DRAGOȘ TATOMIRESCU OANA IRIZOIU

SISTEME DE OPERARE

LUCRĂRI PRACTICE

Manualul Studentului

Editura POLITEHNICA

Colecţia "MANUALUL STUDENTULUI"

SISTEME DE OPERARE

LUCRĂRI PRACTICE

Lucrarea abordează un segment important în pregătirea științifică a viitorilor ingineri, și anume gestionarea sistemelor de operare pe diverse dispozitive. Astfel, devine o unealtă necesară în arsenalul cadrelor didactice dintr-o universitate cu specific tehnic. Cartea acoperă o serie de concepte esențiale în domeniul sistemelor de operare, oferind studenților suportul necesar pentru înțelegerea și aplicarea practică a noțiunilor teoretice.

Referent ştiinţific: Conf. univ. dr. Ing. Attila SIMÓ

Acest volum completează cursul de Sisteme de Operare şi, parcurgându-l, studenții sunt capabili să învețe să folosească apelurile moderne ale sistemului de operare și bibliotecile de sincronizare în interfețe software/hardware; elementele de bază ale comenzilor UNIX și implementarea programării în linia de comandă; să aplice concepte tehnice, să analizeze, să sintetizeze date pentru a proiecta și crea produse și soluții noi pentru problemele din viața reală.

Referent ştiinţific: Lector univ. dr. Alexandra POPESCU

DRAGOŞ TATOMIRESCU OANA IRIZOIU

SISTEME DE OPERARE

LUCRĂRI PRACTICE

Colecţia "MANUALUL STUDENTULUI"

EDITURA POLITEHNICA
TIMIŞOARA – 2024

Copyright © Editura Politehnica, 2024

Nicio parte din această lucrare nu poate fi reprodusă, stocată sau transmisă prin indiferent ce formă, fără acordul prealabil scris al Editurii Politehnica.

EDITURA POLITEHNICA
Bd. Republicii nr. 9
300159 Timişoara, România

Tel. 0256.403.822
E-mail: editura@upt.ro

Redactor: Claudia MIHALI

Bun de imprimat: 27.09.2024
Coli de tipar: 11
ISBN 978-606-35-0600-0

Tiparul executat sub comanda nr. 22
la Tipografia Universităţii Politehnica Timişoara

CUPRINS

PREFAȚĂ ... 9

1. INSTALAREA SISTEMULUI DE OPERARE WINDOWS 11 ÎN ORACLE VM VIRTUALBOX ... 11

2. INSTALAREA SISTEMULUI DE OPERARE UBUNTU 22 ÎN ORACLE VM VIRTUALBOX ... 20

3. COMENZI UTILE ÎN SISTEME LINUX – GESTIONAREA FOLDERELOR ... 27

3.1. Crearea de foldere și fișiere ... 32

3.2. Crearea de fișiere folosind redirecționare ... 34

3.3. Mutarea și manipularea fișierelor ... 36

3.4. Ștergerea fișierelor și folderelor ... 39

4. COMENZI UTILE ÎN SISTEME LINUX - LINIA DE COMANDĂ, SUPERUTILIZATORUL ȘI FIȘIERELE ASCUNSE ... 41

4.1. Linia de comandă și superutilizatorul ... 44

4.2. Fișiere ascunse ... 47

5. COMENZI UTILE ÎN SISTEME LINUX – PERMISIUNI ȘI CONTROLUL PROCESELOR ... 50

5.1. Permisiuni ... 50

5.1.1. Permisiuni pentru fișiere ... 51

5.1.2. Permisiuni pentru directoare ... 53

5.1.3. Trecerea la Superutilizator pentru o scurtă perioadă ... 53

5.1.4. Modificarea dreptului de proprietate asupra fișierului ... 55

5.1.5. Schimbarea grupului de proprietari ... 55

5.2. Controlul proceselor ... 55

5.2.1. Punerea unui program în fundal ... 56

5.2.2. Listarea proceselor în derulare ... 57

5.2.3. Terminarea unui proces ... 58

6. SCRIPTURI SHELL ÎN SISTEME LINUX – EDITOARE DE TEXT, SCRIPTURILE IMPLICITE ȘI CONSTRUIREA UNEI APLICAȚII ... 60

6.1. Scrierea unui Script (editoare de text) ... 60

6.2. Editarea scripturilor pe care le avem deja ... 64

6.3. Construirea unei aplicații ... 68

7. SCRIPTURI SHELL ÎN SISTEME LINUX - VARIABILE, CONSTANTE ȘI FUNCȚII 72

7.1. Variabile ... 72

7.2. Variabile de mediu 74
7.3. Înlocuirea comenzilor și constantele 76
7.4. Atribuirea unui rezultat al unei comenzi către o variabilă 77
7.5. Constante 77
7.6. Funcții Shell 78
7.7. Păstrarea funcțională a scripturilor 81
7.8. Functia show_uptime 83
7.9. Functia drive_space 84
7.10. Functia home_space 84
7.11. Functia system_info 85
8. SCRIPTURI SHELL ÎN SISTEME LINUX – CONTROLUL FLUXULUI DE EXECUȚIE: RAMIFICARE CU IF 86
8.1. Exit Status 86
8.2. Testarea pentru Root 90
8.3. “Stay Out of Trouble” 91
8.4. Variabile goale 92
8.5. Ghilimele lipsă 93
8.6. Izolarea problemelor 94
8.7. Urmărirea rulării scriptului 95
8.8. Intrare de la tastatură 96
8.9. Aritmetică 98
9. SCRIPTURI SHELL ÎN SISTEME LINUX - CONTROLUL FLUXULUI DE EXECUȚIE: BUCLE CU WHILE / UNTIL 102
9.1. Bucle (Loops) 105
9.2. Construirea unui meniu 107
9.3. Când computerul se blochează... 110
9.4. Parametrii de poziție 111
9.5. Opțiuni pentru linia de comandă 115
9.6. Obținerea argumentului unei opțiuni 116
9.7. Integrarea procesorului de linie de comandă în script 117
9.8. Adăugarea modului interactiv 119
10. SCRIPTURI SHELL ÎN SISTEME LINUX - CONTROLUL FLUXULUI DE EXECUȚIE: BUCLE CU FOR 121
10.1. Erori, semnale și capcane 129
10.2. Stare de ieșire 129
10.3. Verificarea stării de ieșire 129

10.4. Funcții pentru ieșiri de eroare ... 131
10.5. Listele AND și OR ... 132
10.6. Îmbunătățirea funcției de ieșire cu un mesaj de eroare ... 134
10.7. Curățenie după noi înșine ... 136
10.8. Semnalul 9 ... 139
10.9. O funcție de curățare ... 139
10.10. Crearea de fișiere temporare sigure ... 142
11. ÎNVĂȚÂND GNUPLOT - INSTALAREA, CITIREA DIN FIȘIER ȘI SALVAREA FIȘIERELOR ... 143
11.1. Invocarea gnuplot și primele grafice ... 143
11.2. Reprezentarea datelor dintr-un fișier ... 145
11.3. Abrevieri și valori implicite ... 148
11.4. Salvarea comenzilor și exportarea graficelor ... 149
11.5. Gestionarea opțiunilor cu set și show ... 152
12. ÎNVĂȚÂND GNUPLOT - VARIABILE, FUNCȚII DEFINITE DE UTILIZATOR ȘI TUTORIALE ... 153
12.1 Variabile și funcții definite de utilizator ... 153
12.2. Exerciții Gnuplot ... 154
12.2.1. Aproximarea polinomială a sinusului ... 154
12.2.2. Suprafață 3D dintr-o matrice de valori Z ... 154
12.2.3. Grafice unghiulare ... 155
12.2.4. Valoare independentă mapată în culori pe suprafața 3D ... 156
12.2.5. Culoare și orientare variabile în stilul de plot „with labels” ... 157
12.2.6. Grafice multiple ... 158
12.2.7. Sisteme de ordinul al 2-lea ... 159
12.2.8. Caracteristica pereți de rețea ... 162
12.2.9. Suprafață în 2 culori (Hidden3d/pm3d) ... 163
12.2.10. Diagrama Gantt simplă ... 164
12.2.11. Modul text îmbunătățit folosind un singur font codificat UTF-8 ... 165
BIBLIOGRAFIE ... 167

PREFAȚĂ

Un sistem de operare (SO) este o interfață între utilizator și hardware-ul computerului. Un sistem de operare este un software care îndeplinește toate sarcinile de bază, cum ar fi gestionarea fișierelor, gestionarea memoriei, gestionarea proceselor, gestionarea intrărilor și ieșirii și controlul dispozitivelor periferice, cum ar fi unitățile de disc și imprimantele. Fără un sistem de operare, un computer este inutil.

Acest îndrumător de laborator a fost elaboarat în acord cu conţinutul programelor de învăţământ atât de la Facultatea de Automatică și Calculatoare din Universitatea Politehnica Timişoara (UPT), cât și în conformitate cu programa Facultății de Fizică a Universității de Vest din Timișoara (UVT). Materialul didactic prezentat este adresat viitorilor ingineri ai Universității Politehnica din Timisoara, cât și studenților Facultății de Fizică ai Universității de Vest.

În acest ghid de laborator, autorii se străduiesc să-i obișnuiască pe studenți să scrie cod care interacționează cu un sistem de operare și să utilizeze mai bine resursele unui computer. Pentru a le permite studenților noștri să stăpânească principiile fundamentale ale științei calculatorului și să dezvolte în ei abilitățile necesare pentru a rezolva probleme practice folosind tehnologii și practici contemporane bazate pe utilizarea unui computer, pentru a cultiva o comunitate de profesioniști care vor servi publicului ca resurse de ultimă generație în domeniul științei calculatorului, știința și tehnologia informației.

Primele două laboratoare se concentrează pe realizarea unui studiu de caz prin instalarea și explorarea diferitelor tipuri de sisteme de operare pe o mașină fizică sau logică (virtuală).

Laboratoarele trei, patru și cinci se concentrează pe lucrul cu interfața liniei de comandă într-un mediu UNIX. Majoritatea utilizatorilor de computere de astăzi sunt familiarizaţi doar cu interfața grafică și au fost învățați de către vânzători și experți că interfața liniei de comandă este un lucru terifiant al trecutului. Acest lucru este regretabil, deoarece o interfață bună de linie de comandă este un mod minunat de expresiv de a comunica cu un computer, în același mod în care este cuvântul scris pentru ființele umane. S-a spus că „interfețele grafice facilitează sarcinile ușoare, în timp ce interfețele cu linia de comandă fac posibile sarcini dificile” și acest lucru este și astăzi foarte adevărat.

Laboratoarele șase până la zece introduc programarea shell, o tehnică desigur rudimentară, dar ușor de învățat, pentru automatizarea multor sarcini de calcul obișnuite. Shell-ul este un program care preia comenzi de la tastatură și le transmite sistemului de operare pentru a le executa. Învățând programarea shell, studenții se vor familiariza cu conceptele care pot fi aplicate la multe alte limbaje de programare. În cei mai simpli termeni, un script shell este un fișier care conține o serie de comenzi. Shell-ul citește acest fișier și execută comenzile ca și cum ar fi fost introduse direct pe linia de comandă.

Laboratoarele unsprezece și doisprezece se concentrează pe învățarea Gnuplot. Vizualizarea datelor este extrem de importantă pentru a comunica rezultatele cercetării dumneavoastră, fie într-un jurnal, fie către publicul larg, și pentru a analiza și a afla mai multe

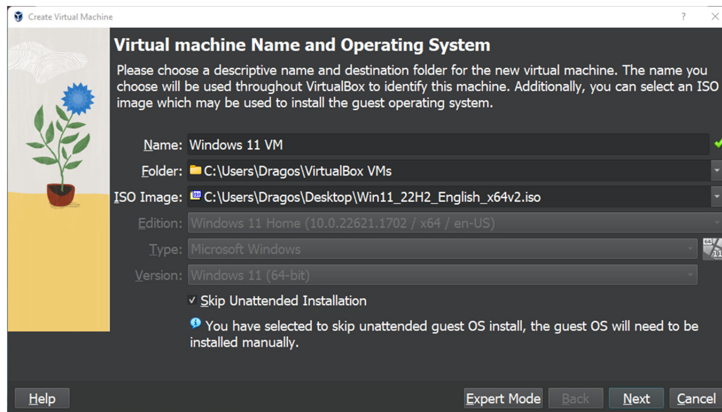
despre caracteristicile datelor (așa-numita „analiză exploratorie a datelor"). Unul dintre cele mai fundamentale instrumente în vizualizarea datelor este diagrama bidimensională (sau graficul). Gnuplot nu este nicidecum cel mai bun sau singurul (dacă trebuie să faceți post-procesare complicată înainte de a face graficul, atunci s-ar putea să fiți mai bine serviți folosind Python), dar este o alegere bună pentru o mulțime de sarcini de lucru, deoarece este suficient de simplu pentru a explora rapid mai multe opțiuni de vizualizare, fiind în același timp capabil să realizeze grafice de calitate odată ce ați decis ce doriți.

Studenților care participă la acest laborator li se recomandă să citească în prealabil laboratorul care urmează să fie întreprins pentru a se obișnui cu conceptele prezentate și pentru a utiliza mai bine timpul disponibil pentru finalizarea respectivului laborator.

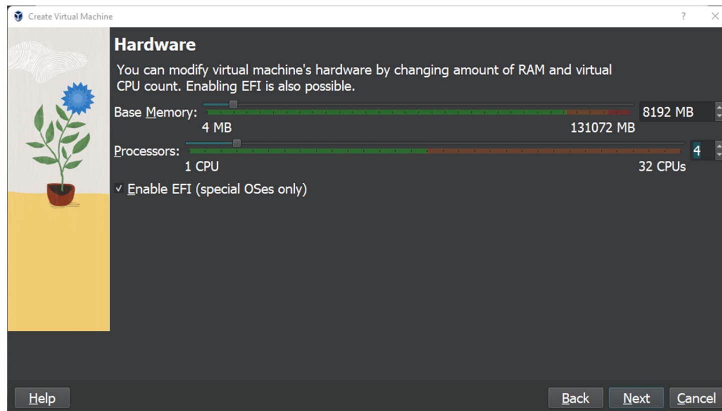
Autorii adresează mulțumiri celor doi referenți și tuturor colegilor care au contribuit la realizarea acestui îndrumător de laborator prin discuţii, sugestii şi sfaturi utile.

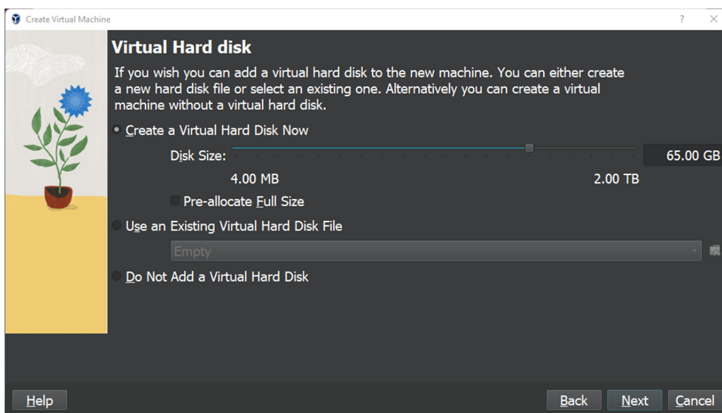
1. INSTALAREA SISTEMULUI DE OPERARE WINDOWS 11 ÎN ORACLE VM VIRTUALBOX

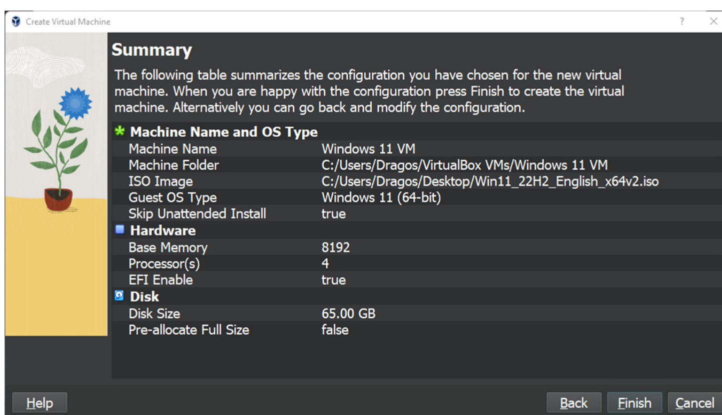
1: Crearea unei mașini virtuale noi (vom bifa opțiunea “skip unattended installation” pentru a observa toți pașii necesari instalării sistemului de operare).



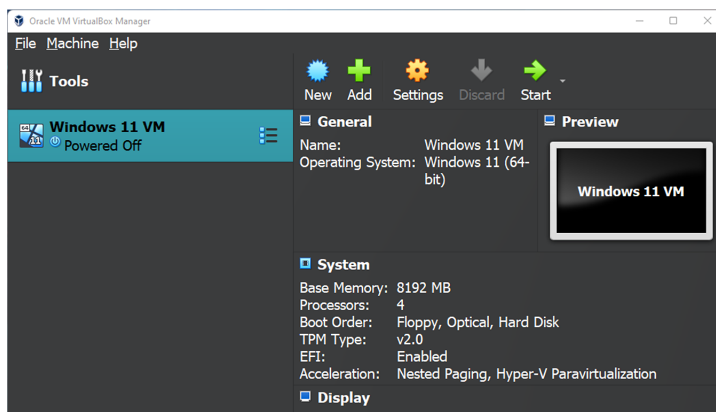
2: Atribuirea de resurse hardware pentru mașina virtuală (RAM, CPU și spațiu de stocare stocare).





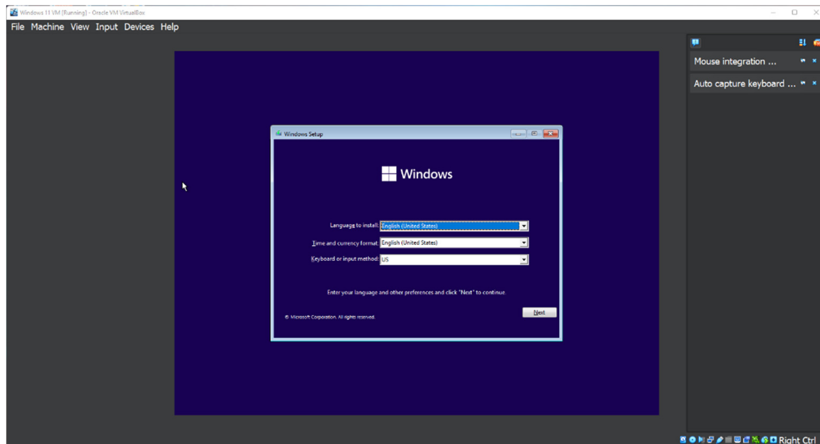


3: După setarea resurselor ce urmează a fi disponibile în noua mașină virtuală creată, putem porni mașina virtuală. Deoarece am bifat opțiunea “skip unattended installation”, la prima pornire a noii mașini virtuale, vom trece prin procesul de instalare a sistemului de operare.

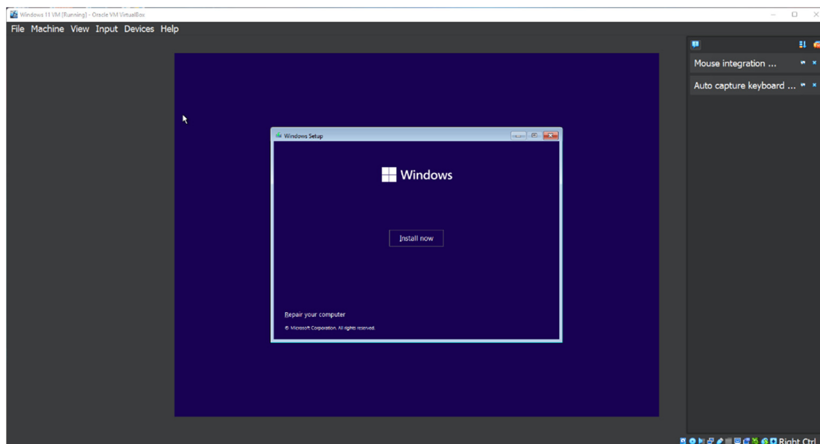


4: Odată ce sistemul bootează din imaginea aleasă, vom trece prin pașii propriu-ziși de instalare Windows 11.

- Alegerea limbii de instalare şi a formatelor pentru dată, monedă şi tastatură



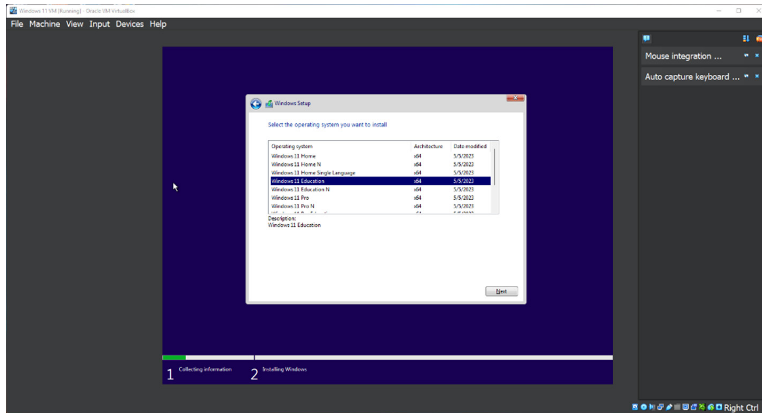
- Meniul de instalare (în cazul în care avem instalat deja un sistem de operare care a devenit corupt, din acest meniu putem alege opțiunea de “repair”)



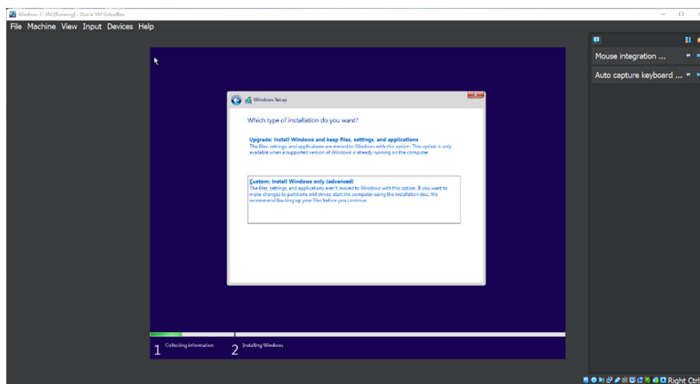
- Alegerea variantei dorite a sistemului de operare. În tabelul următor avem lista de caracteristici pentru fiecare variantă.

