsește, memorează rezultatul într-o tabelă internă.

La următoarele execuții nu mai este necesară parcurgerea întregului `PATH`, ceea ce reduce timpul necesar rezolvării comenzii.

`BASH_CMDS` oferă acces la această structură internă.

Pot fi modificate?

Documentația precizează un comportament interesant.

Dacă adaugi un element în:

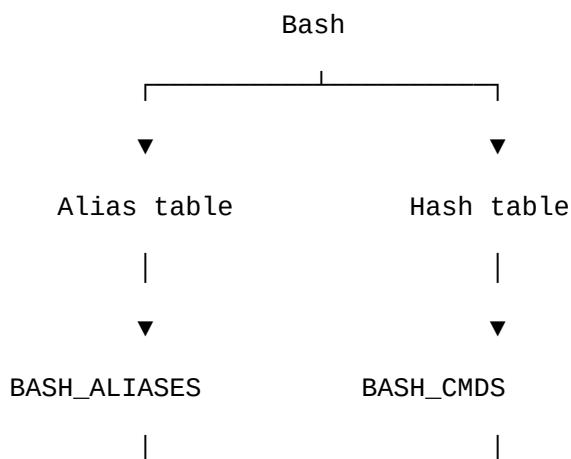
`BASH_ALIASES`

acesta apare și în lista aliasurilor.

În schimb, dacă elimini un element din tablou folosind `unset`, aliasul nu este eliminat automat din lista internă. Același comportament este descris și pentru `BASH_CMDS`: eliminarea unui element din tablou nu șterge automat intrarea din tabela de hash.

Cu alte cuvinte, sincronizarea este completă la adăugare, dar nu și la ștergere.

Modelul mental



▼
alias builtin

▼
hash builtin

Caseta „În spatele cortinei”

Atât BASH_ALIASES, cât și BASH_CMDS sunt **tablouri asociative**, nu tablouri indexate.

Cheile nu sunt numere (0, 1, 2), ci șiruri de caractere, cum ar fi numele unui alias sau al unei comenzi. Dacă aceste variabile sunt eliminate cu `unset`, ele își pierd proprietățile speciale, chiar dacă sunt recreate ulterior.

Greșeală din lumea reală

Un administrator execută:

```
unset 'BASH_ALIASES[ll]'
```

și se așteaptă ca aliasul:

```
ll
```

să dispară.

Acest lucru nu se întâmplă.

Forma corectă este:

```
unalias ll
```

În mod similar, pentru tabela comenzilor trebuie utilizat built-in-ul:

```
hash
```

și nu modificarea directă a tabloului BASH_CMDS.

Administrator Notes

Aceste două variabile sunt utile în special pentru:

- depanarea comportamentului Bash;
- înțelegerea modului în care shell-ul gestionează aliasurile;
- analiza mecanismului de căutare și memorare a comenzilor.

În administrarea de zi cu zi, comenzile `alias`, `unalias` și `hash` rămân interfața recomandată.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- că BASH_ALIASES este un tablou asociativ legat de built-in-ul alias;
 - că BASH_CMDS este un tablou asociativ legat de built-in-ul hash;
 - faptul că adăugarea elementelor este reflectată în structurile interne ale Bash;
 - faptul că eliminarea elementelor din aceste tablouri nu elimină automat aliasurile sau intrările din tabela de hash.
-

Laborator 15.7

Nivel 1 – Înțelegere

Definește un alias:

```
alias ll='ls -aF'
```

Apoi execută:

```
declare -p BASH_ALIASES
```

Identifică aliasul nou creat.

Nivel 2 – Aplicare

Execută câteva comenzi uzuale:

```
ls  
pwd  
grep --help
```

Apoi verifică tabela de hash:

```
hash
```

și compar-o cu:

```
declare -p BASH_CMDS
```

Observă că ambele reflectă comenzile memorate de Bash.

Nivel 3 – Administrator

Creează un script care:

1. definește mai multe aliasuri;
2. afișează conținutul lui `BASH_ALIASES`;
3. execută câteva comenzi externe;
4. afișează conținutul lui `BASH_CMDS`.

Explică rolul fiecărei structuri și de ce Bash utilizează o tabelă de hash pentru accelerarea căutării executabilelor.