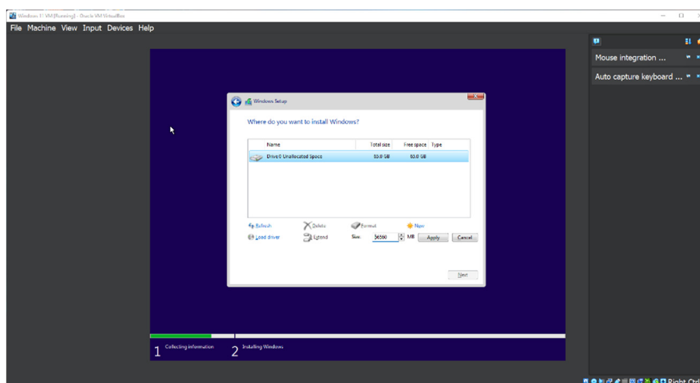
Features	Home	Pro	Pro for Workstations	Enterprise	Education	SE
Supported RAM (Maximum)	4GB (32bit) 128GB (64bit)	4GB (32bit) 2TB (64bit)	4GB (32bit) 6TB(64bit)	4GB (32bit) 6TB (64bit)	4GB (32bit) 2TB(64bit)	
Microsoft Account required for initial setup	✓					
Windows 11 in S Mode option	✓					
Cloud-managed only						✓
Only IT admins can install apps						✓
OneDrive cloud storage required						✓
Microsoft Office pre-installed						✓
Only available pre-installed (no image available)						✓
Runs all Windows apps (UWP, PWA, and Win32)	✓	✓	✓	✓	✓	
Microsoft Store	✓	✓	✓	✓	✓	
Optimized for low-memory & small-screen devices						✓
Remote Desktop		✓	✓	✓	✓	
Hyper-V virtualization		✓	✓	✓	✓	
Windows Sandbox		✓	✓	✓	✓	
Resilient File System (ReFS)			✓	✓	✓	
Bitlocker Device Encryption		✓	✓	✓	✓	
Device Encryption	✓	✓	✓	✓	✓	
Find My Device	✓	✓	✓	✓	✓	

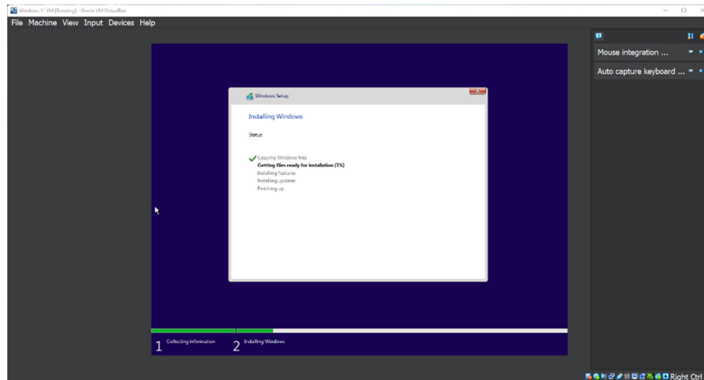
Firewall and Network Protection	✓	✓	✓	✓	✓
Internet Protection	✓	✓	✓	✓	✓
Parental Controls and Protection	✓	✓	✓	✓	✓
Secure Boot	✓	✓	✓	✓	✓
Windows Hello	✓	✓	✓	✓	✓
Windows Information Protection (WIP)		✓	✓	✓	✓
Windows Security	✓	✓	✓	✓	✓
Assigned Access		✓	✓	✓	✓
Dynamic Provisioning		✓	✓	✓	✓
Enterprise State Roaming with Azure		✓	✓	✓	✓
Group Policy		✓	✓	✓	✓
Kiosk Mode Setup		✓	✓	✓	✓
Microsoft Store for Business		✓	✓	✓	✓
Mobile Device Management		✓	✓	✓	✓
Support for Active Directory		✓	✓	✓	✓
Support for Azure Active Directory		✓	✓	✓	✓
Windows Update for Business		✓	✓	✓	✓
AppLocker		✓	✓	✓	✓
Persistent Memory			✓	✓	✓
SMB Direct			✓	✓	✓
Servicing Timeline	24 months from release date	24 months from release date	24 months from release date	36 months from release date	36 months from release date



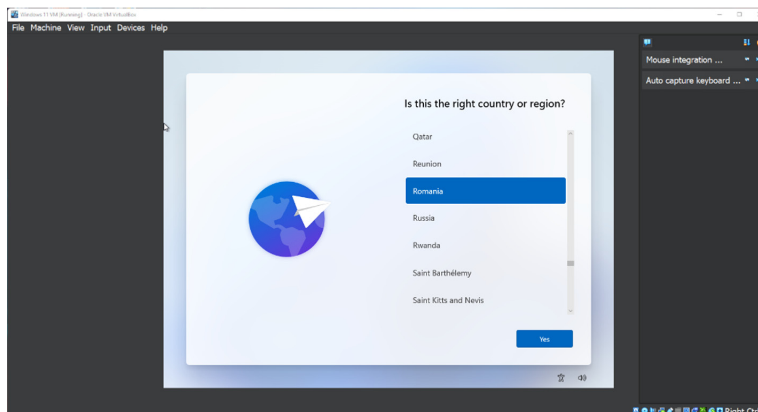
- În continuare, alegem tipul de instalare dorit (dacă avem deja o versiune instalată și dorim să facem upgrade sau vrem o instalare nouă; în cazul nostru, dorim o instalare curată).

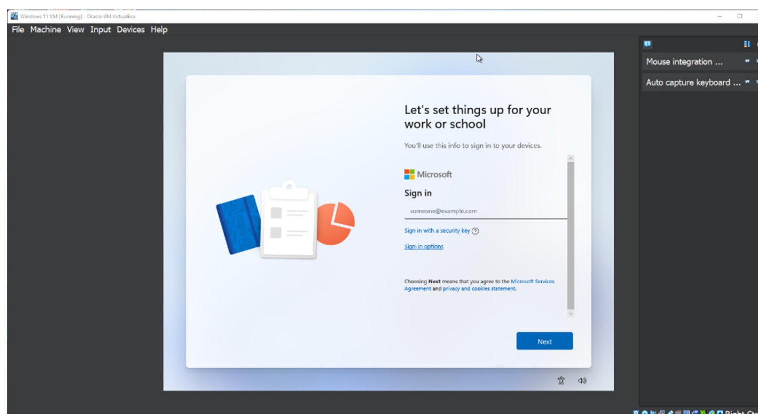


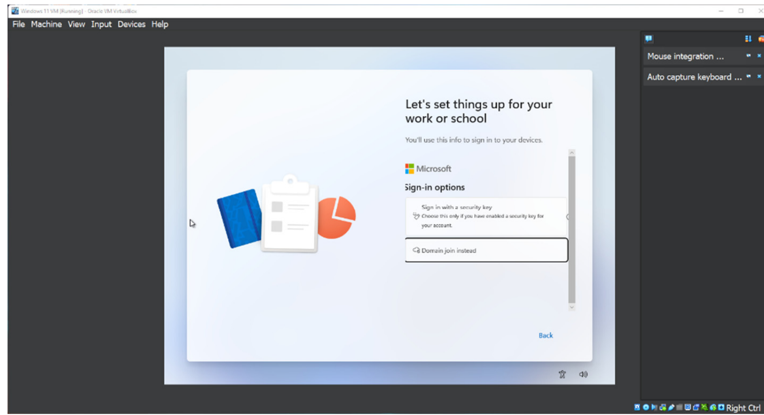


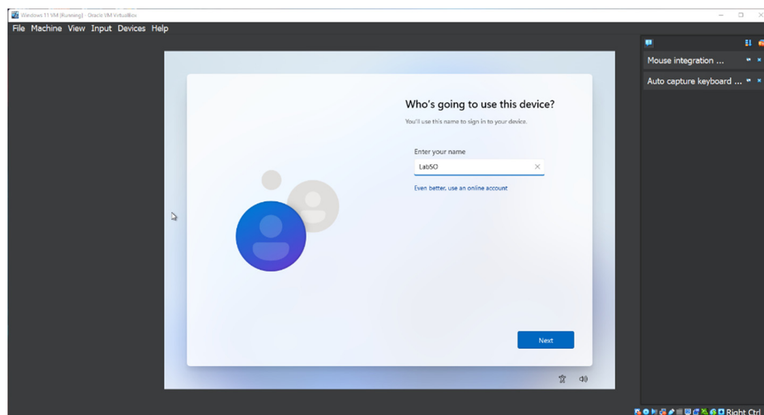


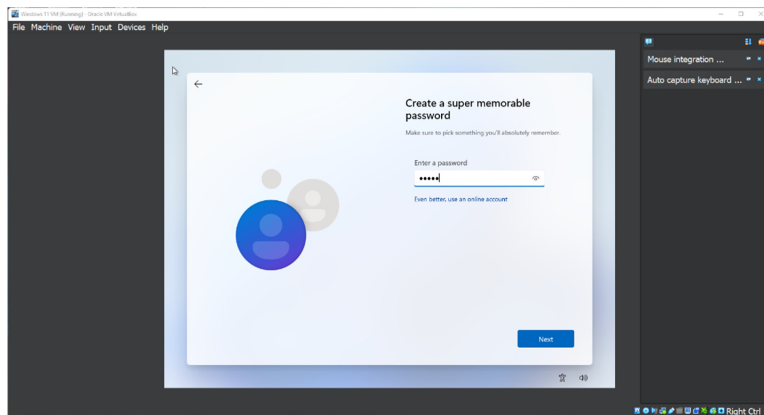
- Ultimii pași în procesul de instalare vor fi pentru setarea regiunii și datelor utilizatorului. Vom alege să creem un cont local (urmând pașii de mai jos) urmat de setările de securitate: parola și întrebările de siguranță (pentru folosire în cazul în care este necesară resetarea parolei fără a avea parola)

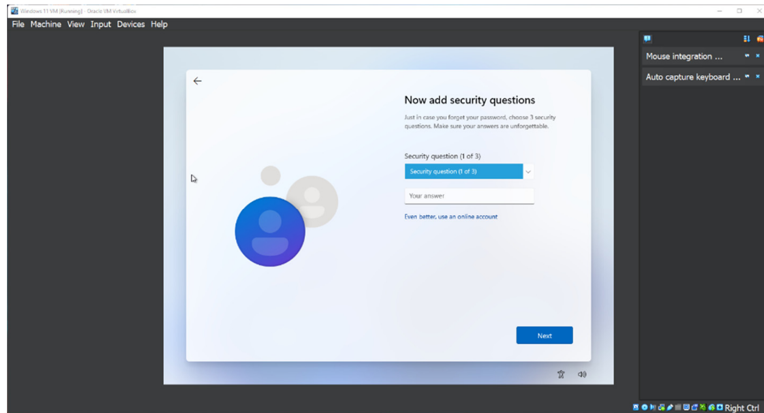




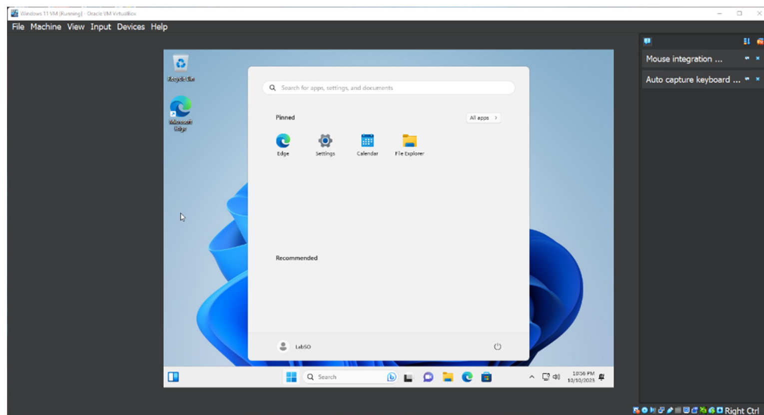

File Machine View Input Devices Help
Mouse integration ...
Auto capture keyboard ...
Let's set things up for your work or school
You'll use this info to sign in to your devices.
Microsoft
Sign-in options
Domain join instead
Back
Right Ctrl


File Machine View Input Devices Help
Mouse integration ...
Auto capture keyboard ...
Who's going to use this device?
You'll use this name to sign in to your device.
Enter your name
LabSO
Even better, use an online account
Next
Right Ctrl


File Machine View Input Devices Help
Mouse integration ...
Auto capture keyboard ...
Create a super memorable password
Make sure to pick something you'll absolutely remember.
Enter a password
Even better, use an online account
Next
Right Ctrl

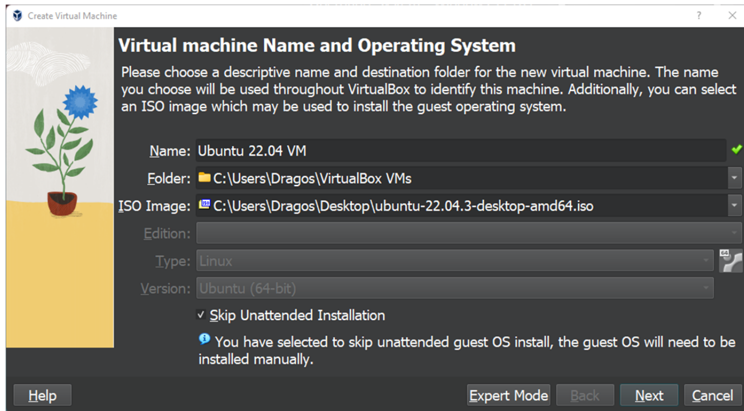


- Dacă toți pașii au fost urmați, ne vom găsi în sistemul de operare nou instalat, precum în imaginea următoare:

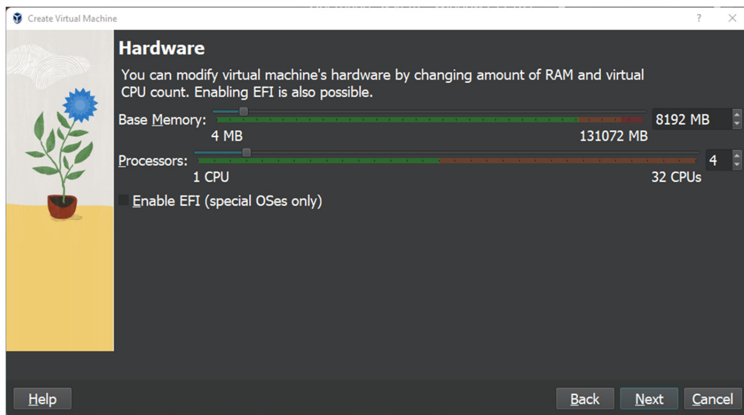


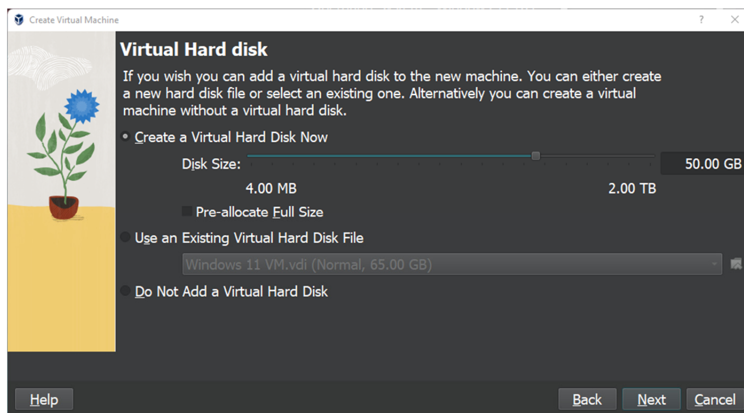
2. INSTALAREA SISTEMULUI DE OPERARE UBUNTU 22 ÎN ORACLE VM VIRTUALBOX

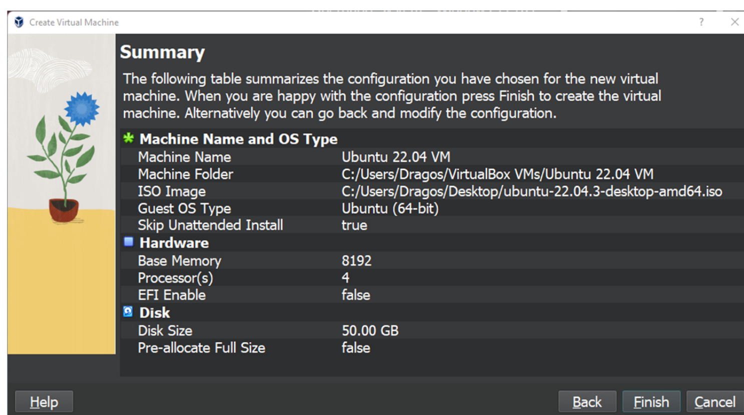
1: Crearea unei mașini virtuale noi (vom bifa optiunea “skip unattended installation” pentru a observa toți pașii necesari instălarii sistemului de operare).



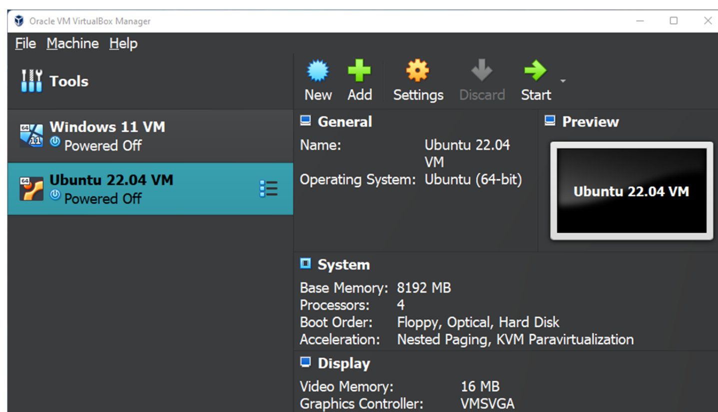
2: Atribuirea de resurse hardware pentru mașina virtuală (RAM, CPU și spațiu de stocare).





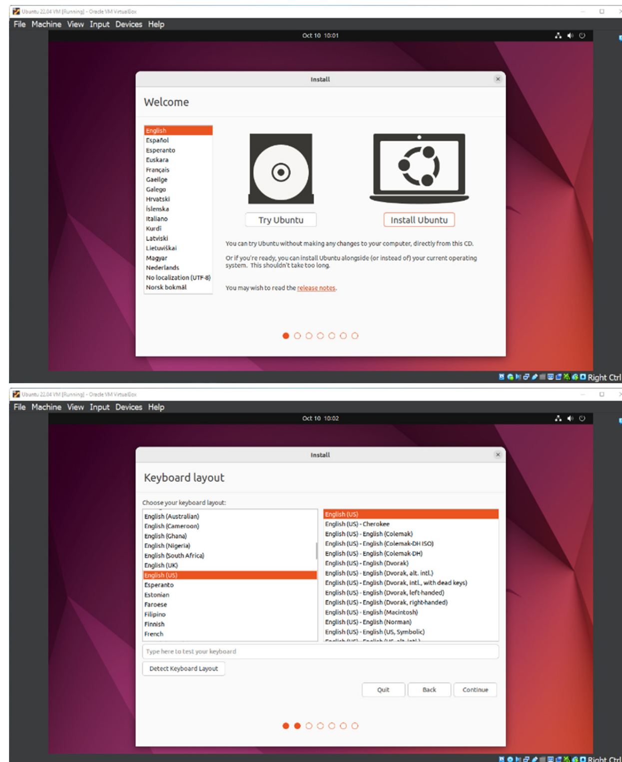


3: După setarea resurselor ce urmează a fi disponibile în noua mașină virtuală creată, putem porni mașina virtuală. Deoarece am bifat opțiunea “skip unattended installation”, la prima pornire a noii mașini virtuale, vom trece prin procesul de instalare a sistemului de operare.

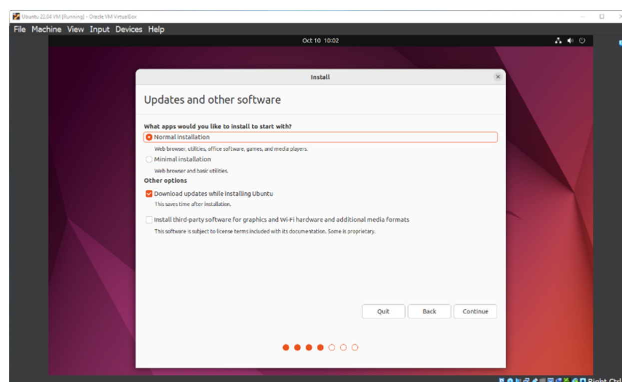


4: Odată ce sistemul bootează din imaginea aleasă, vom trece prin pașii propriu-ziși de instalare Ubuntu 22.04.

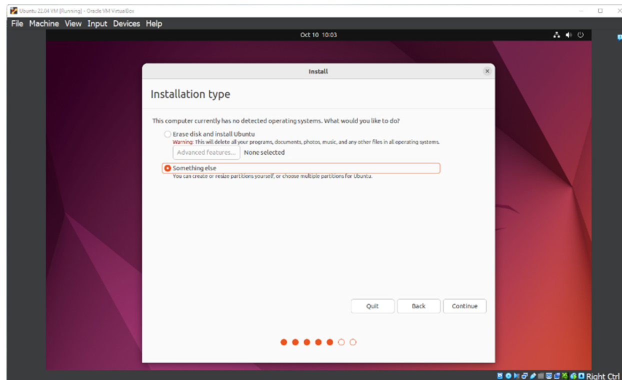
- Alegerea limbii de instalare și a formatelor pentru dată, monedă și tastatură

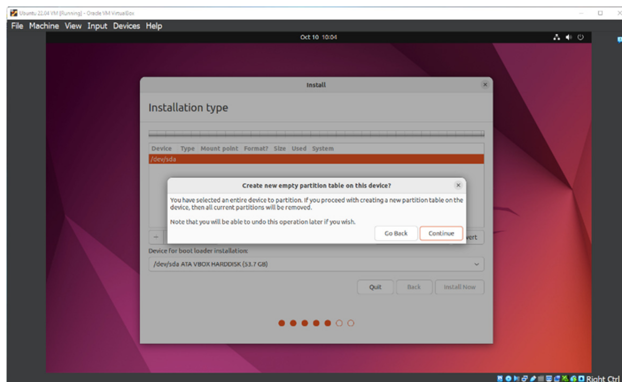


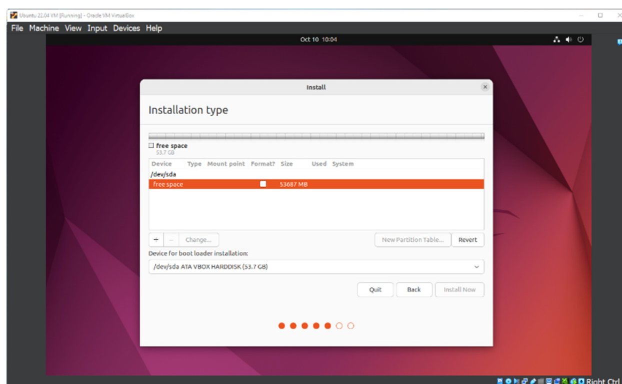
- Meniul de instalare (opțiuni de instalare: normală sau minimală, cu sau fără access internet și instalarea de software terța-parte în cazuri specifice)



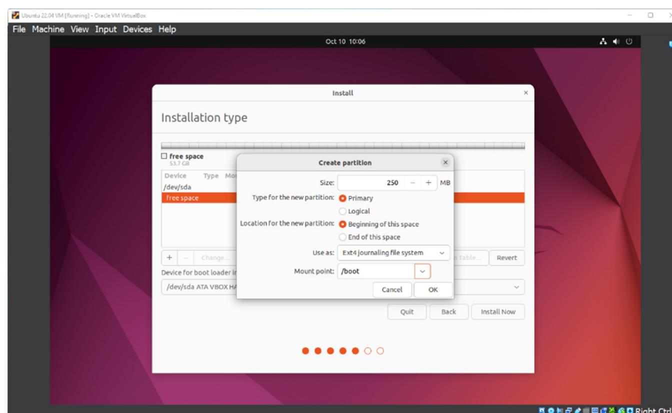
- În continuare, alegem tipul de instalare dorit ("something else" ne permite să creem un nou tablou de partiție pe dispozitiv, și să definim dimensiunile diferitelor sisteme de fișiere specifice Ubuntu)

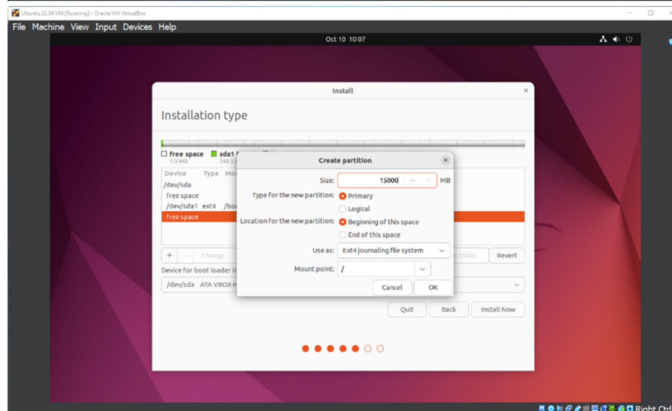


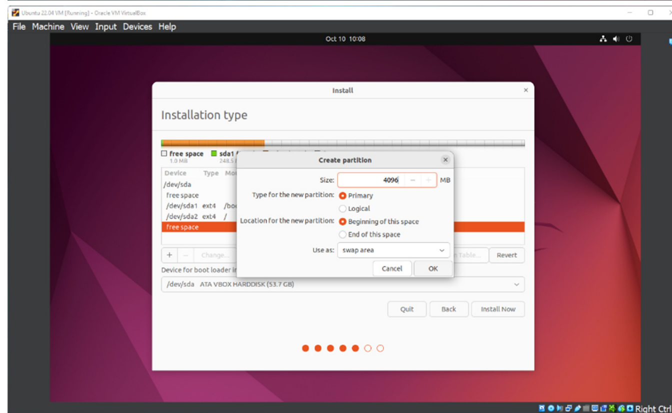


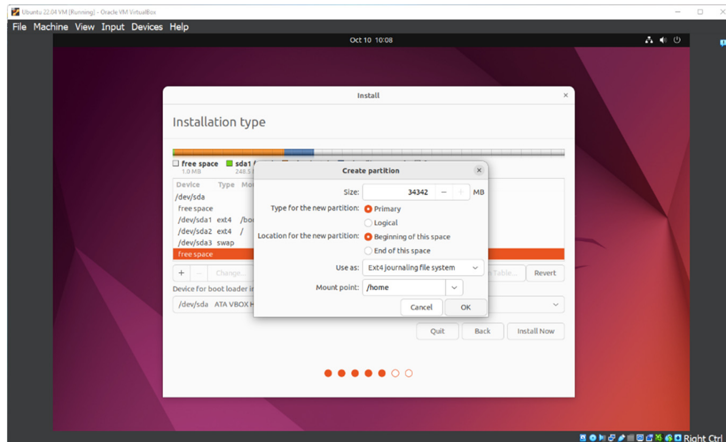


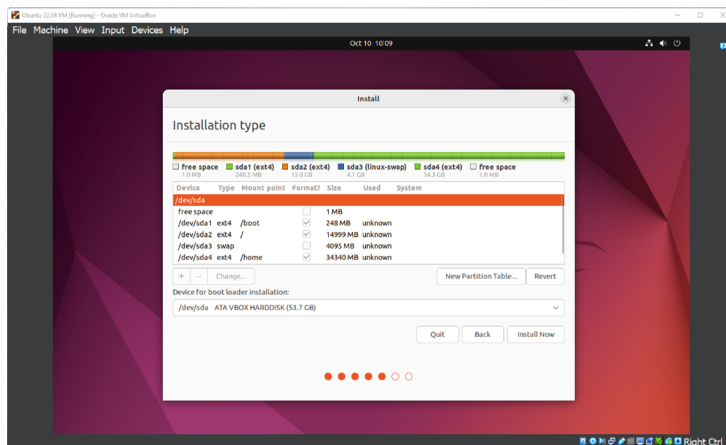
- Urmează să împărțim spațiul de stocare. Vom creea o partiție /boot de 250 MB, o partiție /root (sau doar /) de 15000 MB, un spațiu swap de 4096 MB și restul spațiul îl vom aloca partiției /home. Crearea partițiilor /boot și /root separat de partiția /home este o procedură care ne permite să facem upgrade sistemului fără a fi nevoie să ne atingem de datele din partiția /home, sau în cazul unor erori, să putem recupera ușor datele din /home.

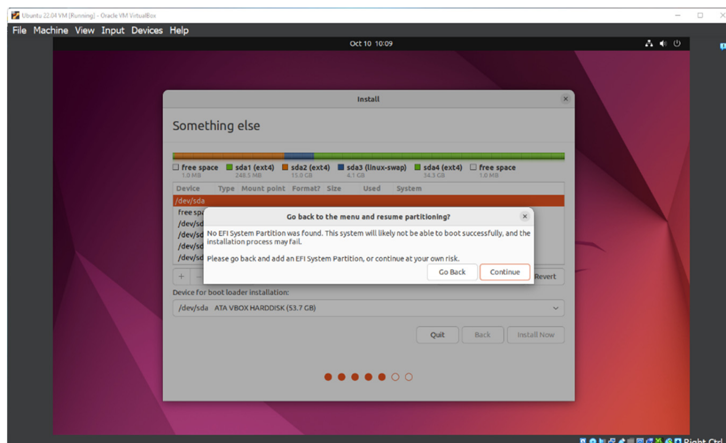

File Machine View Input Devices Help
Oct 10 10:06
Install
Installation type
free space
Device Type
/dev/sda
free space
Create partition
Size: 250 MB
Type for the new partition: Primary
Logical
Location for the new partition: Beginning of this space
End of this space
Use as: Ext4 journaling file system
Mount point: /boot
Cancel OK
Revert
Device for boot loader
Quit Back Install Now
Right Ctrl


File Machine View Input Devices Help
Oct 10 10:07
Install
Installation type
free space sda1
Device Type
/dev/sda
free space
/dev/sda1 ext4
free space
Create partition
Size: 15000 MB
Type for the new partition: Primary
Logical
Location for the new partition: Beginning of this space
End of this space
Use as: Ext4 journaling file system
Mount point: /
Cancel OK
Revert
Device for boot loader
Quit Back Install Now
Right Ctrl

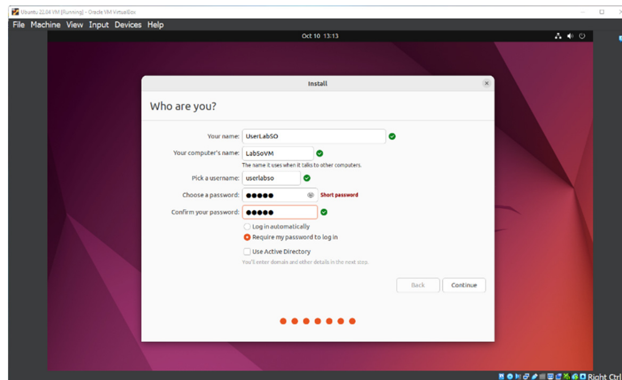

File Machine View Input Devices Help
Oct 10 10:08
Install
Installation type
free space sda1
Device Type
/dev/sda
free space
/dev/sda1 ext4
/dev/sda2 ext4 /
free space
Create partition
Size: 4096 MB
Type for the new partition: Primary
Logical
Location for the new partition: Beginning of this space
End of this space
Use as: swap area
Cancel OK
Revert
Device for boot loader
/dev/sda ATA VBOX HARDDISK (53.7 GB)
Quit Back Install Now
Right Ctrl

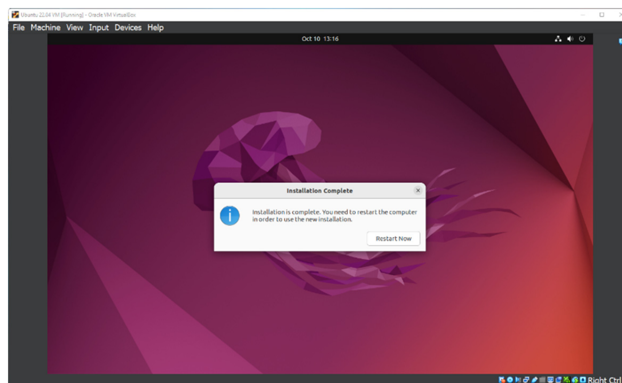

File Machine View Input Devices Help
Oct 10 10:08
Install
Installation type
Create partition
Size: 34342 MB
Type for the new partition: Primary
Logical
Location for the new partition: Beginning of this space
End of this space
Use as: Ext4 journaling file system
Mount point: /home
Cancel
OK
Revert
Quit
Back
Install Now
Right Ctrl


File Machine View Input Devices Help
Oct 10 10:09
Install
Installation type
free space sda1 (ext4) sda2 (ext4) sda3 (linux-swap) sda4 (ext4) free space
Device Type Mount point Format? Size Used System
/dev/sda
free space 1 MB
/dev/sda1 ext4 /boot 248 MB unknown
/dev/sda2 ext4 / 14999 MB unknown
/dev/sda3 swap 4095 MB unknown
/dev/sda4 ext4 /home 34340 MB unknown
Change...
New Partition Table...
Revert
Device for boot loader installation:
/dev/sda ATA VBOX HARDDISK (53.7 GB)
Quit
Back
Install Now
Right Ctrl

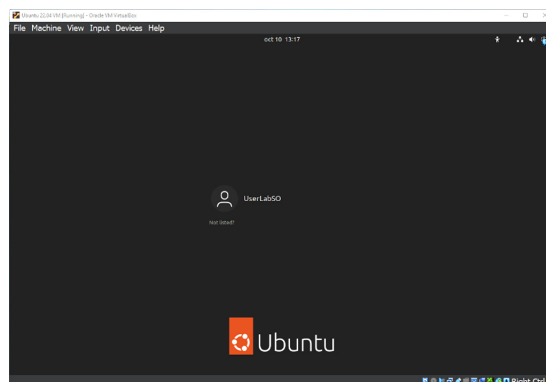

File Machine View Input Devices Help
Oct 10 10:09
Install
Something else
free space sda1 (ext4) sda2 (ext4) sda3 (linux-swap) sda4 (ext4) free space
Device Type Mount point Format? Size Used System
Go back to the menu and resume partitioning?
No EFI System Partition was found. This system will likely not be able to boot successfully, and the installation process may fail.
Please go back and add an EFI System Partition, or continue at your own risk.
Go Back
Continue
Device for boot loader installation:
/dev/sda ATA VBOX HARDDISK (53.7 GB)
Quit
Back
Install Now
Right Ctrl

- Ultimii pași în procesul de instalare vor fi pentru setarea regiunii și datelor utilizatorului. Vom alege să creem un cont local (urmând pașii de mai jos) urmat de setările de securitate.





- Dacă toți pașii au fost urmați, ne vom găsi în sistemul de operare nou instalat, precum în imaginea următoare:

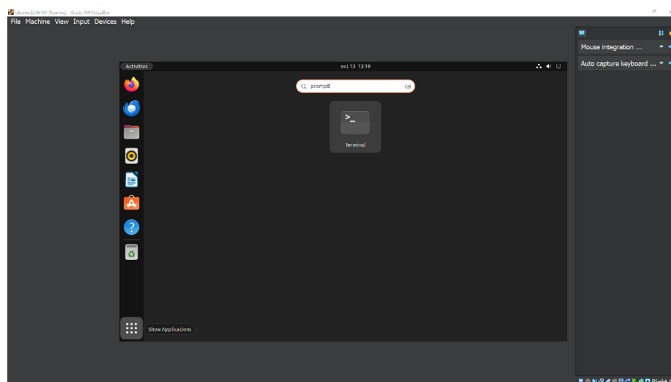


3. COMENZI UTILE ÎN SISTEME LINUX – GESTIONAREA FOLDERELOR

Linia de comandă Linux este o interfață text către computer. Deseori denumit shell, terminal, consolă, prompt sau diverse alte nume, poate da aspectul de a fi complex și greu de utilizat. Cu toate acestea, capacitatea de a copia și lipi comenzi de pe un site web, combinată cu puterea și flexibilitatea oferite de linia de comandă, înseamnă că utilizarea acesteia poate fi esențială atunci când încercați să urmați instrucțiuni de pe un site web.

Acest laborator vă va învăța puțin din istoria liniei de comandă, apoi vă va ghida prin câteva exerciții practice pentru a vă familiariza cu câteva comenzi și concepte de bază. Vom presupune că nu există cunoștințe anterioare, dar până la sfârșit sperăm că vă veți simți ceva mai confortabil data viitoare când vă confruntați cu câteva instrucțiuni care încep cu „Deschideți un terminal".

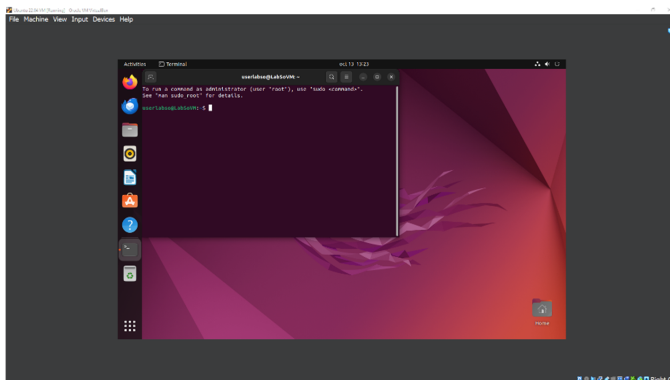
Pe un sistem Ubuntu 22.04 puteți găsi un lansator pentru terminal făcând clic pe elementul Activități din partea stângă jos a ecranului, apoi tastând primele litere de „terminal", „comandă", „prompt" sau „shell". Da, dezvoltatorii au configurat lansatorul cu toate cele mai comune sinonime, așa că nu ar trebui să aveți probleme în a-l găsi.



Alte versiuni de Linux sau Ubuntu, vor avea de obicei un lansator de terminal situat în același loc cu celelalte lansatoare de aplicații. S-ar putea să fie ascuns într-un submeniu sau ar putea fi necesar să-l căutați din lansatorul dvs., dar este cu siguranta acolo undeva.

Dacă nu puteți găsi un lansator sau dacă doriți doar o modalitate mai rapidă de a afișa terminalul, majoritatea sistemelor Linux folosesc aceeași comandă rapidă implicită de la tastatură pentru a-l porni: Ctrl-Alt-T.

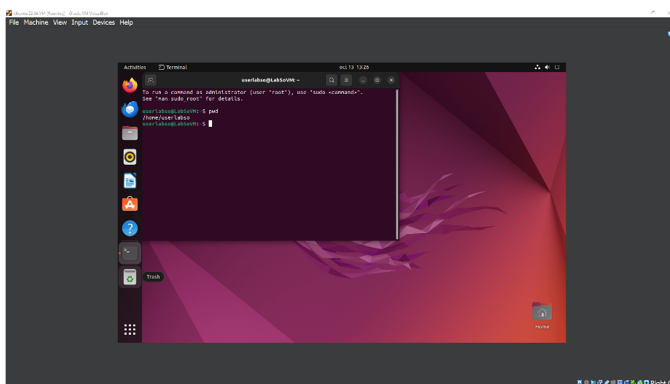
Oricum ai lansa terminalul, ar trebui să ajungi cu o fereastră destul de plictisitoare, cu un pic de text ciudat în partea de sus, la fel ca imaginea de mai jos. În funcție de sistemul Linux, culorile pot să nu fie aceleași, iar textul va spune probabil ceva diferit, dar aspectul general al unei ferestre cu o zonă de text mare (în mare parte goală) ar trebui să fie similară.



Să rulăm prima noastră comandă. Faceți clic cu mouse-ul în fereastră pentru a vă asigura că acolo vor merge apăsările de taste, apoi tastați următoarea comandă, toate cu litere mici, înainte de a apăsa tasta Enter sau Return pentru a o executa.

```
pwd
```

Ar trebui să vedeți o cale de director tipărită (probabil ceva de genul /home/YOUR_USERNAME), apoi o altă copie a acelui fragment ciudat de text.



Există câteva elemente de bază de înțeles aici, înainte de a intra în detalii despre ceea ce a făcut de fapt comanda. În primul rând, atunci când tastați o comandă, aceasta apare pe aceeași linie cu textul ciudat. Acest text este acolo pentru a vă spune că computerul este gata să accepte o comandă. De fapt, este de obicei denumit prompt și uneori s-ar putea să vedeți instrucțiuni care spun „afișați un prompt”, „deschideți un prompt de comandă”, „la promptul bash” sau similar. Toate sunt doar moduri diferite de a vă cere să deschideți un terminal pentru a ajunge la un shell.

În ceea ce privește sinonimele, un alt mod de a privi promptul este de a spune că există o linie în terminal în care tastați comenzi - o linie de comandă. Din nou, dacă vedeți menționarea „liniei de comandă”, este doar un alt mod de a vorbi despre un shell care rulează într-un terminal.

Al doilea lucru de înțeles este că, atunci când rulați o comandă, orice ieșire pe care o produce va fi de obicei tipărită direct în terminal, apoi vi se va afișa un alt prompt odată ce a terminat. Unele comenzi pot scoate mult text, altele vor funcționa silențios și nu vor scoate absolut nimic. Nu vă alarmați dacă executați o comandă și apare imediat un alt prompt, deoarece asta înseamnă de obicei că comanda a reușit. Dacă te gândești la conexiunile lente de rețea ale terminalelor din anii '70, acei programatori timpurii au decis că, dacă totul merge bine, ar putea la fel de bine să salveze câțiva octeți prețioși de transfer de date fără a spune nimic.

ATENTIE!

Fiți deosebit de atenți la majuscule atunci când introduceți text în linia de comandă. Tastarea PWD în loc de pwd va produce o eroare, dar uneori, majusculele greșite pot avea ca rezultat o comandă care pare să ruleze, dar nu face ceea ce vă așteptați. Ne vom uita mai mult la un astfel de caz mai tarziu, dar, deocamdată, asigurați-vă că introduceți toate rândurile exact cum este afișat.

Sa revenim la comanda în sine. pwd este o abreviere pentru „print working directory”. Tot ce face este să imprime directorul de lucru curent al shell-ului. Dar ce este un director de lucru?

Un concept important de înțeles este că shell-ul are o noțiune de locație implicită în care vor avea loc orice operațiuni de fișier. Acesta este directorul său de lucru. Dacă încercați să creați fișiere sau directoare noi, să vedeți fișiere existente sau chiar să le ștergeți, shell-ul va presupune că le căutați în directorul de lucru curent, dacă nu luați măsuri pentru a specifica altfel. Deci, este destul de important să aveți o idee despre directorul în care se află shell-ul la un moment dat, la urma urmei, ștergerea fișierelor din directorul greșit ar putea fi dezastruoasă. Dacă aveți vreodată îndoieli, comanda pwd vă va spune exact care este directorul de lucru curent.