Ce trebuie să reții

- `BASH_ALIASES` este un tablou asociativ care reflectă lista internă de aliasuri gestionată de `alias`.
- `BASH_CMDS` este un tablou asociativ care reflectă tabela internă de hash gestionată de hash.
- Adăugarea elementelor în aceste tablouri este reflectată în structurile interne ale Bash.
- Eliminarea elementelor prin `unset` nu elimină automat aliasurile sau intrările din tabela de hash.

Capitolul 15

15.8 Variabila `DIRSTACK`

„`DIRSTACK` este un tablou care conține conținutul curent al stivei de directoare. Ordinea elementelor este aceeași cu cea afișată de built-in-ul `dirs`.”

Ce vei învăța

În acest capitol vei înțelege cum păstrează Bash stiva de directoare și cum poate fi inspectată prin intermediul variabilei speciale `DIRSTACK`.

Introducere

În capitolul dedicat comenzilor:

`pushd`

```
popd
dirs
```

am învățat că Bash poate memora mai multe directoare într-o stivă.

Dar unde este păstrată această informație?

Răspunsul este:

```
DIRSTACK
```

Ce este DIRSTACK?

DIRSTACK este un **tablou** (array).

El conține toate directoarele existente în stiva internă administrată de Bash.

Putem vedea conținutul lui astfel:

```
declare -p DIRSTACK
```

Exemplu:

```
declare -a DIRSTACK=(
  [0]="/home/student"
  [1]="/etc"
  [2]="/var/log"
)
```

Relația cu dirs

Execută:

```
dirs -v
```

Rezultat posibil:

```
0  /home/student
1  /etc
2  /var/log
```

Observă că ordinea este identică cu cea din DIRSTACK. Documentația Bash precizează explicit acest lucru.

Accesarea elementelor

Fiind un tablou, putem utiliza sintaxa obișnuită:


```
echo "${DIRSTACK[0]}"
```

sau:

```
printf '%s\n' "${DIRSTACK[@]}"
```

Aceste forme permit inspectarea conținutului fără a apela comanda `dirs`.

Modificarea unui element

Documentația Bash precizează că putem modifica un element existent:

```
DIRSTACK[1]="/opt"
```

Această atribuire modifică elementul din stivă.

Totuși, există o limitare foarte importantă.

Ce NU face **DIRSTACK**

Presupunem:

```
pwd
```

Rezultat:

```
/home/student
```

Apoi executăm:

```
DIRSTACK[0]="/etc"
```

Directorul curent rămâne:

```
/home/student
```

Documentația este foarte clară:

Atribuirea unei valori lui `DIRSTACK` nu modifică directorul curent.

Pentru schimbarea directorului trebuie utilizată comanda:

```
cd
```

Adăugarea și eliminarea directoarelor

Acesta este un alt detaliu important.

Nu trebuie să folosim:

```
DIRSTACK+=(/tmp)
```

pentru a adăuga un director în stivă.

Documentația precizează că **pushd** și **popd** sunt mecanismele corecte pentru adăugarea și eliminarea elementelor.

Modelul mental

```
DIRSTACK
+-----+
[0] -> | /home/student |
[1] -> | /etc           |
[2] -> | /var/log       |
+-----+
      ▲
      |
dirs / pushd / popd
```

Relația cu pushd

Execută:

```
pushd /tmp
```

Apoi:

```
declare -p DIRSTACK
```

Vei observa că Bash actualizează automat tabloul.

Nu este nevoie să modifichi manual variabila.

Caseta „În spatele cortinei”

DIRSTACK este doar o reprezentare a stivei interne de directoare.

Interpreterul Bash gestionează această structură și o sincronizează cu comenzile `pushd`, `popd` și `dirs`. Dacă variabila este eliminată folosind:

```
unset DIRSTACK
```

ea își pierde proprietățile speciale, chiar dacă este creată din nou ulterior.

Greșeală din lumea reală

Un administrator încearcă:

```
DIRSTACK+=("/backup")
```

și se așteaptă ca:

```
dirs
```

să afișeze noul director.

Rezultatul este imprevizibil și nu reprezintă modul recomandat de administrare a stivei.

Forma corectă este:

```
pushd /backup
```

iar pentru eliminare:

```
popd
```

Acestea sunt mecanismele descrise și recomandate de documentația Bash.