Puteți schimba directorul de lucru folosind comanda cd, o abreviere pentru „change directory”. Încercați să introduceți următoarele:

```
cd /
pwd
```

Acum directorul tău de lucru este „/”. Dacă veniți dintr-un fundal Windows, probabil că sunteți obișnuiti ca fiecare unitate de stocare să aibă propria sa literă, cea principala fiind de obicei „C:”. Sistemele Unix nu împart unitățile așa. În schimb, au un singur sistem de fișiere unificat, iar unitățile individuale pot fi atașate („mounted”) în orice locație din sistemul de fișiere care are cel mai mult sens. Directorul „/”, adesea denumit director rădăcină (root), este baza acelui sistem de fișiere unificat. De acolo totul se ramifică pentru a forma un arbore de directoare și subdirectoare.

Prea multe “root”

Atenție: deși directorul „/” este uneori denumit director root, cuvântul „root” are o altă semnificație. root este, de asemenea, numele care a fost folosit pentru superutilizator încă din primele zile ale Unix. Superutilizatorul, așa cum sugerează și numele, are mai multe puteri decât un utilizator obișnuit, așa că poate face ravagii cu ușurință cu o comandă scrisă prost.

Din directorul rădăcină, următoarea comandă vă va muta în directorul „home” (care este un subdirector imediat al „/”):

```
cd home
pwd
```

Pentru a merge la directorul părinte, în acest caz înapoi la „/”, utilizați sintaxa specială a două puncte (..) atunci când schimbați directorul (rețineți spațiul dintre cd și .., spre deosebire de DOS, nu puteți doar să tastați cd.. ca o singură comandă):

```
cd ..
pwd
```

Tastarea unui cd este o comandă rapidă pentru a reveni la directorul dvs. de acasă:

```
cd
pwd
```

De asemenea, puteți utiliza .. de mai multe ori dacă trebuie să treceți în sus prin mai multe niveluri ale directoarelor părinte:

```
cd ../..
pwd
```

Observați că în exemplul anterior am descris o rută de urmat prin directoare. Calea pe care am folosit-o înseamnă „începând din directorul de lucru, mutați la părinte / din noua locație mutați din nou în părinte”. Deci, dacă am dori să mergem direct din directorul nostru principal în directorul „etc” (care se află direct în rădăcina sistemului de fișiere), am putea folosi această abordare:

```
cd
pwd
cd ../../etc
pwd
```

Cele mai multe dintre exemplele pe care le-am analizat până acum folosesc căi relative. Adică, locul în care ajungi depinde de directorul tău de lucru actual. Hai să încercați să faceți cd în folderul „etc”. Dacă vă aflați deja în directorul rădăcină, va funcționa:

```
cd /
pwd
cd etc
pwd
```

Dar dacă ești în directorul tău de pornire?

```
cd
pwd
```

```
cd etc
pwd
```

Veți vedea o eroare care spune „Nu există un astfel de fișier sau director” înainte de a rula ultimul pwd. Schimbarea directorului prin specificarea numelui directorului sau folosind .. va avea efecte diferite în funcție de unde porniți. Calea are sens doar în raport cu directorul de lucru.

Dar am văzut două comenzi care sunt absolute. Indiferent care este directorul de lucru curent, ele vor avea același efect. Prima este atunci când rulați cd pe cont propriu pentru a merge direct în directorul dvs. de acasă. Al doilea este atunci când ați folosit cd / pentru a comuta la directorul rădăcină. De fapt, orice cale care începe cu o bară oblică este o cale absolută. Vă puteți gândi la asta ca spunând „comutați la directorul rădăcină, apoi urmați traseul de acolo”. Asta ne oferă o modalitate mult mai ușoară de a comuta la directorul etc, indiferent unde ne aflăm în prezent în sistemul de fișiere:

```
cd
pwd
cd /etc
pwd
```

De asemenea, ne oferă o altă modalitate de a reveni la directorul home și chiar la folderele din acesta. Să presupunem că doriți să mergeți direct în folderul „Desktop” de oriunde de pe disc (rețineți majuscula „D”). În următoarea comandă, va trebui să înlocuiți USERNAME cu propriul nume de utilizator, comanda whoami vă va reaminti numele de utilizator, în cazul în care nu sunteți sigur:

```
whoami
cd /home/USERNAME/Desktop
pwd
```

Există o altă comandă rapidă la îndemână care funcționează ca o cale absolută. După cum ați văzut, folosirea „/” la începutul căii înseamnă „pornirea din directorul rădăcină”. Folosirea caracterului tilda ("~") la începutul căii înseamnă în mod similar „pornirea din directorul meu home”.

```
cd ~
pwd
cd ~/Desktop
pwd
```

Acum, acel text ciudat din prompt ar putea avea un pic de sens. Ați observat că se schimbă pe măsură ce vă deplasați prin sistemul de fișiere? Pe un sistem Ubuntu, arată numele dvs. de utilizator, numele rețelei computerului dvs. și directorul de lucru curent. Dar dacă vă aflați undeva în directorul home, va folosi „~” ca abreviere. Să ne plimbăm puțin prin sistemul de fișiere și să urmărim promptul în timp ce faceți acest lucru:

```
cd
cd /
cd ~/Desktop
cd /etc
cd /var/log
cd ..
cd
```

Trebuie să vă plictisiți doar să vă mutați prin sistemul de fișiere până acum, dar o bună înțelegere a căilor absolute și relative va fi neprețuită pe măsură ce trecem la crearea unor foldere și fișiere noi!

3.1. Crearea de foldere și fișiere

În această secțiune vom crea câteva fișiere reale cu care să lucrăm. Pentru a evita călcarea accidentală peste fișiere reale, vom începe prin a crea un director nou, departe de folderul home, care va servi ca un mediu mai sigur în care să experimentam:

```
mkdir /tmp/tutorial
cd /tmp/tutorial
```

Observați ca utilizam o clee absoluta pentru a ne asigura ca directorul tutorial este creat in /tmp. Fără bara oblică de la început, comanda mkdir ar încerca să găsească un director tmp în directorul de lucru curent, apoi ar încerca să creeze un director tutorial în interiorul acestuia. Dacă nu poate găsi un director tmp, comanda ar eșua.

În cazul în care nu ați ghicit, mkdir este prescurtarea pentru „make directory”. Acum că ne aflăm în siguranță în zona noastră de testare (verificați cu pwd dacă nu sunteți sigur), să creăm câteva subdirectoare:

```
mkdir dir1 dir2 dir3
```

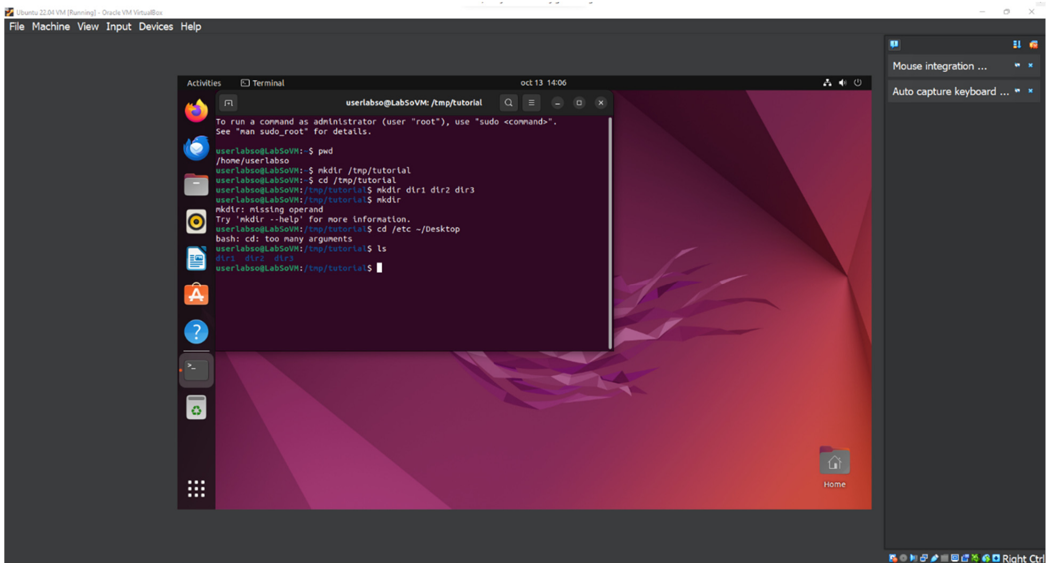
Există ceva puțin diferit la aceasta comandă. Până acum am văzut doar comenzi care funcționează singure (cd, pwd) sau care au un singur element ulterior (cd /, cd ~/Desktop). Dar de data aceasta am adăugat trei lucruri după comanda mkdir. Aceste lucruri sunt denumite parametri sau argumente, iar comenzi diferite pot accepta un număr diferit de argumente. Comanda mkdir așteaptă cel puțin un argument, în timp ce comanda cd poate funcționa cu zero sau unul, dar nu mai mult. Vedeți ce se întâmplă când încercați să transmiteți un număr greșit de parametri unei comenzi:

```
mkdir
cd /etc ~/Desktop
```

Înapoi la noile noastre directoare. Comenzile de mai sus vor fi creat trei subdirectoare noi în folderul nostru. Putem sa le vedem cu comanda ls (list):

```
ls
```

Dacă ați urmat ultimele comenzi, terminalul dvs. ar trebui să arate cam așa:



Observați că mkdir a creat toate folderele într-un singur director. Nu a creat dir3 în dir2 în interiorul dir1 sau orice altă structură nested. Dar uneori este util să poți face exact asta, iar mkdir are o modalitate:

```
mkdir -p dir4/dir5/dir6
ls
```

De data aceasta, veți vedea că numai dir4 a fost adăugat la listă, deoarece dir5 se află în el, iar dir6 este în interiorul acesteia. Mai târziu vom instala un instrument util pentru a vizualiza structura, dar aveți deja suficiente cunoștințe pentru a o confirma:

```
cd dir4
ls
cd dir5
ls
cd ../..
```

„-p” pe care l-am folosit se numește o opțiune sau un comutator (în acest caz înseamnă „creați și directoarele părinte”). Opțiunile sunt folosite pentru a modifica modul în care funcționează o comandă, permițând unei singure comenzi să se comporte într-o varietate de moduri diferite. Din păcate, din cauza ciudățeniei istoriei și a naturii umane, opțiunile pot lua forme diferite în diferite comenzi. Le veți vedea adesea ca caractere unice precedate de o cratimă (ca în acest caz) sau cuvinte mai lungi precedate de două cratime. Forma cu un singur caracter permite combinarea mai multor opțiuni, deși nu toate comenzile vor accepta acest lucru. Și pentru a încurca și mai mult lucrurile, unele comenzi nu își identifică deloc în mod clar opțiunile, dacă ceva este sau nu o opțiune este dictat doar de ordinea argumentelor! Nu

trebuie să vă faceți griji cu privire la toate posibilitățile, trebuie doar să știți că opțiunile există și pot lua mai multe forme diferite. De exemplu, toate acestea înseamnă exact același lucru:

Nu introduceți acestea, sunt aici doar în scopuri demonstrative

```
mkdir --parents --verbose dir4/dir5
mkdir -p --verbose dir4/dir5
mkdir -p -v dir4/dir5
mkdir -pv dir4/dir5
```

Acum știm cum să creăm mai multe directoare doar prin trecerea lor ca argumente separate la comanda mkdir. Dar să presupunem că vrem să creăm un director cu un spațiu în nume? Hai să încercăm:

```
mkdir alt folder
ls
```

Probabil că nici nu a fost nevoie să-l tastați pentru a ghici ce s-ar întâmpla: două foldere noi, unul numit alt și celălalt numit folder. Dacă doriți să lucrați cu spații în nume de director sau fișiere, trebuie să procedam un pic diferit. Introduceți următoarele comenzi pentru a încerca diferite moduri de a crea foldere cu spații în nume:

```
mkdir "folder 1"
mkdir 'folder 2'
mkdir folder\ 3
mkdir "folder 4" "folder 5"
mkdir -p "folder 6"/"folder 7"
ls
```

Deși linia de comandă poate fi folosită pentru a lucra cu fișiere și foldere cu spații în numele lor, necesitatea de a le declara cu ghilimele sau apostroafe face lucrurile puțin mai dificile. Adesea, puteți spune unei persoane care folosește linia de comandă mult doar din numele fișierelor sale: va avea tendința de a rămâne la litere și numere și va folosi litere de subliniere ("_") sau cratime ("-") în loc de spații.

3.2. Crearea de fișiere folosind redirecționare

Dosarul nostru demonstrativ începe să arate destul de plin de directoare, dar lipsesc oarecum fișierele. Să remediem acest lucru redirecționând rezultatul dintr-o comandă, astfel încât, în loc să fie imprimat pe ecran, să ajungă într-un fișier nou. Mai întâi, amintiți-vă ce afișează comanda ls în prezent:

```
ls
```

Să presupunem că dorim să captăm rezultatul acelei comenzi ca un fișier text pe care îl putem privi sau manipula în continuare. Tot ce trebuie să facem este să adăugăm caracterul mai mare (">") la sfârșitul liniei noastre de comandă, urmat de numele fișierului în care să scriem:

```
ls > output.txt
```

De data aceasta, nu este nimic tipărit pe ecran, deoarece rezultatul este redirecționat către fișierul nostru. Dacă rulați ls singur, ar trebui să vedeți că fișierul output.txt a fost creat. Putem folosi comanda cat pentru a ne uita la conținutul acesteia:

```
cat output.txt
```

OK, deci nu este exact ceea ce a fost afișat anterior pe ecran, dar conține toate aceleași date și este într-un format mai util pentru procesarea ulterioară. Să ne uităm la o altă comandă, echo:

```
echo „Acesta este un test”
```

Da, echo își imprimă argumentele din nou (de unde și numele). Dar combinați-l cu o redirecționare și aveți o modalitate de a crea cu ușurință fișiere de testare mici:

```
echo „Acesta este un test” > test_1.txt
echo „Acesta este un al doilea test” > test_2.txt
echo „Acesta este un al treilea test” > test_3.txt
ls
```

Ar trebui să accesați fiecare dintre aceste fișiere pentru a le verifica conținutul. Dar cat este mai mult decât un simplu vizualizator de fișiere - numele său provine de la „concatenate”. Dacă îi transmiteți mai mult de un nume de fișier, acesta va afișa fiecare dintre ele, unul după altul, ca un singur bloc de text:

```
cat test_1.txt test_2.txt test_3.txt
```

Acolo unde doriți să treceți mai multe nume de fișiere unei singure comenzi, există câteva comenzi rapide utile care vă pot economisi multă tastare dacă fișierele au nume similare. Un semn de întrebare ("?") poate fi folosit pentru a indica „orice caracter individual” în numele fișierului. Un asterisc ("*") poate fi folosit pentru a indica „zero sau mai multe caractere”. Acestea sunt uneori denumite caractere „wildcard”. Câteva exemple ar putea ajuta, următoarele comenzi fac toate același lucru:

```
cat test_1.txt test_2.txt test_3.txt
cat test_?.txt
cat test_*
```

Sunt necesare mai multe evadări

După cum probabil ați ghicit, această capacitate înseamnă, de asemenea, că trebuie să scăpați de numele fișierelor cu ? sau * caractere din ele, de asemenea. De obicei, este mai bine să evitați orice punctuație în numele fișierelor dacă doriți să le manipulați din linia de comandă.

Dacă te uiți la rezultatul lui ls, vei observa că singurele fișiere sau foldere care încep cu „t” sunt cele trei fișiere de test pe care tocmai le-am creat, așa că am putea chiar simplifica și mai mult ultima comandă la cat t*, adică „concatenează toate fișierele ale căror nume încep cu t și sunt urmate de zero sau mai multe alte caractere”. Să folosim această capacitate pentru a uni toate fișierele noastre într-un singur fișier nou, apoi să îl vedem:

```
cat t* > combinat.txt
cat combinat.txt
```

Ce crezi că se va întâmpla dacă rulăm acele două comenzi a doua oară? Se va plânge computerul, pentru că fișierul există deja? Va adăuga textul la fișier, astfel încât să conțină două copii? Sau îl va înlocui în întregime? Încercați să vedeți ce se întâmplă, dar pentru a evita să tastați din nou comenzile, puteți utiliza tastele săgeată sus și săgeată în jos pentru a vă deplasa înainte și înapoi prin istoricul comenzilor pe care le-ați folosit. Apăsați săgeata sus de câteva ori pentru a ajunge la primul cat și apăsați Enter pentru a rula, apoi faceți același lucru din nou pentru a ajunge la al doilea.

După cum vedeti, fișierul arată la fel. Asta nu pentru că a fost lăsat neatins, ci pentru că shell-ul șterge tot conținutul fișierului înainte de a scrie ieșirea comenzii cat în el. Din acest motiv, ar trebui să fiți foarte atenți când utilizați redirecționarea pentru a vă asigura că nu suprascrieți accidental un fișier de care aveți nevoie. Dacă doriți să adăugați, în loc să înlocuiți, conținutul fișierelor, dublați caracterul mai mare decât:

```
cat t* >> combinat.txt
echo "Am adăugat o linie!" >> combinat.txt
cat combinat.txt
```

Repetați primul cat de câteva ori, folosind săgeata în sus pentru comoditate și, poate, adăugați câteva comenzi echo arbitrare, până când documentul text este atât de mare încât nu va încăpea totul în terminal deodată când utilizați cat pentru a-l afișa. Pentru a vedea întregul fișier, acum trebuie să folosim un alt program, numit pager (pentru că vă afișează fișierul câte o „pagină” odată). Pager-ul standard de odinioară se numea more, deoarece pune o linie de text în partea de jos a fiecărei pagini care spune „–more–” pentru a indica că încă nu ați citit totul. În zilele noastre există un pager mult mai bun pe care ar trebui să îl utilizați: deoarece înlocuiește more, programatorii au decis să-l numească less.

```
less combinat.txt
```

Când vizualizați un fișier cu less, puteți utiliza tastele Săgeată sus, Săgeată în jos, Pagina în sus, Pagina în jos, Home și End pentru a vă deplasa prin fișier. Încearcati-le să vedeti diferența dintre ele. Când ați terminat de vizualizat fișierul, apăsați pe q pentru a ieși din less și a reveni la linia de comandă.

3.3. Mutarea și manipularea fișierelor

Acum că avem câteva fișiere, să ne uităm la felul de sarcini de zi cu zi pe care ar putea fi necesar să le efectuați. În practică, cel mai probabil veți folosi un program grafic atunci când

doriți să mutați, redenumiți sau ștergeți unul sau două fișiere, dar a ști cum să faceți acest lucru folosind linia de comandă poate fi util pentru modificări în bloc sau când fișierele sunt răspândite între foldere diferite. În plus, veți afla mai multe lucruri despre linia de comandă pe parcurs.

Să începem prin a pune fișierul nostru combinat.txt în directorul nostru dir1, folosind comanda mv (mutare):

```
mv combinat.txt dir1
```

Puteți confirma că lucrarea a fost efectuată folosind ls pentru a vedea că lipsește din directorul de lucru, apoi cd dir1 pentru a schimba în dir1, ls pentru a vedea că este acolo, apoi cd .. pentru a muta directorul de lucru înapoi. Sau puteți economisi o mulțime de tastare trecând direct o cale către comanda ls pentru a ajunge direct la confirmarea pe care o căutați:

```
ls dir1
```

Acum să presupunem că se dovedește că fișierul nu ar trebui să fie în dir1 până la urmă. Să-l mutăm înapoi în directorul de lucru. Am putea cd în dir1, apoi folosim mv combinat.txt .. pentru a spune „mutați combined.txt în directorul părinte”. Dar putem folosi o altă comandă rapidă pentru a evita schimbarea directorului. În același mod în care două puncte (..) reprezintă directorul părinte, deci un singur punct (.) poate fi folosit pentru a reprezenta directorul de lucru curent. Deoarece știm că există un singur fișier în dir1, putem, de asemenea, să folosim „*” pentru a se potrivi cu orice nume de fișier din acel director, economisindu-ne încă câteva apăsări de taste. Comanda noastră de a muta fișierul înapoi în directorul de lucru devine așadar aceasta (rețineți spațiul dinaintea punctului, sunt doi parametri trecuți către mv):

```
mv dir1/* .
```

Comanda mv ne permite, de asemenea, să mutăm mai mult de un fișier odată. Dacă treceți mai mult de două argumente, ultimul este considerat directorul de destinație, iar celelalte sunt considerate fișiere (sau directoare) de mutat. Să folosim o singură comandă pentru a muta combinat.txt, toate fișierele noastre test_n.txt și dir3 în dir2. Se întâmplă multe in urmatoarea comanda, dar dacă ne uitam la fiecare argument pe rand, ar trebui să putem descoperi ce se întâmplă:

```
mv combinat.txt test_* dir3 dir2

ls

ls dir2
```

Cu combinat.txt mutat acum în dir2, ce se întâmplă dacă decidem că este din nou în locul greșit? În loc de dir2, ar fi trebuit să fie introdus în dir6, care este cel care se află în dir5, care este în dir4. Cu ceea ce știm acum despre căi, nici asta nu este o problemă:

```
mv dir2/combinat.txt dir4/dir5/dir6

ls dir2

ls dir4/dir5/dir6
```

Observați cum comanda noastră mv ne permite să mutăm fișierul dintr-un director în altul, chiar dacă directorul nostru de lucru este cu totul diferit. Aceasta este o proprietate

importantă a liniei de comandă: indiferent de locul în care vă aflați în sistemul de fișiere, este încă posibil să operați pe fișiere și foldere în locații total diferite.