Administrator Notes

În practică, `DIRSTACK` este folosită rar pentru modificarea directă a elementelor.

Este însă foarte utilă pentru:

- depanare;
- afișarea stivei într-un script;
- verificarea rapidă a conținutului fără apelarea lui `dirs`.

Pentru manipularea stivei, utilizează întotdeauna:

- `pushd`;
 - `popd`;
 - `dirs`.
-

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- că DIRSTACK este un tablou;
 - relația sa cu dirs;
 - faptul că pushd și popd sunt metodele recomandate pentru modificarea stivei;
 - că modificarea lui DIRSTACK nu schimbă directorul curent.
-

Laborator 15.8

Nivel 1 – Înțelegere

Execută:

```
pushd /tmp  
pushd /etc
```

```
dirs -v
```

Apoi afișează:

```
declare -p DIRSTACK
```

Compară ordinea elementelor.

Nivel 2 – Aplicare

Modifică un element existent:

```
DIRSTACK[1]="/opt"
```

Afișează din nou:

```
dirs -v
```

și verifică dacă elementul a fost actualizat.

Execută apoi:

```
pwd
```

Observă că directorul curent nu s-a schimbat.

Nivel 3 – Administrator

Construiește un script care:

1. utilizează `pushd` pentru a crea o stivă cu mai multe directoare;
2. afișează conținutul lui `DIRSTACK`;
3. parcurge tabloul:

```
for dir in "${DIRSTACK[@]}"
do
    echo "$dir"
done
```

4. elimină elementele folosind `popd`.

Explică de ce manipularea stivei prin `pushd` și `popd` este preferabilă modificării directe a tabloului.

Ce trebuie să reții

- `DIRSTACK` este un tablou care reflectă stiva internă de directoare a Bash.
- Ordinea elementelor este aceeași cu cea afișată de `dirs`.
- `pushd` și `popd` sunt mecanismele recomandate pentru adăugarea și eliminarea elementelor din stivă.
- Modificarea lui `DIRSTACK` nu schimbă directorul curent.
- Dacă variabila este eliminată cu `unset`, își pierde proprietățile speciale.

Capitolul 15

15.9 Identitatea sistemului și a utilizatorului

„Bash pune la dispoziția scripturilor variabile speciale care descriu gazda pe care rulează, tipul sistemului și identitatea utilizatorului curent.”

Ce vei învăța

În acest capitol vei înțelege cum poate un script să afle pe ce sistem rulează și sub ce identitate este executat.

Introducere

Foarte multe scripturi trebuie să răspundă unor întrebări simple:

- Pe ce calculator rulez?
- Ce sistem de operare este acesta?
- Ce arhitectură are?
- Cu ce utilizator sunt executat?
- Am privilegii de administrator?

Bash oferă răspunsurile printr-un grup de variabile speciale.

Variabila HOSTNAME

Execută:

```
echo "$HOSTNAME"
```

Rezultat posibil:

```
server01
```

Conform documentației, această variabilă este inițializată automat cu numele gazdei curente.

Este utilă pentru:

- jurnale;
 - identificarea serverului;
 - afișarea promptului.
-

Variabila HOSTTYPE

Execută:

```
echo "$HOSTTYPE"
```

Rezultat posibil:

```
x86_64
```

HOSTTYPE descrie tipul mașinii pe care rulează Bash. Valoarea exactă depinde de platforma pe care a fost compilat interpreterul.

Variabila MACHTYPE

Execută:

```
echo "$MACHTYPE"
```

Rezultat posibil:

```
x86_64-pc-linux-gnu
```

Spre deosebire de HOSTTYPE, MACHTYPE oferă o descriere completă a platformei în formatul GNU:

```
CPU-FURNIZOR-SISTEM
```

Documentația precizează că valoarea este dependentă de sistemul pe care rulează Bash.

Variabila OSTYPE

Execută:

```
echo "$OSTYPE"
```

Rezultat posibil:

```
linux-gnu
```

OSTYPE descrie sistemul de operare.

Exemple posibile:

```
linux-gnu  
darwin  
freebsd
```

Valoarea exactă depinde de platformă.

Variabila UID

Execută:

```
echo "$UID"
```

Rezultat posibil:

```
1000
```

UID reprezintă identificatorul numeric al utilizatorului real care rulează shell-ul.

Această variabilă este doar pentru citire (readonly).

Variabila EUID

Execută:

```
echo "$EUID"
```

Rezultat posibil:

```
1000
```

EUID reprezintă identificatorul efectiv al utilizatorului curent și este inițializat la pornirea shell-ului. Și această variabilă este readonly.

În cele mai multe situații:

```
UID == EUID
```

dar ele pot diferi în contexte speciale, cum ar fi executarea unor programe cu bitul **setuid**.

Variabila GROUPS

Execută:

```
declare -p GROUPS
```

Rezultat posibil:

```
declare -a GROUPS=( [0]="1000" [1]="27" [2]="4" )
```

GROUPS este un **tablou** care conține identificatorii grupurilor din care face parte utilizatorul curent. Atribuirile asupra acestei variabile nu au efect.

Comparație

Variabilă	Descriere
HOSTNAME	numele gazdei
HOSTTYPE	tipul mașinii
MACHTYPE	descriere completă a platformei
OSTYPE	sistemul de operare
UID	identificatorul utilizatorului real
EUID	identificatorul utilizatorului efectiv

Variabilă	Descriere
GROUPS	tabloul grupurilor utilizatorului

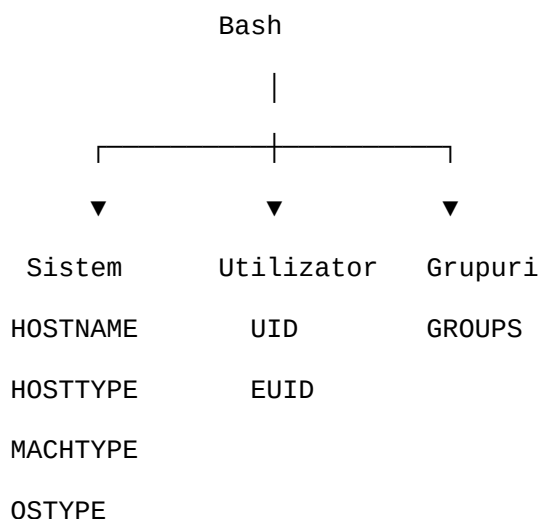
Exemplu practic

Foarte multe scripturi administrative verifică dacă sunt executate cu privilegii de administrator:

```
if (( EUID != 0 ))
then
    echo "Acest script trebuie executat ca root."
    exit 1
fi
```

Această verificare este preferată în locul comparării numelui utilizatorului, deoarece identificatorul numeric este mai sigur și independent de localizare.

Modelul mental



Caseta „În spatele cortinei”

Aceste variabile sunt inițializate automat la pornirea shell-ului pe baza informațiilor furnizate de sistemul de operare. Unele dintre ele (UID, EUID) sunt readonly, iar GROUPS este un tablou special ale cărui atribuiri sunt ignorate. Dacă GROUPS este eliminată cu `unset`, își pierde proprietățile speciale.

Greșeală din lumea reală

Un administrator verifică privilegiile astfel:

```
if [ "$USER" = "root" ]
```

Scriptul funcționează în multe situații, dar se bazează pe un nume de utilizator.

Forma recomandată este:

```
if (( EUID == 0 ))
```

deoarece verifică direct identitatea efectivă a procesului.

Administrator Notes

În scripturile de producție este recomandat să folosești:

- HOSTNAME pentru identificarea serverului;
- OSTYPE și MACHTYPE pentru adaptarea comportamentului la platformă;
- EUID pentru verificarea privilegiilor;
- GROUPS atunci când accesul depinde de apartenența la anumite grupuri.

Aceste variabile sunt mai eficiente decât apelarea repetată a comenzilor externe precum `hostname`, `id` sau `uname`, deoarece informațiile sunt deja disponibile în shell.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul variabilelor HOSTNAME, HOSTTYPE, MACHTYPE și OSTYPE;
 - diferența dintre UID și EUID;
 - faptul că GROUPS este un tablou cu grupurile utilizatorului și că atribuirea asupra lui nu are efect.
-

Laborator 15.9

Nivel 1 – Înțelegere

Afișează:

```
echo "$HOSTNAME"  
echo "$HOSTTYPE"
```

```
echo "$MACHTYPE"  
echo "$OSTYPE"
```

Identifică informațiile despre sistemul pe care rulează shell-ul.