Deoarece se pare că folosim (și mutăm) mult acel fișier, poate că ar trebui să păstrăm o copie a acestuia în directorul nostru de lucru. La fel cum comanda mv mută fișiere, așadar comanda cp le copiază (din nou, rețineți spațiul dinaintea punctului):

```
cp dir4/dir5/dir6/combinat.txt .
ls dir4/dir5/dir6
ls
```

Grozav! Acum să creăm o altă copie a fișierului, în directorul nostru de lucru, dar cu un alt nume. Putem folosi din nou comanda cp, dar în loc să îi dăm o cale de director ca ultim argument, îi vom da un nou nume de fișier:

```
cp combined.txt backup_combinat.txt
ls
```

Este bine, dar poate că alegerea numelui de rezervă ar putea fi mai bună. De ce să nu-l redenumiți astfel încât să apară întotdeauna lângă fișierul original într-o listă sortată. Linia de comandă tradițională Unix gestionează o redenumire ca și cum ați muta fișierul de la un nume la altul, așa că vechiul nostru prieten mv este comanda de utilizat. În acest caz, specificați doar două argumente: fișierul pe care doriți să îl redenumiți și noul nume pe care doriți să îl utilizați.

```
mv backup_combinat.txt combinat_backup.txt
ls
```

Acest lucru funcționează și pe directoare, oferindu-ne o modalitate de a le sorta pe cele dificile cu spații în nume pe care l-am creat mai devreme. Pentru a evita retastarea fiecărei comenzi după prima, utilizați săgeata sus pentru a folosi comanda anterioară din istoric. Puteți edita apoi comanda înainte de a o rula, deplasând cursorul la stânga și la dreapta cu tastele săgeți și eliminând caracterul la stânga cu Backspace sau pe cel pe care se află cursorul cu Delete. În cele din urmă, tastați noul caracter la locul său și apăsați Enter sau Return pentru a rula comanda după ce ați terminat. Asigurați-vă că modificați ambele apariții ale numărului din fiecare dintre aceste rânduri.

```
mv „folder 1” folder_1
mv „folder 2” folder_2
mv „folder 3” folder_3
mv „folder 4” folder_4
mv „folder 5” folder_5
mv „folder 6” folder_6
ls
```

3.4. Ștergerea fișierelor și folderelor

Atenție

În secțiunea următoare vom începe să ștergem fișiere și foldere. Pentru a vă asigura că nu ștergeți accidental nimic din folderul home, utilizați comanda pwd pentru a verifica din nou dacă vă aflați încă în directorul /tmp/tutorial înainte de a continua.

Acum știm cum să mutăm, să copiem și să redenumim fișierele și directoarele. Având în vedere că acestea sunt doar fișiere de testare, totuși, poate că nu avem nevoie de trei copii diferite ale combinat.txt până la urmă. Să facem puțin ordine, folosind comanda rm (eliminare):

```
rm dir4/dir5/dir6/combined.txt combined_backup.txt
```

Poate că ar trebui să eliminăm și unele dintre acele directoare în exces:

```
rm folder_*
```

Ce s-a întâmplat acum? Ei bine, se pare că rm are o mică plasă de siguranță. Sigur, îl puteți folosi pentru a șterge fiecare fișier dintr-un director cu o singură comandă, ștergând accidental mii de fișiere într-o clipă, fără nicio modalitate de a le recupera. Dar nu vă va permite să ștergeți un director. Presupun că asta vă ajută să vă împiedicați să ștergeți accidental alte mii de fișiere, dar pare puțin meschin ca o comandă atât de distructivă să se oprească la eliminarea unui director gol. Din fericire, există o comandă rmdir (elimină directorul) care va face treaba pentru noi:

```
rmdir folder_*
```

Ei bine, este puțin mai bine, dar există încă o eroare. Dacă rulați ls, veți vedea că majoritatea dosarelor au dispărut, dar folder_6 încă mai așteaptă. După cum vă amintiți, folder_6 are încă un folder 7 în el, iar rmdir va șterge doar folderele goale. Din nou, este o mică plasă de siguranță pentru a vă împiedica să ștergeți accidental un folder plin de fișiere atunci când nu ați vrut.

În acest caz, totuși, ne dorim. Adăugarea de opțiuni la comenzile noastre rm sau rmdir ne va permite să efectuăm acțiuni periculoase fără ajutorul unei plase de siguranță! În cazul rmdir putem adăuga un comutator -p pentru a-i spune să elimine și directoarele părinte. Gândiți-vă la el ca la contrapunctul pentru mkdir -p. Deci, dacă ar fi să rulați rmdir -p dir1/dir2/dir3, mai întâi ar șterge dir3, apoi dir2, apoi în cele din urmă șterge dir1. Totuși, urmează regulile normale rmdir de ștergere doar a directoarelor goale, deci, dacă ar exista și un fișier în dir1, de exemplu, numai dir3 și dir2 ar fi eliminate.

O abordare mai obișnuită, când sunteți cu adevărat, într-adevăr, foarte sigur că doriți să ștergeți un întreg director și orice din el, este să îi spuneți lui rm să lucreze recursiv folosind comutatorul -r, caz în care va șterge cu plăcere folderele ca precum și fișierele. Având în vedere acest lucru, iată comanda pentru a scăpa de acel neplăcut folder_6 și de subdirectorul din el:

```
rm -r folder_6
ls
```

Amintiți-vă: deși rm -r este rapid și convenabil, este și periculos. Cel mai sigur este să ștergeți în mod explicit fișierele pentru a șterge un director, apoi cd .. către directorul părinte înainte de a utiliza rmdir pentru a-l elimina.

Atenție!

Spre deosebire de interfețele grafice, rm nu mută fișierele într-un folder numit „coș de gunoi” sau similar. În schimb, le șterge total, total și irevocabil. Trebuie să fiți extrem de atenți cu parametrii pe care îi utilizați cu rm pentru a vă asigura că ștergeți doar fișierele pe care intenționați. Ar trebui să aveți grijă deosebită când utilizați metacaracterele, deoarece este ușor să ștergeți accidental mai multe fișiere decât ați vrut. Un caracter de spațiu rătăcit din comanda dumneavoastră îl poate schimba complet: rm t* înseamnă „șterge toate fișierele care încep cu t”, în timp ce rm t * înseamnă „șterge fișierul t, precum și orice fișier al cărui nume este format din zero sau mai multe caractere, care ar fi totul în director! Dacă nu sunteți deloc sigur, utilizați opțiunea -i (interactivă) pentru rm, care vă va cere să confirmați ștergerea fiecărui fișier; introduceți Y pentru a-l șterge, N pentru a-l păstra și apăsați Ctrl-C pentru a opri complet operațiunea.

4. COMENZI UTILE ÎN SISTEME LINUX - LINIA DE COMANDĂ, SUPERUTILIZATORUL ȘI FIȘIERELE ASCUNSE

Calculatoarele și telefoanele din ziua de astăzi au genul de capacități grafice și audio pe care utilizatorii noștri de terminale din anii ‘70 nici nu și-ar putea imagina. Cu toate acestea, textul predomină ca mijloc de organizare și clasificare a fișierelor. Fie că este vorba de numele fișierului în sine, de coordonatele GPS încorporate în fotografiile pe care le faceți pe telefon sau de metadatele stocate într-un fișier audio, textul încă joacă un rol vital în fiecare aspect al calculului. Este norocos pentru noi că linia de comandă Linux include câteva instrumente puternice pentru manipularea conținutului text și modalități de a uni aceste instrumente pentru a crea ceva și mai capabil.

Să începem cu o întrebare simplă. Câte linii există în fișierul output.txt? Comanda wc (word count) ne poate spune, folosind comutatorul -l pentru a-i spune, ca vrem doar numărarea liniilor (poate face și numărătoare de caractere și, după cum sugerează și numele, numărătoare de cuvinte):

```
wc -l output.txt
```

În mod similar, dacă doriți să știți câte fișiere și foldere sunt în directorul personal și apoi să faceți ordine, puteți face acest lucru:

```
ls ~ > lista_fișiere.txt
wc -l lista_fișiere.txt
rm lista_fișiere.txt
```

Această metodă funcționează, dar crearea unui fișier temporar care să rețină rezultatul de la ls doar pentru a-l șterge două linii mai târziu pare puțin excesiv. Din fericire, linia de comandă Unix oferă o comandă rapidă care evită să creați un fișier temporar, luând rezultatul de la o comandă (denumită ieșire standard sau STDOUT) și introducându-l direct ca intrare la o altă comandă (intrare standard sau STDIN). Este ca și cum ați conecta o conductă între ieșirea unei comenzi și intrarea următoarei comenzi, atât de mult încât acest proces este de fapt denumit “piping” a datelor de la o comandă la alta. Iată cum să transferăm rezultatul comenzii noastre ls în wc:

```
ls ~|wc -l
```

Observați că nu este creat niciun fișier temporar și nu este necesar un nume de fișier. Tuburile funcționează în întregime în memorie și majoritatea instrumentelor de linie de comandă Unix se vor aștepta să primească intrare de la un tub dacă nu specificați un fișier pentru care să lucreze. Privind la linia de mai sus, puteți vedea că sunt două comenzi, ls ~ (enumeră conținutul directorului principal) și wc -l (numără liniile), separate printr-un caracter de bară verticală ("|"). Acest proces de introducere a unei comenzi în alta este atât de frecvent utilizat încât caracterul în sine este adesea denumit caracterul “pipe”, așa că dacă vedeți acel termen, acum știți că înseamnă doar bara verticală.

Rețineți că spațiile din jurul caracterului pipe nu sunt importante, le-am folosit pentru claritate, dar următoarea comandă funcționează la fel de bine, de data aceasta pentru a ne spune câte articole sunt în directorul /etc:

```
ls /etc | wc -l
```

Destul de multe fișiere, nu? Dacă am dori să le enumerăm pe toate, ar umple în mod clar mai mult de un singur ecran. După cum am descoperit mai devreme, atunci când o comandă produce o mulțime de rezultate, este mai bine să folosiți less pentru a o vizualiza, iar acel sfat se aplică în continuare atunci când utilizați o conductă (rețineți că apăsați q pentru a ieși):

```
ls /etc | less
```

Revenind la propriile fișiere, știm cum să obținem numărul de linii în output.txt, dar având în vedere că a fost creat prin concatenarea acelorași fișiere de mai multe ori, mă întreb câte linii unice există? Unix are o comandă, uniq, care va scoate numai linii unice în fișier. Așa că trebuie să scoatem fișierul și să-l conducem prin uniq. Dar tot ce ne dorim este un număr de linii, așa că trebuie să folosim și wc. Din fericire, linia de comandă nu vă limitează la o singură conductă odată, așa că putem continua să înlănțuim câte comenzi avem nevoie:

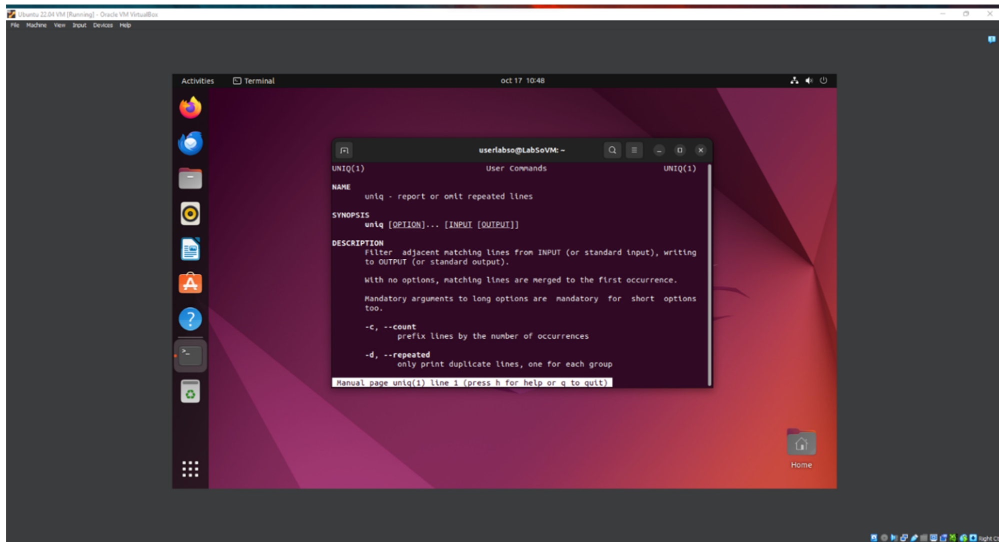
```
cat output.txt | uniq | wc -l
```

Acea linie a dus probabil la un număr care este destul de aproape de numărul total de linii din fișier, dacă nu chiar același. Sigur nu poate fi corect? Decupați ultima țeavă pentru a vedea rezultatul comenzii pentru o idee mai bună despre ceea ce se întâmplă. Dacă fișierul este foarte lung, este posibil să doriți să îl treceți prin less pentru a fi mai ușor de inspectat:

```
cat output.txt | uniq | less
```

Se pare că foarte puține dintre liniile noastre duplicate sunt eliminate. Pentru a înțelege de ce, trebuie să ne uităm la documentația pentru comanda uniq. Majoritatea instrumentelor din linia de comandă vin cu un manual de instrucțiuni scurt (uneori nu chiar scurt), accesat prin comanda man (manual). Ieșirea este transmisă automat prin pager, care de obicei va fi less, astfel încât să vă puteți deplasa înainte și înapoi prin continut, apoi apăsați q când ați terminat:

```
man uniq
```



Deoarece acest tip de documentație este accesata prin comanda man, o veți auzi denumită „pagină de manual", ca în „verificați pagina de manual pentru mai multe detalii". Formatul paginilor de manual este adesea concis, gândiți-vă la ele mai mult ca la o prezentare rapidă a unei comenzi decât la un tutorial complet. Acestea sunt adesea foarte tehnice, dar de obicei puteți sări peste cea mai mare parte a conținutului și doar să căutați detaliile opțiunii sau argumentului pe care îl utilizați.

Pagina de manual uniq este un exemplu tipic, deoarece începe cu o scurtă descriere pe o linie a comenzii, trece la un rezumat al modului de utilizare, apoi are o descriere detaliată a fiecărei opțiuni sau parametru. Dar, deși paginile de manual sunt neprețuite, ele pot fi, de asemenea, impenetrabile. Sunt utilizate cel mai bine atunci când aveți nevoie de un memento al unui anumit comutator sau parametru, mai degrabă decât ca o resursă generală pentru a învăța cum să utilizați linia de comandă. Cu toate acestea, prima linie a secțiunii DESCRIERE pentru man uniq răspunde la întrebarea de ce liniile duplicate nu au fost eliminate: funcționează doar pe liniile adiacente care se potrivesc.

Întrebarea, atunci, este cum să rearanjam liniile din fișierul nostru, astfel încât intrările duplicate să fie pe linii adiacente. Dacă ar fi să sortăm conținutul fișierului în ordine alfabetică, asta ar rezolva problema. Unix oferă o comandă de sortare sort pentru a face exact asta. O verificare rapidă a man sort arată că putem transmite un nume de fișier direct la comandă, așa că haideți să vedem ce face fișierul nostru:

```
sort output.txt | less
```

Ar trebui să puteți vedea că liniile au fost reordonate și că acum este potrivit pentru conectarea direct în uniq. În sfârșit, ne putem îndeplini sarcina de a număra liniile unice din fișier:

```
sort output.txt | uniq | wc -l
```

După cum puteți vedea, capacitatea de a transfera date de la o comandă la alta, construind lanțuri lungi pentru a vă manipula datele, este un instrument puternic, care reduce nevoia de fișiere temporare și vă economisește mult timp de tastare. Din acest motiv, veți vedea că este folosit destul de des în liniile de comandă. Un lanț lung de comenzi poate părea intimidant la început, dar amintiți-vă că puteți împărți chiar și cel mai lung lanț în comenzi individuale (și vă uitați la paginile lor de manual) pentru a înțelege mai bine ceea ce face.

4.1. Linia de comandă și superutilizatorul

Un motiv bun pentru a învăța câteva elemente de bază ale liniei de comandă este faptul că instrucțiunile online vor favoriza adesea utilizarea comenzilor shell în detrimentul unei interfețe grafice. Acolo unde aceste instrucțiuni necesită modificări ale datelor care depășesc modificarea câtorva fișiere din directorul home, vă veți confrunta inevitabil cu comenzi care trebuie să fie executate ca administrator al mașinii (sau superutilizator în limbajul Unix). Înainte de a începe să rulați comenzi arbitrare pe care le găsiți într-un colț întunecat al internetului, merită să înțelegeți implicațiile rulării ca administrator și cum să identificați acele instrucțiuni care necesită acest lucru, astfel încât să puteți evalua mai bine dacă sunt sigure de rulat sau nu.

Superutilizatorul este, după cum sugerează și numele, un utilizator cu "super-puteri". În sistemele mai vechi era un utilizator real, cu un nume de utilizator real (aproape întotdeauna „root") pe care te puteai autentifica ca și cum ai avea parola. Cât despre acele super-puteri: root poate modifica sau șterge orice fișier din orice director din sistem, indiferent de cine le deține; root poate rescrie regulile firewall sau porni servicii de rețea care ar putea deschide sistemul la un atac; root poate opri mașina chiar dacă alți oameni încă o folosesc. Pe scurt, root poate face aproape orice, sărind cu ușurință peste măsurile de siguranță care sunt de obicei puse în aplicare pentru a împiedica utilizatorii să-și depășească limitele.

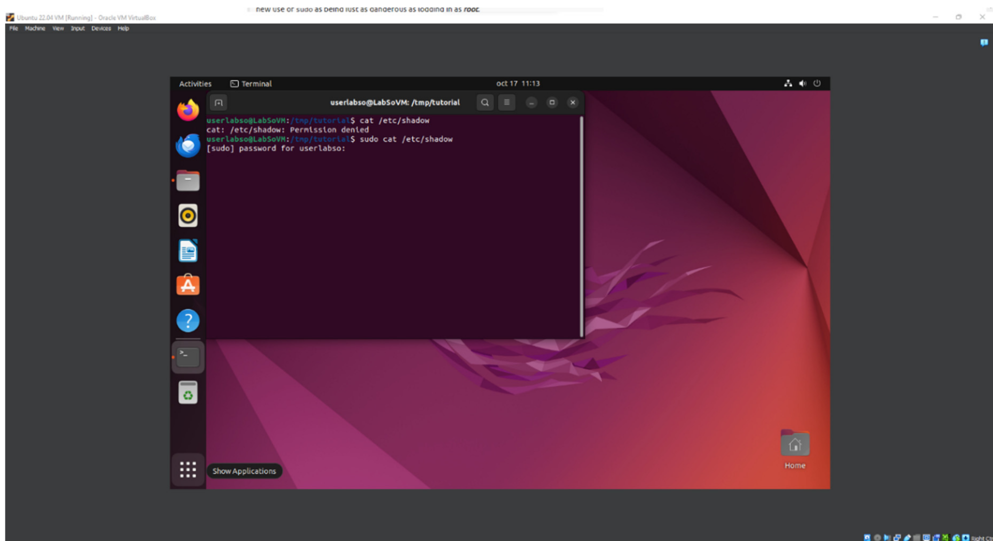
Desigur, o persoană conectată ca root este la fel de capabilă să facă greșeli ca oricine altcineva. Analele istoriei informatice sunt pline de povești despre o comandă greșită care șterge întregul sistem de fișiere sau ucide un server vital. Apoi, există posibilitatea unui atac rău intenționat: dacă un utilizator este conectat ca root și își părăsește biroul, atunci nu este prea dificil pentru un coleg nemulțumit să comute pe sistemul sau și să facă ravagii. În ciuda acestui fapt, natura umană fiind ceea ce este, mulți administratori de-a lungul anilor s-au făcut vinovați de folosirea root ca si cont principal sau unic.

Într-un efort de a reduce aceste probleme, multe distribuții Linux au început să încurajeze utilizarea comenzii su. Aceasta este descrisa în mod diferit ca fiind prescurtare pentru „superuser" sau „ switch user" și vă permite să treceți la un alt utilizator pe mașină fără a fi nevoie să vă deconectați și să vă conectați din nou. Când este folosit fără argumente, presupune că doriți să treceți la utilizatorul root (de unde prima interpretare a numelui), dar îi puteți transmite un nume de utilizator pentru a trece la un anumit cont de utilizator (a doua interpretare). Prin încurajarea utilizării su, scopul a fost de a convinge administratorii să-și petreacă cea mai mare parte a timpului folosind un cont normal, să treacă la contul de superutilizator numai atunci când au nevoie și apoi să folosească comanda de deconectare (sau comanda rapidă Ctrl-D) cât mai curând posibil, pentru a reveni la contul lor la nivel de utilizator.

Prin reducerea la minimum a timpului petrecut conectat ca root, utilizarea su reduce fereastra de oportunitate în care să faci o greșeală catastrofală. În ciuda acestui fapt, mulți administratori s-au făcut vinovați de lăsarea terminalelor deschise pentru o lungă durată în care au folosit su pentru a trece la contul root. În acest sens, su a fost doar un mic pas înainte pentru securitate.

Presupunând că sunteți pe un sistem Linux care utilizează sudo și contul dvs. este configurat ca administrator, încercați următoarele pentru a vedea ce se întâmplă atunci când încercați să accesați un fișier care este considerat sensibil (conține parole criptate):

```
cat /etc/shadow
sudo cat /etc/shadow
```



Dacă introduceți parola când vi se solicită, ar trebui să vedeți conținutul fișierului /etc/shadow. Acum rulați din nou sudo cat /etc/shadow. De data aceasta, fișierul va fi afișat fără a vă solicita o parolă, deoarece este încă în cache.

Pentru instrucțiuni care vizează Ubuntu, un aspect obișnuit al sudo este atunci cand instalați software nou pe sistem folosind comenzile apt sau apt-get. Dacă instrucțiunile vă cer să adăugați mai întâi un nou depozit de software la sistem, folosind comanda apt-add-repository, prin editarea fișierelor în /etc/apt sau folosind un „PPA” (Personal Package Archive), ar trebui să fiți atent deoarece aceste surse nu sunt mentinute de Canonical. Dar adesea instrucțiunile vă cer doar să instalați software din depozitele standard, care ar trebui să fie sigure.