Nivel 2 – Aplicare

Execută:

```
echo "UID=$UID"  
echo "EUID=$EUID"
```

```
declare -p GROUPS
```

Compară valorile și identifică grupurile din care face parte utilizatorul curent.

Nivel 3 – Administrator

Scrie un script care:

1. verifică dacă este executat cu privilegii de administrator folosind EUID;
 2. afișează HOSTNAME și OSTYPE;
 3. listează toate grupurile din GROUPS;
 4. salvează aceste informații într-un fișier jurnal.
-

Ce trebuie să reții

- HOSTNAME, HOSTTYPE, MACHTYPE și OSTYPE descriu sistemul pe care rulează Bash.
- UID identifică utilizatorul real, iar EUID utilizatorul efectiv.
- GROUPS este un tablou cu grupurile utilizatorului, iar atribuirea asupra lui este ignorată.
- Pentru verificarea privilegiilor într-un script, EUID este variabila recomandată.

Capitolul 15

15.10 Variabilele SHLVL, PWD și OLDPWD

„Bash menține automat informații despre nivelul shell-ului și despre directorul curent prin intermediul variabilelor speciale SHLVL, PWD și OLDPWD.”

Ce vei învăța

În acest capitol vei înțelege cum urmărește Bash nivelul shell-ului și cum gestionează directorul curent și directorul anterior.

Introducere

De fiecare dată când deschizi un terminal, Bash cunoaște:

- unde te afli în sistemul de fișiere;
- din ce director ai venit;
- dacă shell-ul curent este primul sau unul lansat din alt shell.

Aceste informații sunt disponibile prin trei variabile speciale.

Variabila PWD

Execută:

```
echo "$PWD"
```

Rezultat posibil:

```
/home/student
```

PWD conține **calea absolută a directorului de lucru curent**. Bash o actualizează automat la fiecare schimbare de director.

Relația cu pwd

Comanda:

```
pwd
```

și variabila:

```
echo "$PWD"
```

afișează, în mod normal, aceeași informație.

Totuși, utilizarea variabilei este mai eficientă deoarece nu necesită lansarea unei comenzi externe.

Variabila OLDPWD

Execută:

```
cd /etc
```

```
echo "$OLDPWD"
```

Rezultat posibil:

```
/home/student
```

OLDPWD conține **directorul anterior** și este actualizată automat după fiecare comandă `cd`.

Acesta este mecanismul utilizat de comanda:

```
cd -
```

care revine în directorul precedent.

Exemplu

```
pwd
```

```
cd /tmp
```

```
echo "$OLDPWD"
```

```
cd -
```

```
pwd
```

Observă că `cd -` folosește valoarea din `OLDPWD` pentru a reveni în directorul anterior.

Variabila SHLVL

Execută:

```
echo "$SHLVL"
```

Rezultat posibil:

```
1
```

SHLVL indică **nivelul shell-ului curent**.

La pornirea unui nou shell Bash dintr-un shell Bash existent:

```
bash
```

valoarea devine:

2

Dacă pornești încă un shell:

bash

valoarea devine:

3

Documentația precizează că Bash incrementează automat această variabilă la pornirea unui nou shell.

Relația dintre SHLVL și BASH_SUBSHELL

Aceste variabile sunt adesea confundate.

Ele descriu lucruri diferite.

Variabilă	Descrie
SHLVL	câte shell-uri Bash au fost lansate unul din altul
BASH_SUBSHELL	nivelul subshell-ului în shell-ul curent

Exemplu:

Terminal

↓

bash

SHLVL=1

↓

bash

SHLVL=2

În schimb:

```
(  
    echo "$BASH_SUBSHELL"  
)
```

afișează nivelul subshell-ului fără a modifica SHLVL.

Modelul mental

Terminal

|



bash

SHLV L=1

PWD=/home/student

OLDPWD=/home

|



bash

SHLV L=2

Caseta „În spatele cortinei”

La fiecare execuție a comenzii `cd`, Bash actualizează mai întâi `OLDPWD`, apoi `PWD`. La pornirea unui nou proces Bash, `SHLV L` este incrementată automat. Dacă această valoare devine anormal de mare, poate indica lansarea recursivă a unor shell-uri.

Greșeală din lumea reală

Un administrator dorește să afle dacă rulează într-un subshell și verifică:

```
echo "$SHLV L"
```

Valoarea nu se modifică după:

```
(  
    ls  
)
```

Motivul este simplu.

`SHLV L` urmărește **shell-urile Bash lansate unul din altul**, nu subshell-urile create de interpret în timpul execuției.

Pentru acestea trebuie utilizată variabila:

```
echo "$BASH_SUBSHELL"
```

Administrator Notes

Aceste trei variabile apar foarte des în scripturi:

- PWD pentru construirea căilor absolute;
- OLDPWD pentru revenirea la directorul anterior;
- SHLVL pentru diagnosticarea mediului de execuție.

În majoritatea cazurilor este preferabil să folosești aceste variabile în locul apelării repetate a comenzilor `pwd` sau a altor utilitare externe.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul variabilei PWD;
 - rolul variabilei OLDPWD;
 - utilizarea comenzii `cd -`;
 - rolul variabilei SHLVL;
 - diferența dintre SHLVL și BASH_SUBSHELL.
-

Laborator 15.10

Nivel 1 – Înțelegere

Execută:

```
echo "$PWD"
```

```
cd /tmp
```

```
echo "$PWD"
```

```
echo "$OLDPWD"
```

Observă modificarea celor două variabile.

Nivel 2 – Aplicare

Execută:

```
echo "$SHLVL"
```

```
bash
```



```
echo "$SHLVL"
```

Ieși din shell folosind:

```
exit
```

și explică modificarea valorii.

Nivel 3 – Administrator

Construiește un script care:

- afișează PWD;
- afișează OLDPWD;
- afișează SHLVL;
- execută câteva schimbări de directoare și pornește un shell Bash nou.

Analizează modul în care fiecare variabilă se modifică în timpul execuției.

Ce trebuie să reții

- PWD conține directorul curent și este actualizată automat de Bash.
- OLDPWD memorează directorul anterior și este utilizată de `cd -`.
- SHLVL indică nivelul shell-ului Bash și este incrementată la lansarea unui nou shell.
- SHLVL și BASH_SUBSHELL descriu concepte diferite și nu trebuie confundate.

Capitolul 15

15.11 Variabilele LINENO și PIPESTATUS

„LINENO se extinde la numărul liniei curente din script sau funcție. PIPESTATUS este un tablou care conține codurile de ieșire ale ultimelor comenzi executate într-o conductă.”

Ce vei învăța

În acest capitol vei înțelege cum poate un script să afle poziția curentă în cod și cum poate verifica rezultatul fiecărei comenzi dintr-un pipeline.

Introducere

Două întrebări apar frecvent în timpul depanării unui script:

- La ce linie s-a produs eroarea?
- Care dintre comenzile dintr-o conductă a eșuat?

Bash oferă răspunsul prin două variabile speciale:

`LINENO`

și

`PIPESTATUS`

Variabila `LINENO`

Execută:

```
echo "$LINENO"
```

Rezultatul poate fi:

12

Valoarea reprezintă **linia din script la care este evaluată variabila**. Ea este calculată automat de Bash în timpul execuției.

Un exemplu

Script:

```
echo "Prima linie"
```

```
echo "$LINENO"
```

```
echo "Ultima linie"
```

Dacă `echo "$LINENO"` se află pe linia 3, rezultatul va fi:

3

Mutarea instrucțiunii pe altă linie va modifica automat valoarea.

La ce este utilă?

LINENO este folosită frecvent în mesajele de eroare:

```
echo "Eroare la linia $LINENO"
```

sau împreună cu:

```
trap
```

pentru a construi mesaje de diagnostic mai clare.

Variabila PIPESTATUS

Să presupunem că executăm:

```
grep "root" /etc/passwd | sort | wc -l
```

După terminarea conducerii:

```
echo "${PIPESTATUS[@]}"
```

Rezultat posibil:

```
0 0 0
```

PIPESTATUS este un **tablou** care conține codul de ieșire al fiecărei comenzi din ultimul pipeline executat.