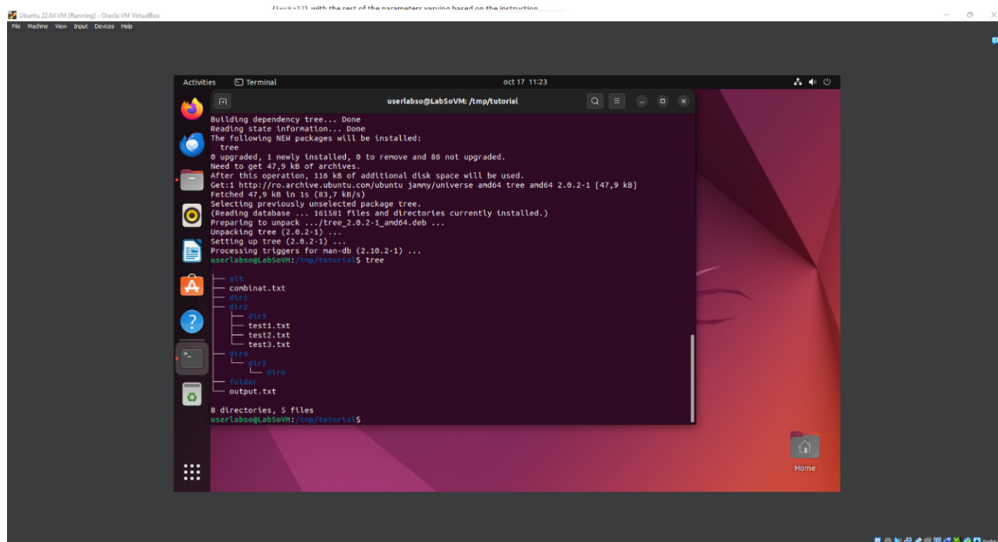
Să instalăm un nou program de linie de comandă din depozitele standard Ubuntu pentru a ilustra această utilizare a sudo:

```
sudo apt install tree
```

Odată ce ați furnizat parola, programul apt va tipări câteva rânduri de text pentru a vă spune ce face. Programul arbore este doar mic, așa că nu ar trebui să dureze mai mult de un minut sau două pentru a descărca și instala pentru majoritatea utilizatorilor. Odată ce revii la

linia de comandă normală, programul este instalat și gata de utilizare. Să-l rulăm pentru a obține o imagine de ansamblu mai bună a cum arată colecția noastră de fișiere și foldere:

```
cd /tmp/tutorial
tree
```



Revenind la comanda care a instalat de fapt noul program (sudo apt install tree), arată ușor diferit de cele pe care le-ați văzut până acum. În practică funcționează astfel:

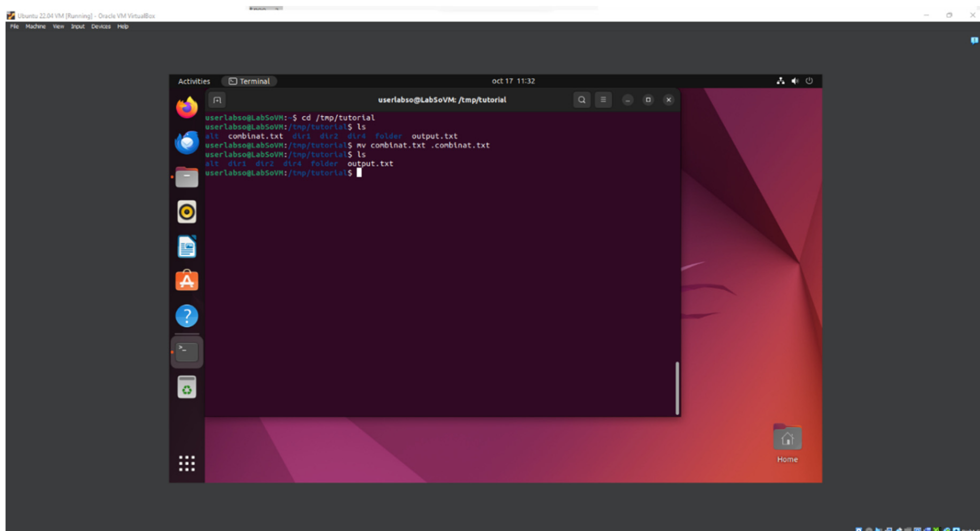
1. Comanda sudo, atunci când este utilizată fără opțiuni, va presupune că primul parametru este o comandă pentru a rula cu privilegii de superutilizator. Orice alți parametri vor fi transferați direct la noua comandă. Comutatoarele sudo încep toate cu una sau două cratime și trebuie să urmeze imediat comanda sudo, astfel încât să nu existe nicio confuzie cu privire la faptul dacă al doilea parametru de pe linie este o comandă sau o opțiune.
2. Comanda în acest caz este apt. Spre deosebire de celelalte comenzi pe care le-am văzut, aceasta nu funcționează direct cu fișiere. În schimb, se așteaptă ca primul său parametru să fie o instrucțiune de executat (install), restul parametrilor variind în funcție de instrucțiune.
3. În acest caz, comanda install îi spune comenzii apt că restul liniei de comandă va consta dintr-unul sau mai multe nume de pachete de instalat din depozitele de software ale sistemului. De obicei, aceasta va adăuga software nou la mașină, dar pachetele ar putea fi orice colecție de fișiere care trebuie instalate în anumite locații, cum ar fi fonturi sau imagini de pe desktop.

Puteți pune sudo în fața oricărei comenzi pentru a o rula ca superutilizator, dar rareori este nevoie. Chiar și fișierele de configurare a sistemului pot fi adesea vizualizate (cu cat sau less) ca un utilizator normal și necesită privilegii root doar dacă trebuie să le editați.

4.2. Fișiere ascunse

Fișierele (și folderele) ascunse sunt utilizate în mod obișnuit pe sistemele Linux pentru a stoca setări și date de configurare și sunt de obicei ascunse pur și simplu pentru a nu aglomera vizualizarea propriilor fișiere. Nu există nimic special la un fișier sau folder ascuns, în afară de numele său: pur și simplu începerea unui nume cu un punct (".") este suficient pentru a-l face să dispară.

```
cd /tmp/tutorial
ls
mv combinat.txt .combinat.txt
ls
```



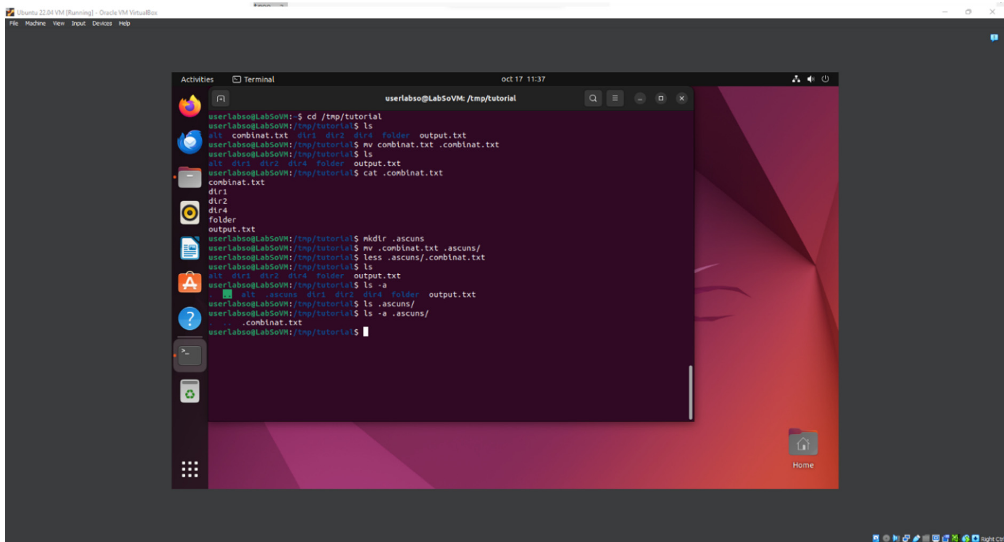
Puteți lucra în continuare cu fișierul ascuns, asigurându-vă că includeți punctul când îi specificați numele fișierului:

```
cat .combinat.txt
mkdir .ascuns
mv .combinat.txt .ascuns
less .ascuns/.combinat.txt
```

Dacă rulați `ls`, veți vedea că directorul `.ascuns` este, așa cum v-ați aștepta, ascuns. Puteți lista în continuare conținutul său folosind `ls .ascuns`, dar deoarece conține doar un singur fișier care este, el însuși, ascuns, nu veți obține prea multe rezultate. Dar puteți folosi comutatorul `-a` (show all) la `ls` pentru a face ca acesta să arate totul dintr-un director, inclusiv fișierele și folderele ascunse:

```
ls
ls -a
```

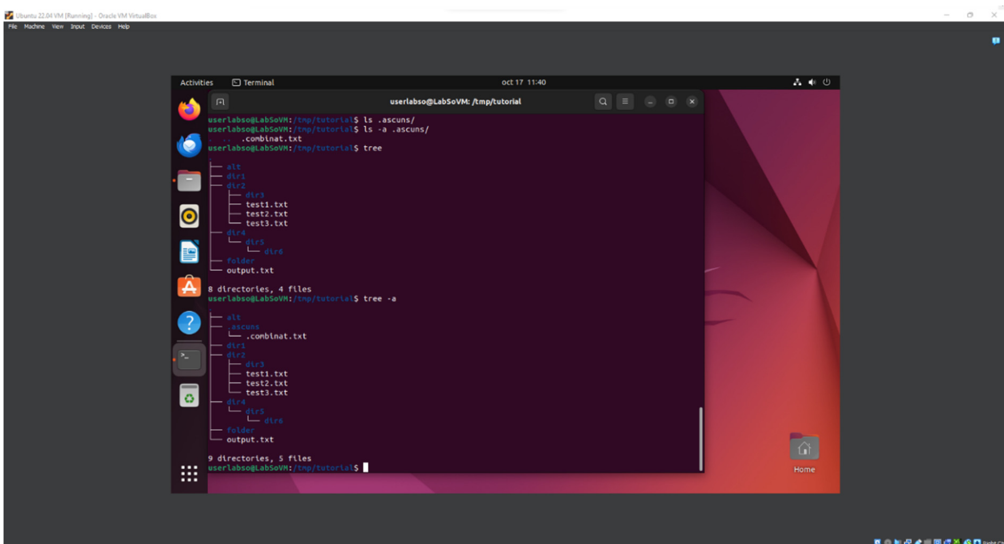
```
ls .ascuns
ls -a .ascuns
```



Observați comenzile rapide pe care le-am folosit anterior, . și .., par, de asemenea, ca și cum ar fi directoare reale.

În ceea ce privește comanda noastră tree instalată recent, aceasta funcționează într-un mod similar (cu excepția apariției . și ..):

```
tree
tree -a
```



Reveniți la directorul home (cd) și încercați să rulați ls fără și apoi cu comutatorul -a. Transmiteți output-ul prin wc -l pentru a vă oferi o idee mai clară despre câte fișiere și foldere

ascunse au fost chiar sub nasul vostru în tot acest timp. Aceste fișiere stochează de obicei configurația personală și este modul în care sistemele Unix au oferit întotdeauna capacitatea de a avea setări la nivel de sistem (de obicei în /etc) care pot fi suprascrise de către utilizatorii individuali (prin fișierele ascunse din directorul lor principal).

De obicei, nu ar trebui să lucrati cu fișiere ascunse, dar, ocazional, anumite instrucțiuni vă pot cere să faceti un cd în .config sau să editați un fișier al cărui nume începe cu un punct. Cel puțin acum veți înțelege ce se întâmplă, chiar și atunci când nu puteți vedea cu ușurință fișierul în interfața grafică.

5. COMENZI UTILE ÎN SISTEME LINUX – PERMISIUNI ȘI CONTROLUL PROCESELOR

5.1. Permisiuni

Sistemele de operare asemănătoare Unix, cum ar fi Ubuntu, diferă de alte sisteme de calcul prin faptul că nu sunt doar multitasking, ci și multi-utilizator.

Ce înseamnă mai exact asta? Înseamnă că mai mult de un utilizator poate opera computerul în același timp. În timp ce un computer desktop sau laptop are doar o tastatură și un monitor, acesta poate fi folosit de mai mult de un utilizator. De exemplu, dacă computerul este atașat la o rețea sau la Internet, utilizatorii de la distanță se pot conecta prin ssh (secure shell) și pot opera computerul. De fapt, utilizatorii de la distanță pot executa aplicații grafice și pot avea rezultatul afișat pe un computer la distanță. Sistemul X Window acceptă acest lucru.

Capacitatea multi-utilizator a sistemelor de tip Unix este o caracteristică care este adânc înrădăcinată în designul sistemului de operare. Dacă ne amintim de mediul în care a fost creat Unix, acest lucru are perfect sens. Cu ani în urmă, înainte ca computerele să fie „personale", acestea erau mari, scumpe și centralizate. Un sistem computerizat universitar tipic consta dintr-un computer central mare situat într-o clădire din campus, iar terminalele erau amplasate în întregul campus, fiecare conectat la computerul central mare. Computerul sprijinea mulți utilizatori în același timp.

Pentru a face acest lucru practic, a trebuit să fie concepută o metodă care să protejeze utilizatorii unul de celălalt. La urma urmei, nu am dori ca acțiunile unui utilizator să blocheze computerul și nici nu am permite unui utilizator să interfereze cu fișierele aparținând altui utilizator.

Această lecție va acoperi următoarele comenzi:

chmod - modifica drepturile de acces la fișiere

su – permite temporar acțiuni ca superutilizator

sudo - permite temporar acțiuni ca superutilizator

chown - schimbă proprietarul fișierului

chgrp - schimba grupul de proprietari al unui fișier

5.1.1. Permisiuni pentru fișiere

Pe un sistem Linux, fiecărui fișier și director i se atribuie drepturi de acces pentru proprietarul fișierului, pentru membrii unui grup de utilizatori înrudiți și pentru toți ceilalți. Drepturile pot fi atribuite pentru a citi un fișier, pentru a scrie un fișier și pentru a executa un fișier (adică, rulați fișierul ca program).

Pentru a vedea setările de permisiuni pentru un fișier, putem folosi comanda ls. Ca exemplu, ne vom uita la programul bash care se află în directorul /bin:

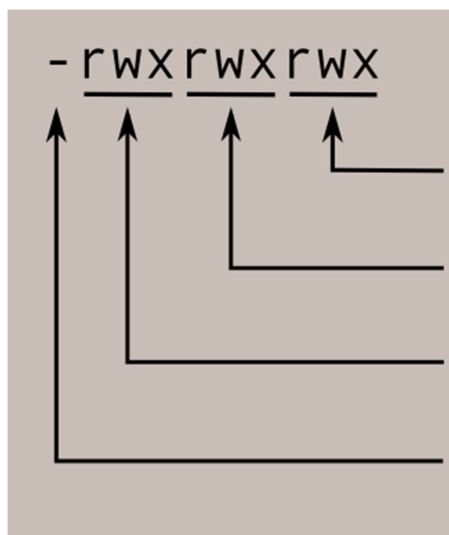
```
ls -l /bin/bash
```

```
userlabso@LabSoVM:~$ ls -l /bin/bash
-rwxr-xr-x 1 root root 1396520 ian  6  2022 /bin/bash
userlabso@LabSoVM:~$
```

Aici putem vedea:

- fișierul „/bin/bash” este deținut de utilizatorul „root”
- superutilizatorul are dreptul de a citi, scrie și executa acest fișier
- fișierul este deținut de grupul „root”
- membrii grupului „root” pot citi și executa și acest fișier
- toți ceilalți pot citi și executa acest fișier

În diagrama de mai jos, vedem cum este interpretată prima porțiune a listării. Constă dintr-un caracter care indică tipul fișierului, urmat de trei seturi de trei caractere care transmit permisiunea de citire, scriere și execuție pentru proprietar, grup și pentru toți ceilalți.



permisiuni de citire, scriere și executare pentru toți ceilalți utilizatori

permisiuni de citire, scriere și executare pentru proprietarul grupului al fișierului

permisiuni de citire, scriere și executare pentru proprietarul fișierului

tip de fișier: „-” indică un fișier obișnuit, d indică un director

```
chmod
```

Comanda chmod este folosită pentru a modifica permisiunile unui fişier sau director. Pentru a o folosi, specificăm setările de permisiuni dorite şi fişierul sau fişierele pe care dorim să le modificăm. Există două modalitati de a specifica permisiunile. În această lecție ne vom concentra pe una dintre acestea, numită metoda notației octale.

Este uşor să ne gândim la setările de permisiuni ca la o serie de biți (aşa cum le consideră computerul). Iată cum funcționează:

```
rwx rwx rwx = 111 111 111
rw- rw- rw- = 110 110 110
rwx --- --- = 111 000 000
```

şi aşa mai departe...

```
rwx = 111 în binar = 7
rw- = 110 în binar = 6
r-x = 101 în binar = 5
r-- = 100 în binar = 4
```

Acum, dacă reprezentăm fiecare dintre cele trei seturi de permisiuni (proprietar, grup şi altele) ca o singură cifră, avem o modalitate destul de convenabilă de a exprima posibilele setări de permisiuni. De exemplu, dacă am dori să setăm "some_file" să aibă permisiunea de citire şi scriere pentru proprietar, dar am vrea să păstrăm fişierul privat față de alții, am:

chmod 600 some_file

```
userlabso@LabSoVM:~$ mkdir /tmp/tutorial
userlabso@LabSoVM:~$ cd /tmp/tutorial/
userlabso@LabSoVM:/tmp/tutorial$ ls > some_file
userlabso@LabSoVM:/tmp/tutorial$ ls -l
total 4
-rw-rw-r-- 1 userlabso userlabso 10 oct 27 14:13 some_file
userlabso@LabSoVM:/tmp/tutorial$ chmod 600 some_file
userlabso@LabSoVM:/tmp/tutorial$ ls -l
total 4
-rw------- 1 userlabso userlabso 10 oct 27 14:13 some_file
userlabso@LabSoVM:/tmp/tutorial$
```

Iată un tabel cu numere care acoperă toate setările comune. Cele care încep cu „7" sunt folosite cu programe (din moment ce permit execuția), iar restul sunt pentru alte tipuri de fişiere.

Valoare	Semnificatie
777	(rwxrwxrwx) Fără restricții privind permisiunile. Oricine poate face orice. În general, nu este o setare de dorit.
755	(rwxr-xr-x) Proprietarul fişierului poate citi, scrie şi executa fişierul. Toți ceilalți pot citi şi executa fişierul. Această setare este comună pentru programele care sunt folosite de toți utilizatorii.

700	`(rwx------)` Proprietarul fișierului poate citi, scrie și executa fișierul. Nimeni altcineva nu are drepturi. Această setare este utilă pentru programele pe care numai proprietarul le poate folosi și trebuie păstrate private față de ceilalți.
666	`(rw-rw-rw-)` Toți utilizatorii pot citi și scrie fișierul.
644	`(rw-r--r--)` Proprietarul poate citi și scrie un fișier, în timp ce toți ceilalți pot citi doar fișierul. O setare comună pentru fișierele de date pe care toată lumea le poate citi, dar numai proprietarul le poate schimba.
600	`(rw-------)` Proprietarul poate citi și scrie un fișier. Toți ceilalți nu au drepturi. O setare comună pentru fișierele de date pe care proprietarul dorește să le păstreze private.

5.1.2. Permisiuni pentru directoare

Comanda `chmod` poate fi folosită și pentru a controla permisiunile de acces pentru directoare. Din nou, putem folosi notația octală pentru a seta permisiunile, dar semnificația atributelor `r`, `w` și `x` este diferită:

r - Permite listarea conținutului directorului dacă este setat și atributul x.

w - Permite crearea, ștergerea sau redenumirea fișierelor din director dacă este setat și atributul x.

x - Permite accesarea unui director (adică `cd dir`).

Iată câteva setări utile pentru directoare:

Valoare	Semnificatie
777	`(rwxrwxrwx)` Fără restricții privind permisiunile. Oricine poate lista fișiere, poate crea fișiere noi în director și poate șterge fișierele din director. În general, nu este o setare bună.
755	`(rwxr-xr-x)` Proprietarul directorului are acces complet. Toți ceilalți pot lista directorul, dar nu pot crea fișiere și nici nu le pot șterge. Această setare este comună pentru directoarele pe care doriți să le partajați cu alți utilizatori.
700	`(rwx------)` Proprietarul directorului are acces deplin. Nimeni altcineva nu are drepturi. Această setare este utilă pentru directoarele pe care numai proprietarul le poate folosi și trebuie păstrate private față de ceilalți.

5.1.3. Trecerea la Superutilizator pentru o scurtă perioadă

Este adesea necesar să devenim superutilizator pentru a efectua sarcini importante de administrare a sistemului, dar după cum știm, nu ar trebui să rămânem conectați ca superutilizator. În majoritatea distribuțiilor, există un program care vă poate oferi acces

temporar la privilegiile superutilizatorului. Acest program se numește su (prescurtare pentru “substitute user”) și poate fi folosit în acele cazuri când trebuie să fii superutilizator pentru un număr mic de sarcini. Pentru a deveni superutilizator, tastați pur și simplu comanda su. Vi se va solicita parola superutilizatorului:

```
su
```

!!! (Cand ajungeti la acest pas solicitati informatii suplimentare)

```
userlabso@LabSoVM:/tmp/tutorial$ sudo passwd root
[sudo] password for userlabso:
New password:
Retype new password:
passwd: password updated successfully
userlabso@LabSoVM:/tmp/tutorial$ su
Password:
root@LabSoVM:/tmp/tutorial#
```

Nota informativa cont root

Activarea contului root:

Activarea contului root este rareori necesară. Aproape tot ce trebuie să faceți ca administrator al unui sistem Ubuntu se poate face prin sudo sau gksudo. Dacă într-adevăr aveți nevoie de o autentificare root persistentă, cea mai bună alternativă este să simulați un shell de conectare rădăcină folosind următoarea comandă:

```
sudo -i
```

Pentru a activa contul root (adică a seta o parolă), utilizați:

```
sudo passwd root
```

Folosiți pe propria răspundere in lucrul pe alte sisteme decat cele virtuale folosite pentru laborator!