Un exemplu cu eroare

Execută:

```
grep "inexistent" /etc/passwd | sort | wc -l
```

Apoi:

```
echo "${PIPESTATUS[@]}"
```

Rezultat posibil:

```
1 0 0
```

Interpretarea este:

Comandă	Exit status
grep	1
sort	0

Comandă Exit status

```
wc 0
```

Astfel putem identifica exact comanda care a produs un cod diferit de zero.

Relația cu \$?

Foarte important:

```
echo $?
```

afișează **doar codul ultimei comenzi** din pipeline.

În schimb:

```
echo "${PIPESTATUS[@]}"
```

afișează codurile **tuturor** comenzilor.

Exemplu:

```
grep | sort | wc
```

```
| | |
```

```
▼ ▼ ▼
```

```
1 0 0
```

```
$? = 0
```

```
PIPESTATUS = 1 0 0
```

Acesta este motivul existenței lui PIPESTATUS.

Relația cu pipefail

Într-un capitol anterior am studiat:

```
set -o pipefail
```

Această opțiune modifică valoarea întoarsă de întregul pipeline.

Totuși:

```
PIPESTATUS
```

continuă să conțină codul fiecărei comenzi individuale.

Cele două mecanisme sunt complementare.

Modelul mental

Pipeline

grep

|

▼

sort

|

▼

wc

|

▼

PIPESTATUS

[1] [0] [0]

Caseta „În spatele cortinei”

PIPESTATUS este actualizată după fiecare pipeline executat.

Dacă rulezi o altă comandă:

```
echo "Test"
```

conținutul lui PIPESTATUS va fi înlocuit cu rezultatul ultimei execuții. Dacă dorești să păstrezi valorile, salvează-le imediat într-o altă variabilă sau într-un tablou.

Greșeală din lumea reală

Un administrator execută:

```
grep "abc" fisier | sort
```

```
echo "OK"
```

```
echo "${PIPESTATUS[@]}"
```

Rezultatul nu mai descrie conducta inițială.

Motivul este că între timp a fost executată comanda:

```
echo "OK"
```

care a modificat starea ultimelor comenzi executate.

Forma recomandată este:

```
grep "abc" fisier | sort
```

```
status=("${PIPESTATUS[@]}")
```

imediat după pipeline.

Administrator Notes

În scripturile complexe:

- `LINENO` este foarte utilă pentru jurnalizarea erorilor;
- `PIPESTATUS` este esențială atunci când trebuie analizat rezultatul fiecărei comenzi dintr-o conductă.

Împreună cu:

- `FUNCNAME`;
- `BASH_SOURCE`;
- `BASH_LINENO`;

aceste variabile permit construirea unor mecanisme de diagnostic foarte detaliate.

Legătura cu LPIC

Pentru examen trebuie să cunoști:

- rolul variabilei `LINENO`;
 - faptul că `PIPESTATUS` este un tablou;
 - diferența dintre `$?` și `PIPESTATUS`;
 - relația dintre `PIPESTATUS` și opțiunea `pipefail`.
-

Laborator 15.11

Nivel 1 – Înțelegere

Scrie un script care afișează:

```
echo "Linia curentă: $LINENO"
```

în mai multe locuri și observă modificarea valorii.

Nivel 2 – Aplicare

Execută:

```
grep root /etc/passwd | sort | wc -l
```

```
echo "${PIPESTATUS[@]}"
```

Apoi repetă experimentul folosind un șir care nu există și compară rezultatele.

Nivel 3 – Administrator

Construiește un script care:

1. execută un pipeline format din trei comenzi;
2. salvează imediat:

```
status=("${PIPESTATUS[@]}")
```

3. afișează codul de ieșire pentru fiecare comandă;
 4. generează un mesaj de eroare dacă oricare dintre valorile din `status` este diferită de 0.
-

Ce trebuie să reții

- `LINENO` indică numărul liniei curente din script sau funcție.
- `PIPESTATUS` este un tablou care conține codurile de ieșire ale fiecărei comenzi din ultimul pipeline.
- `$?` oferă doar codul ultimei comenzi, în timp ce `PIPESTATUS` păstrează rezultatele tuturor comenzilor din conductă.
- Dacă ai nevoie de valorile din `PIPESTATUS`, salvează-le imediat după executarea pipeline-ului.