Conectarea la X ca root poate cauza probleme foarte grave. Dacă credeți că aveți nevoie de un cont root pentru a efectua o anumită acțiune, vă rugăm să consultați mai întâi canalele oficiale de asistență, pentru a vă asigura că nu există o alternativă mai bună.

Re-dezactivarea contului root:

Dacă dintr-un motiv oarecare v-ați activat contul root și doriți să-l dezactivați din nou, utilizați următoarea comandă în terminal:

```
sudo passwd -dl root
```

După executarea comenzii su, avem o nouă sesiune shell ca superutilizator. Pentru a ieși din sesiunea de superutilizator, tastați exit și vom reveni la sesiunea anterioară.

În majoritatea distribuțiilor moderne, este utilizată o metodă alternativă. În loc să folosească su, aceste sisteme folosesc comanda sudo. Cu sudo, unuia sau mai multor utilizatori

li se acordă privilegii de superutilizator în funcție de necesități. Pentru a executa o comandă ca superutilizator, comanda dorită este pur și simplu precedată de comanda sudo. După ce este introdusă comanda, utilizatorului i se solicită propria parolă, mai degrabă decât cea a superutilizatorului:

```
userlabso@LabSoVM:/tmp/tutorial$ sudo passwd root
[sudo] password for userlabso:
```

De fapt, distribuțiile moderne nici măcar nu setează parola contului root, făcând astfel imposibilă conectarea ca utilizator root. Un shell rădăcină este încă posibil cu sudo utilizând opțiunea „-i”

5.1.4. Modificarea dreptului de proprietate asupra fișierului

Putem schimba proprietarul unui fișier folosind comanda chown. Iată un exemplu: Să presupunem că dorim să schimbăm proprietarul some_file din „your_username” în „another_user”. Am putea:

(nu rulați!) `sudo chown another_user some_file`

Observați că pentru a schimba proprietarul unui fișier, trebuie să avem privilegii de superutilizator. Pentru a face acest lucru, exemplul nostru a folosit comanda sudo pentru a executa chown. Aceasta comandă funcționează în același mod pe directoare ca și pe fișiere.

5.1.5. Schimbarea grupului de proprietari

Grupul de proprietari al unui fișier sau director poate fi schimbat cu chgrp. Această comandă este folosită astfel:

(nu rulati!) `chgrp new_group some_file`

În exemplul de mai sus, am schimbat proprietatea grupului pentru some_file din grupul anterior în „new_group”. Trebuie să fim proprietarul fișierului sau directorului pentru a efectua un chgrp.

5.2. Controlul proceselor

Anterioar, ne-am uitat la unele dintre implicațiile pe care Ubuntu le are fiind un sistem de operare cu mai mulți utilizatori. Acum, vom examina natura multitasking a sistemului și modul în care este controlat cu interfața liniei de comandă.

Ca și în cazul oricărui sistem de operare multitasking, Ubuntu execută mai multe procese simultan. Ei bine, cel putin aparent simultan. De fapt, un singur nucleu de procesor

poate executa doar un proces la un moment dat, dar kernelul sistemului reușește să dea fiecărui proces rândul său la procesor și fiecare pare să ruleze în același timp.

Există mai multe comenzi care sunt folosite pentru a controla procesele. Sunt:

ps - listează procesele care rulează pe sistem

kill - trimite un semnal către unul sau mai multe procese (de obicei pentru a „ucide" un proces)

jobs - o modalitate alternativă de a vă enumera propriile procese

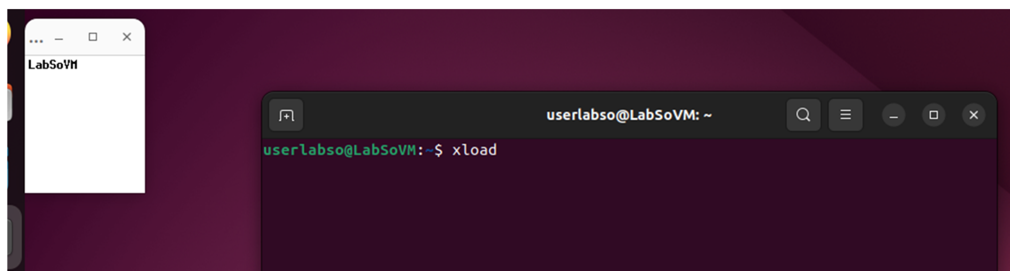
bg - pune un proces în fundal

fg - pune un proces în prim plan

Un exemplu practic

Deși poate părea că acest subiect este destul de obscur, poate fi foarte practic pentru utilizatorul obișnuit care lucrează în mare parte cu interfața grafică cu utilizatorul. Deși s-ar putea să nu fie evident, majoritatea (dacă nu toate) programele grafice pot fi lansate din linia de comandă. Iată un exemplu: există un mic program furnizat cu sistemul X Window numit xload care afișează un grafic care reprezintă încărcarea sistemului. Putem executa acest program tastând următoarele:

```
xload
```



Observați că apare o fereastra mică xload și începe să afișeze graficul de încărcare a sistemului. Daca xload nu este disponibil, încercați în schimb gedit. Observați, de asemenea, că promptul nostru nu a reapărut după lansarea programului. Shell-ul așteaptă ca programul să se termine înainte ca controlul să revină. Dacă închidem fereastra xload, programul xload se încheie și promptul revine.

5.2.1. Punerea unui program în fundal

Acum, pentru a ne ușura puțin viața, vom lansa din nou programul xload, dar de data aceasta îl vom pune în fundal, astfel încât promptul să revină. Pentru a face acest lucru, executăm xload astfel:

```
xload &
```

În acest caz, promptul a revenit deoarece procesul a fost pus în fundal.

Acum imaginați-vă că am uitat să folosim simbolul „&” pentru a pune programul în fundal. Încă mai există speranță. Putem tasta Ctrl-z și procesul va fi suspendat. Putem verifica acest lucru văzând că fereastra programului este înghețată. Procesul încă există, dar este inactiv. Pentru a relua procesul în fundal, tastați comanda bg (prescurtare pentru background). Iată un exemplu:

xload (urmat de tastarea Ctrl + z)

bg

5.2.2. Listarea proceselor în derulare

Acum că avem un proces în fundal, ar fi util să afișam o listă a proceselor pe care le-am lansat. Pentru a face acest lucru, putem folosi fie comanda jobs, fie comanda ps.

jobs

ps

5.2.3. Terminarea unui proces

Să presupunem că avem un program care nu răspunde; cum scapam de el? Folosim comanda kill, desigur. Să încercăm asta pe xload. În primul rând, trebuie să identificăm procesul pe care vrem să-l omorâm. Putem folosi fie jobs, fie ps, pentru a face asta. Dacă folosim jobs, vom primi înapoi un număr de job. Cu ps, ni se dă un *ID de proces* (PID). O vom face în ambele moduri:

kill %1

xload &

ps

kill 2745 (înlocuiți cu propriul PID afișat de comanda ps)

```
userlabso@LabSoVM:~$ jobs
[1]+  Running                 xload &
userlabso@LabSoVM:~$ ps
    PID TTY          TIME CMD
   2593 pts/0    00:00:00 bash
   2736 pts/0    00:00:00 xload
   2738 pts/0    00:00:00 ps
userlabso@LabSoVM:~$ kill %1
userlabso@LabSoVM:~$ xload &
[2] 2745
[1]   Terminated              xload
userlabso@LabSoVM:~$ ps
    PID TTY          TIME CMD
   2593 pts/0    00:00:00 bash
   2745 pts/0    00:00:00 xload
   2746 pts/0    00:00:00 ps
userlabso@LabSoVM:~$ kill 2745
userlabso@LabSoVM:~$
[2]+  Terminated              xload
userlabso@LabSoVM:~$
```

Un pic mai mult despre kill

În timp ce comanda kill este folosită pentru a „ucide" procesele, scopul său real este de a trimite semnale către procese. De cele mai multe ori semnalul este destinat să spună procesului să dispară, dar există mai mult decât atât. Programele (dacă sunt scrise corect) ascultă semnalele de la sistemul de operare și răspund la acestea, cel mai adesea pentru a permite o metodă grațioasă de terminare. De exemplu, un editor de text ar putea asculta orice semnal care indică faptul că utilizatorul se deconectează sau că computerul se închide. Când primește acest semnal, ar putea salva lucrările în curs înainte de a ieși. Comanda kill poate trimite o varietate de semnale către procese. Tastând:

kill -l

se va tipări o listă cu semnalele pe care le acceptă. Multe sunt destul de obscure, dar multe sunt la îndemână de știut:

Nr. Semnal	Nume	Descriere
1	SIGHUP	Semnal de închidere. Programele pot asculta acest semnal și pot acționa după el. Acest semnal este trimis proceselor care rulează într-un terminal atunci când închideți terminalul.
2	SIGINT	Semnal de întrerupere. Acest semnal este dat proceselor pentru a le întrerupe. Programele pot procesa acest semnal și pot acționa după el. De asemenea, putem emite acest semnal direct tastând Ctrl+c în fereastra terminalului în care rulează programul.
15	SIGTERM	Semnal de terminare. Acest semnal este dat proceselor pentru a le termina. Din nou, programele pot procesa acest semnal și pot acționa după el. Acesta este semnalul implicit trimis de comanda kill dacă nu este specificat niciun semnal.
9	SIGKILL	Semnal de ucidere. Acest semnal determină terminarea imediată a procesului de către kernelul Ubuntu. Programele nu pot asculta acest semnal.

Acum să presupunem că avem un program care este agățat fără speranță și vrem să scăpăm de el. Iată ce facem:

Utilizați comanda ps pentru a obține ID-ul procesului (PID) pe care vrem să-l încheiem.

Lansați o comandă kill pentru acel PID.

Dacă procesul refuză să se termine (adică ignoră semnalul), trimiteți semnale din ce în ce mai dure până când se încheie.

```
userlabso@LabSoVM:~$ ps x | grep bad_program
   2770 pts/0    S+     0:00 grep --color=auto bad_program
userlabso@LabSoVM:~$ kill -SIGTERM 2770
bash: kill: (2770) - No such process
userlabso@LabSoVM:~$ kill -SIGKILL 2770
bash: kill: (2770) - No such process
userlabso@LabSoVM:~$
```

În exemplul de mai sus am folosit comanda ps cu opțiunea x pentru a lista toate procesele noastre (chiar și cele care nu sunt lansate de pe terminalul curent). În plus, am transmis ieșirea comenzii ps în grep pentru a lista doar programul care ne interesează. Apoi, am folosit kill pentru a emite un semnal SIGTERM programului deranjant. În practică, este mai obișnuit să o faceți în felul următor, deoarece semnalul implicit trimis de kill este SIGTERM și kill poate folosi și numărul semnalului în loc de numele semnalului:

```
userlabso@LabSoVM:~$ kill 2770
userlabso@LabSoVM:~$ kill -9 2770
```

A doua comandă este forțarea cu semnalul SIGKILL, dacă procesul nu se încheie.

6. SCRIPTURI SHELL ÎN SISTEME LINUX – EDITOARE DE TEXT, SCRIPTURILE IMPLICITE ȘI CONSTRUIREA UNEI APLICAȚII

Cu miile de comenzi disponibile pentru utilizatorul liniei de comandă, cum le putem aminti pe toate? Răspunsul este că nu. Adevărata putere a computerului este capacitatea lui de a face treaba pentru noi. Pentru a face asta, folosim puterea shell-ului pentru a automatiza lucrurile. Scriem scripturi shell.

Ce sunt scripturile Shell?

În cei mai simpli termeni, un script shell este un fișier care conține o serie de comenzi. Shell-ul citește acest fișier și execută comenzile ca și cum ar fi fost introduse direct pe linia de comandă.

Shell-ul este oarecum unic, prin faptul că este atât o interfață puternică de linie de comandă pentru sistem, cât și un interpret pentru limbajul de scripting. După cum vom vedea, majoritatea lucrurilor care pot fi făcute pe linia de comandă pot fi făcute în scripturi, iar majoritatea lucrurilor care pot fi făcute în scripturi pot fi făcute pe linia de comandă.

Am acoperit deja multe caracteristici shell, dar ne-am concentrat pe acele caracteristici utilizate cel mai des direct pe linia de comandă. Shell-ul oferă, de asemenea, un set de caracteristici utilizate de obicei (dar nu întotdeauna) la scrierea programelor. Scripturile deblochează puterea mașinii noastre Linux. Deci hai sa ne distram puțin!

Pentru a implementa cu succes un script shell, trebuie să facem trei lucruri:

- scrierea unui script;
- atribuirea de permisiuni de execuție;
- localizat undeva unde poate fi găsit de shell.

6.1. Scrierea unui Script (editoare de text)

Un script shell este un fișier care conține text in format ASCII. Pentru a crea un script shell, folosim un editor de text. Un editor de text este un program, ca un procesor de text, care citește și scrie fișiere text ASCII. Există multe, multe editoare de text disponibile pentru sistemele Linux, atât pentru linia de comandă, cât și în medii GUI (graphical user interface). Iată o listă cu unele dintre cele mai comune:

Nume	Descriere	Interfața
`vi`, `vim`	Bunicul editoarelor de text Unix, vi, este notoriu pentru interfața sa de utilizator obtuză. Din fericire, vi este puternic, ușor și rapid. Învățarea vi este un rit de trecere, deoarece este disponibil universal pe sisteme Unix. Pe majoritatea distribuțiilor Linux, o versiune îmbunătățită a vi numită vim este furnizată în locul lui vi. vim este un editor remarcabil și merită să vă faceți timp pentru a-l învăța.	linia de comandă

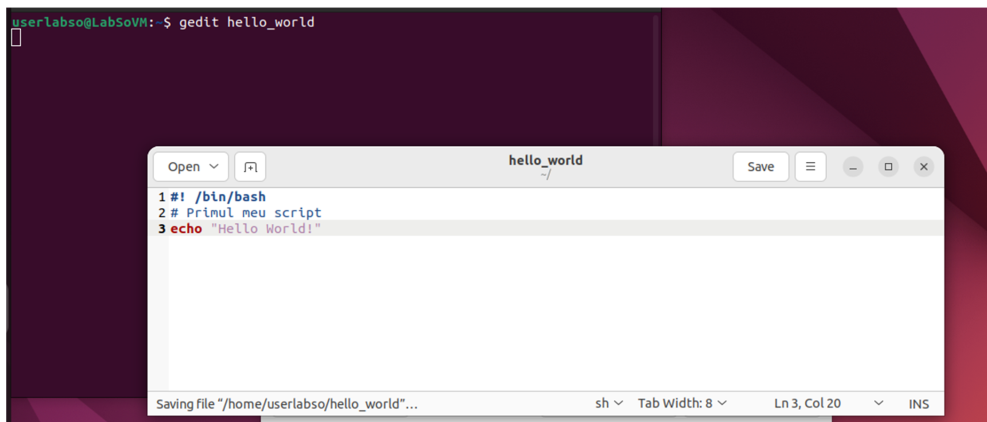
`emacs`	Adevăratul gigant în lumea editoarelor de text este emacs scris inițial de Richard Stallman. emacs conține (sau poate fi făcut să conțină) fiecare caracteristică concepută vreodată pentru un editor de text. Trebuie remarcat faptul că fanii vi și emacs luptă în „războaie religioase" amare pentru care este mai bun.	linia de comandă
`nano`	nano este o clonă gratuită a editorului de text furnizat cu programul de e-mail pine. nano este foarte ușor de utilizat, dar are foarte puține funcții în comparație cu vim și emacs. nano este recomandat pentru utilizatorii debutanți care au nevoie de un editor de text in linia de comandă.	linia de comandă
`gedit`	gedit este editorul furnizat cu mediul desktop GNOME. gedit este ușor de utilizat și conține suficiente funcții pentru a fi un editor bun la nivel de începător.	grafica
`kwrite`	kwrite este „editorul avansat" furnizat cu KDE. Are evidențiere de sintaxă, o caracteristică utilă pentru programatori și scrierea de scripturi.	grafica

Să pornim editorul nostru de text și să introducem primul nostru script după cum urmează:

```
#!/bin/bash
# Primul meu script
echo „Hello world!”
```

Dacă ați reușit să faceți copy/paste textul în editorul de text, bravo! ;) Dacă nu, aceștia sunt pașii ce trebuie urmați:

- *gedit hello_world*
- copy/paste pentru text în interfața grafică a gedit
- salvați noul fișier



Acesta este un program tradițional „Hello World”. Formele acestui program apar în aproape fiecare carte introductivă de programare. Vom salva fișierul cu un nume descriptiv, cum ar fi hello_world.

Prima linie a scenariului este importantă. Este o comandă specială, numită shebang, dată sistemului care indică ce program urmează să fie utilizat pentru a interpreta scriptul. În acest caz, /bin/bash. Alte limbaje de scripting, cum ar fi Perl, awk, tcl, Tk și python folosesc, de asemenea, acest mecanism.

A doua linie este un comentariu. Tot ceea ce apare după simbolul „#” este ignorat de bash. Pe măsură ce scripturile noastre devin mai mari și mai complicate, comentariile devin vitale. Ele sunt folosite de programatori pentru a explica ce se întâmplă, astfel încât alții să-și dea seama. Ultima linie este comanda echo. Această comandă își imprimă pur și simplu argumentele pe afișaj (dupa cum deja stim).

Setarea permisiunilor

Următorul lucru pe care trebuie să-l facem este să dăm permisiuni shell-ului să execute scriptul nostru. Acest lucru se face cu comanda chmod după cum urmează:

```
chmod 755 hello_world
```

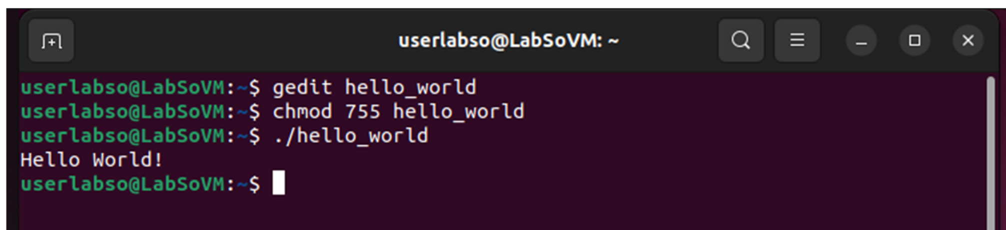
„755” ne va da permisiunea de citire, scriere și executare. Toți ceilalți vor primi doar permisiunea de citire și executare. Pentru a face scriptul privat, (adică doar noi putem citi și executa), utilizați „700” în schimb.

Adaugarea acestuia la calea noastră (PATH)

În acest moment, scriptul nostru va rula. Încearcă asta:

```
./hello_world
```

Ar trebui să vedem „Hello World!” afișat.



Înainte de a merge mai departe, trebuie să vorbim despre căi. Când introducem numele unei comenzi, sistemul nu caută în întregul computer pentru a găsi unde se află programul. Asta ar dura prea mult timp. Vedem că de obicei nu trebuie să specificăm un nume complet de cale pentru programul pe care vrem să-l rulăm, shell-ul pare să știe. Iată cum: shell-ul menține o listă de directoare în care sunt păstrate fișierele executabile (programe) și caută doar directoarele din acea listă. Dacă nu găsește programul după ce a căutat fiecare director din listă, va emite celebrul mesaj de eroare de comandă negăsită.

Această listă de directoare se numește PATH. Putem vizualiza lista de directoare cu următoarea comandă:

```
echo $PATH
```

Aceasta va returna o listă de directoare separate prin două puncte, care vor fi căutate dacă nu este dat un nume de cale specifică atunci când este introdusă o comandă. În prima noastră încercare de a executa noul nostru script, am specificat o cale ("./") către fișier.

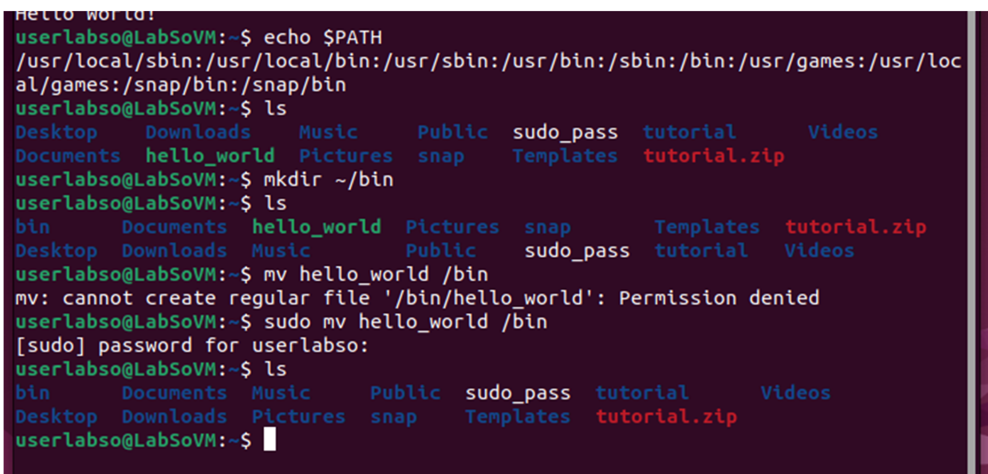
Putem adăuga directoare în calea noastră cu următoarea comandă, unde folder este numele directorului pe care vrem să-l adăugăm:

```
export PATH=$PATH:folder
```

O modalitate mai bună ar fi să editați fișierul nostru .bash_profile pentru a include comanda de mai sus. În acest fel, s-ar face automat de fiecare dată când ne conectăm.

Majoritatea distribuțiilor Linux încurajează practica în care fiecare utilizator are un director specific pentru programele pe care le folosește personal. Acest director se numește bin și este un subdirector al directorului nostru home. Dacă nu avem deja unul, îl putem crea cu următoarea comandă:

```
mkdir ~/bin
```



Dacă mutăm sau copiem scriptul nostru în noul nostru director bin, vom fi gata. Acum trebuie doar să tastam:

```
hello_world
```

iar scriptul nostru va rula. Majoritatea distribuțiilor vor avea directorul ~/bin deja în PATH, dar pe unele distribuții, în special Ubuntu (și alte distribuții bazate pe Debian), ar putea fi nevoie să repornim sesiunea noastră de terminal înainte ca directorul bin nou creat să fie adăugat la PATH.

```
userlabso@LabSoVM:~$ ls
bin      Documents  Music     Public  sudo_pass  tutorial      Videos
Desktop  Downloads  Pictures  snap    Templates  tutorial.zip
userlabso@LabSoVM:~$ hello_world
Hello World!
userlabso@LabSoVM:~$
```

6.2. Editarea scripturilor pe care le avem deja

Înainte de a începe să scriem scripturi mai complexe, vom arunca o privire asupra unor scripturi pe care le avem deja. Aceste scripturi au fost introduse în directorul nostru principal când a fost creat contul și sunt folosite pentru a configura comportamentul sesiunilor noastre pe computer. Putem edita aceste scripturi pentru a schimba lucrurile.

În această lecție, ne vom uita la câteva dintre aceste scripturi și vom afla câteva concepte noi importante despre shell.

În timpul sesiunii noastre shell, sistemul reține o serie de fapte despre lume în memoria sa. Această informație se numește mediu (*environment*). Mediul conține lucruri precum calea noastră, numele nostru de utilizator și multe altele. Putem examina o listă completă a ceea ce este în mediu cu comanda set.

Două tipuri de comenzi sunt adesea conținute în mediu. Acestea sunt aliasuri și funcții shell.

Cum este stabilit Mediul?

Când ne conectăm la sistem, programul bash pornește și citește o serie de scripturi de configurare numite fișiere de pornire. Acestea definesc mediul implicit folosit de toți utilizatorii. Acesta este urmat de mai multe fișiere de pornire în directorul nostru de pornire care definesc mediul nostru personal. Secvența exactă depinde de tipul de sesiune shell care este începută. Există două tipuri: o *sesiune shell login* și o *sesiune shell non-login*. O sesiune shell login este una în care ni se solicită numele de utilizator și parola; când începem o sesiune de consolă virtuală, de exemplu. O sesiune shell non-login are loc de obicei atunci când lansăm o sesiune de terminal în GUI.

Shell-urile login citesc unul sau mai multe fișiere de pornire, așa cum se arată mai jos:

Fișier	Conținut
`/etc/profile`	Un script de configurare global care se aplică tuturor utilizatorilor.
`~/.bash_profile`	Fișierul personal de pornire al unui utilizator. Poate fi folosit pentru a extinde sau a suprascrie setările din scriptul de configurare globală.
`~/.bash_login`	Dacă ~/.bash_profile nu este găsit, bash încearcă să citească acest script.

`~/.profile`	Dacă nu se găsesc nici ~/.bash_profile, nici ~/.bash_login, bash încearcă să citească acest fișier. Aceasta este implicit în distribuțiile bazate pe Debian, cum ar fi Ubuntu.

Sesiunile shell non-login citesc următoarele fișiere de pornire:

Fisier	Continut
`/etc/bash.bashrc`	Un script de configurare global care se aplică tuturor utilizatorilor.
`~/.bashrc`	Fișierul personal de pornire al unui utilizator. Poate fi folosit pentru a extinde sau a suprascrie setările din scriptul de configurare globală.

Pe lângă citirea fișierelor de pornire de mai sus, shell-urile non-login moștenesc și mediul din procesul lor părinte, de obicei un shell de conectare.

Aruncați o privire la sistemul dvs. și vedeți pe care dintre aceste fișiere de pornire le aveți. Amintiți-vă - deoarece majoritatea numelor de fișiere enumerate mai sus încep cu un punct (însemnând că sunt ascunse), va trebui să utilizați opțiunea „-a” când utilizați ls.

```
userlabso@LabSoVM:~$ ls -a
.              .cache     .lesshst  Public                      .thunderbird
..             .config    .local    snap                        tutorial
.bash_history  Desktop    .mozilla  .ssh                        tutorial.zip
.bash_logout   Documents  Music     .sudo_as_admin_successful   Videos
.bashrc        Downloads  Pictures  sudo_pass
bin            .gnupg     .profile  Templates
userlabso@LabSoVM:~$
```

Fișierul ~/.bashrc este probabil cel mai important fișier de pornire din punctul de vedere al utilizatorului obișnuit, deoarece este aproape întotdeauna citit. Shell-urile non-login îl citesc implicit și majoritatea fișierelor de pornire pentru shell-uri de autentificare sunt scrise în așa fel încât să citească și fișierul ~/.bashrc.

Dacă aruncăm o privire în interiorul unui .profile tipic (acesta este luat dintr-un sistem Ubuntu), arată cam așa:

```
Open    .profile /home/userlabso    Save
1 # ~/.profile: executed by the command interpreter for login shells.
2 # This file is not read by bash(1), if ~/.bash_profile or ~/.bash_login
3 # exists.
4 # see /usr/share/doc/bash/examples/startup-files for examples.
5 # the files are located in the bash-doc package.
6
7 # the default umask is set in /etc/profile; for setting the umask
8 # for ssh logins, install and configure the libpam-umask package.
9 #umask 022
10
11 # if running bash
12 if [ -n "$BASH_VERSION" ]; then
13     # include .bashrc if it exists
14     if [ -f "$HOME/.bashrc" ]; then
15         . "$HOME/.bashrc"
16     fi
17 fi
18
19 # set PATH so it includes user's private bin if it exists
20 if [ -d "$HOME/bin" ] ; then
21     PATH="$HOME/bin:$PATH"
22 fi
23
24 # set PATH so it includes user's private bin if it exists
25 if [ -d "$HOME/.local/bin" ] ; then
26     PATH="$HOME/.local/bin:$PATH"
27 fi
sh   Tab Width: 8   Ln 22, Col 3   INS
```

Liniile care încep cu „#” sunt comentarii și nu sunt citite de shell. Acestea sunt acolo pentru lizibilitatea umană. Primul lucru interesant apare pe a 14-a linie, cu următorul cod:

```
if [ -f  “$HOME/.bashrc” ]; then

  . “$HOME/.bashrc”

fi
```

Aceasta se numește o comandă if compusă, pe care o vom acoperi pe deplin mai tarziu, dar deocamdată vom traduce:

Dacă fișierul „~/.bashrc” există, atunci citiți fișierul „~/.bashrc”.

Putem vedea că acest fragment de cod este modul în care un shell de autentificare obține conținutul .bashrc. Următorul lucru pe care îl face în fișierul nostru de pornire este să setați variabila PATH pentru a adăuga directorul ~/bin la cale.

Aliasuri

Un alias este o modalitate ușoară de a crea o nouă comandă care acționează ca o abreviere pentru una mai lungă. Are următoarea sintaxă:

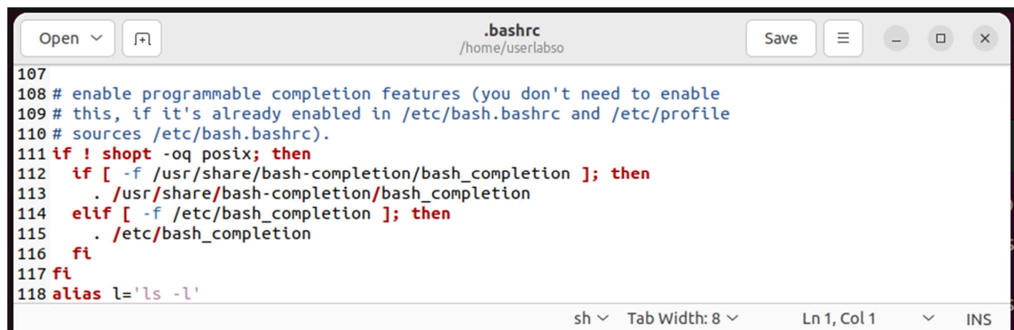
```
alias nume=valoare
```

unde nume este numele noii comenzi și valoare este textul care trebuie executat ori de câte ori numele este introdus pe linia de comandă.

Să creăm un alias numit „l” și să facem din acesta o abreviere pentru comanda „ls -l”. Ne vom muta în directorul nostru principal și, folosind editorul nostru de text favorit, deschidem fișierul .bashrc și vom adăuga această linie la sfârșitul fișierului:

```
userlabso@LabSoVM:~$ sudo gedit .bashrc
```

```
alias l='ls -l'
```



Adăugând comanda alias în fișier, am creat o nouă comandă numită „l” care va executa „ls -l”. Pentru a încerca noua noastră comandă, închideți sesiunea terminalului și începeți una nouă. Aceasta va reîncărca fișierul .bashrc. Folosind această tehnică, putem crea orice număr de comenzi personalizate pentru noi înșine. Iată încă una de încercat:

```
alias azi='date +"%A, %-d %B %Y"'
```

Acest alias creează o nouă comandă numită „azi” care va afișa data de astăzi cu o formatare plăcută.

Ne putem crea aliasurile direct la promptul de comandă, dar acestea vor rămâne în vigoare doar în timpul sesiunii curente de shell. De exemplu:

```
userlabso@LabSoVM:~$ alias l='ls -l'
userlabso@LabSoVM:~$ l
total 52
drwxrwxr-x 2 userlabso userlabso 4096 nov  1 13:06 bin
drwxr-xr-x 2 userlabso userlabso 4096 oct 13 13:17 Desktop
drwxr-xr-x 2 userlabso userlabso 4096 oct 13 13:17 Documents
drwxr-xr-x 2 userlabso userlabso 4096 oct 13 13:17 Downloads
drwxr-xr-x 2 userlabso userlabso 4096 oct 13 13:17 Music
drwxr-xr-x 2 userlabso userlabso 4096 oct 13 13:17 Pictures
drwxr-xr-x 2 userlabso userlabso 4096 oct 13 13:17 Public
drwx------ 4 userlabso userlabso 4096 oct 23 10:49 snap
-rw-rw-r-- 1 userlabso userlabso   17 oct 27 14:30 sudo_pass
drwxr-xr-x 2 userlabso userlabso 4096 oct 13 13:17 Templates
drwxrwxr-x 7 userlabso userlabso 4096 oct 23 10:44 tutorial
-rw-rw-r-- 1 userlabso userlabso 1566 oct 23 10:49 tutorial.zip
drwxr-xr-x 2 userlabso userlabso 4096 oct 13 13:17 Videos
userlabso@LabSoVM:~$
```

Funcții Shell

Aliasurile sunt bune pentru comenzi foarte simple, dar pentru a crea ceva mai complex, avem nevoie de funcții shell. Funcțiile Shell pot fi gândite ca „scripturi în cadrul scripturilor” sau sub-scripte mici. Să încercăm unul. Deschideți .bashrc din nou cu editorul de text și înlocuiți aliasul pentru „azi” cu următorul:

```
azi() {
     echo -n "Data de azi este: "
     date +"%A, %-d %B %Y"
}
```

Ca și în cazul aliasului, putem introduce funcții shell direct la promptul de comandă.

```
userlabso@LabSoVM:~$ azi () {
> echo -n "Data de azi este: "
> date +"%A, %-d %B %Y"
> }
userlabso@LabSoVM:~$ azi
Data de azi este: miercuri, 1 noiembrie 2023
userlabso@LabSoVM:~$
```

Cu toate acestea, ca și alias, funcțiile shell definite direct pe linia de comandă durează doar atât timp cât sesiunea shell curentă este pornită.

6.3. Construirea unei aplicații

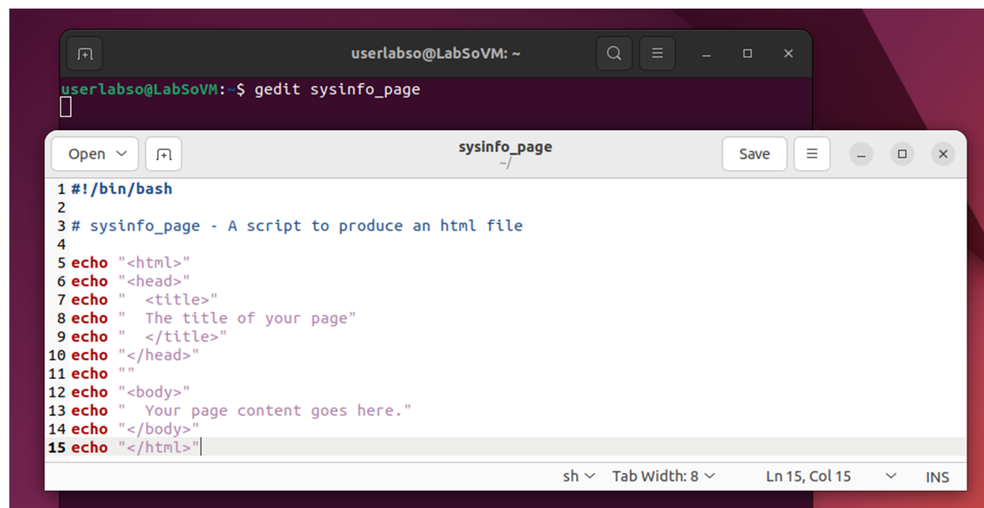
În continuare, vom construi o aplicație utilă. Această aplicație va produce un document HTML care conține informații despre sistemul nostru. Pe măsură ce ne construim scriptul, vom descoperi pas cu pas instrumentele necesare pentru a rezolva problema în cauză.

Scrierea unui fișier HTML cu un script

După cum știm (sau poate ca nu), un fișier HTML bine format conține următoarele:

```
<html>
<head>
    <title>
    The title of your page
    </title>
</head>

<body>
    Your page content goes here.
</body>
</html>
```

Acum, cu ceea ce știm deja, am putea scrie un script numit sysinfo_page pentru a produce conținutul de mai sus:



Acest script poate fi folosit după cum urmează:

```
./sysinfo_page > sysinfo_page.html
```

```
userlabso@LabSoVM:~$ ./sysinfo_page > sysinfo_page.html
userlabso@LabSoVM:~$
userlabso@LabSoVM:~$ ls
bin        Downloads  Public     sysinfo_page       tutorial
Desktop    Music      snap       sysinfo_page.html  tutorial.zip
Documents  Pictures   sudo_pass  Templates          Videos
userlabso@LabSoVM:~$
```

Se poate spune că cei mai buni programatori sunt și cei mai leneși. Ei scriu programe pentru a munci mai putin. La fel, atunci când programatorii inteligenți scriu programe, ei încearcă să tasteze cat mai putin.

Prima îmbunătățire a acestui script va fi înlocuirea utilizării repetate a comenzii echo cu o singură instanță, folosind mai eficient citatul:

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

echo "<html>
 <head>
   <title>
   The title of your page
   </title>
 </head>

 <body>
   Your page content goes here.
 </body>
 </html>"
```

Folosind ghilimele, este posibil să încorporam returnări în textul nostru astfel incat argumentul comenzii echo să acopere mai multe linii.

Deși aceasta este cu siguranță o îmbunătățire, are o limitare. Deoarece multe tipuri de marcare utilizate în HTML încorporează ghilimele în sine, utilizarea unui șir între ghilimele este puțin dificilă. Se poate folosi un șir între ghilimele, dar fiecare ghilimea încorporate va trebui să fie eliminate cu un caracter backslash.

Pentru a evita tastarea suplimentară, trebuie să căutăm o modalitate mai bună de a ne produce textul. Din fericire, shell-ul oferă unul. Se numește “here script”.

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

cat << _EOF_
```

```
<html>
<head>
    <title>
    The title of your page
    </title>
</head>

<body>
    Your page content goes here.
</body>
</html>
_EOF_
```

Un here script (numit uneori și here document) este o formă suplimentară de redirecționare I/O. Oferă o modalitate de a include conținut care va fi dat intrării standard a unei comenzi. În cazul scriptului de mai sus, intrarea standard a comenzii cat a primit un flux de text din scriptul nostru.

Un here script este construit astfel:

```
comanda << simbol
conținut pentru a fi utilizat ca intrare standard a comenzii
simbol
```

simbolul poate fi orice șir de caractere. „_EOF_" (EOF este prescurtarea pentru „End Of File") este tradițional, dar putem folosi orice, atâta timp cât nu intră în conflict cu un cuvânt rezervat bash. Indicatorul care termină scriptul aici trebuie să se potrivească exact cu cel care îl pornește, altfel restul scriptului nostru va fi interpretat ca o intrare mai standard la comandă, ceea ce poate duce la unele eșecuri de script cu adevărat interesante.

Există un truc suplimentar care poate fi folosit cu un here script. Adesea, am putea dori să indentăm porțiunea de conținut a scriptului de aici pentru a îmbunătăți lizibilitatea scriptului. Putem face acest lucru dacă schimbăm scriptul după cum urmează:

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

cat <<- _EOF_
    <html>
    <head>
        <title>
        The title of your page
        </title>
```

```
    </head>

    <body>
        Your page content goes here.
    </body>
    </html>
_EOF_
```

Schimbarea „<<” în „<<-” face ca bash să ignore tab-urile principale (dar nu spațiile) în scriptul here. Ieșirea de la comanda cat nu va conține niciunul dintre caracterele tab principale. Această tehnică este puțin problematică, deoarece multe editoare de text sunt configurate (și este de dorit) să folosească secvențe de spații mai degrabă decât caractere tab.

OK, hai să ne facem pagina. Vom edita pagina noastră pentru a o face să spună ceva:

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

cat <<- _EOF_
    <html>
    <head>
        <title>
        My System Information
        </title>
    </head>

    <body>
    <h1>My System Information</h1>
    </body>
    </html>
_EOF_
```

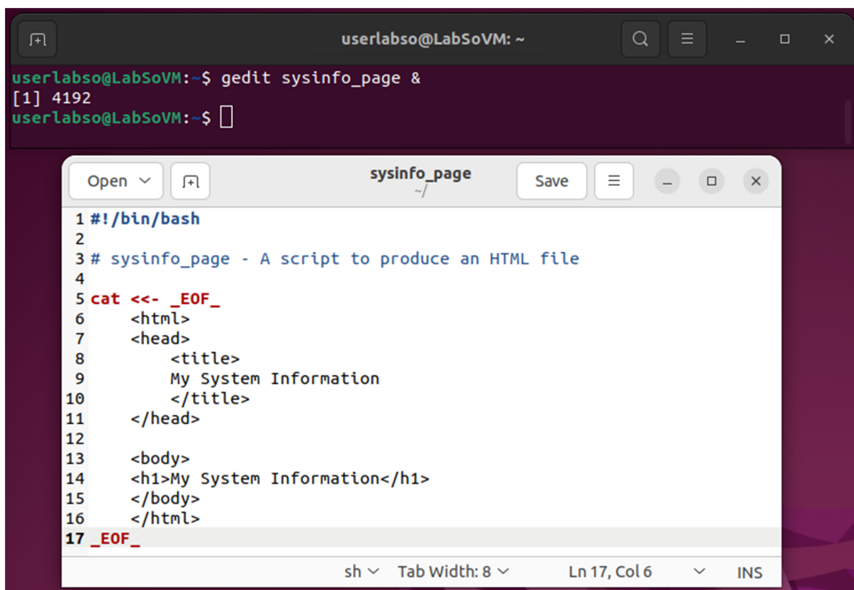
Data viitoare vom face ca scriptul nostru să producă câteva informații reale despre sistem.

7. SCRIPTURI SHELL ÎN SISTEME LINUX - VARIABILE, CONSTANTE ȘI FUNCȚII

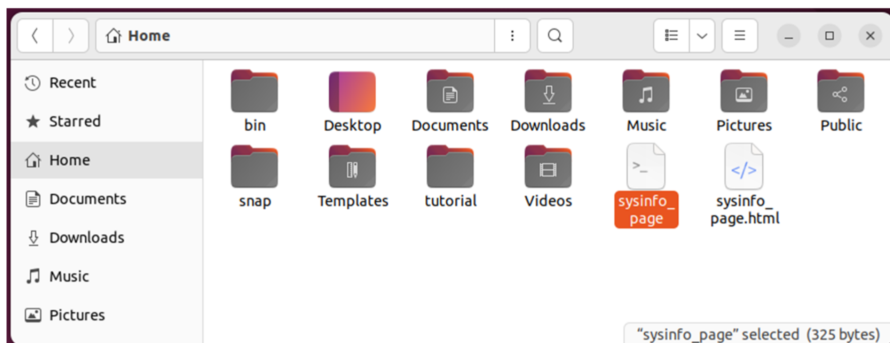
7.1. Variabile

Acum că avem scriptul de lucrarea anterioară (mai sus), hai să-l îmbunătățim. În primul rând, haideți să deschidem scriptul într-un editor de text (de exemplu gedit), și, pentru a ne face viața mai ușoară, haideți să îl deschidem în fundal.

```
gedit sysinfo_page &
```



Bineînțeles, putem chiar să îl deschidem și din interfața grafică – deși asta e mai plictisitor :)



```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

cat <<- _EOF_
    <html>
    <head>
        <title>
        My System Information
        </title>
    </head>

    <body>
    <h1>My System Information</h1>
    </body>
    </html>
_EOF_
```

În continuare, vom face câteva modificări pentru că vrem să fim leneși. În scriptul de mai sus, vedem că se repetă expresia „My System Information”. Este o tastare irosită (și muncă suplimentară!), așa că o vom îmbunătăți astfel:

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

title="My System Information"

cat <<- _EOF_
    <html>
    <head>
        <title>
        $title
        </title>
    </head>

    <body>
    <h1>$title</h1>
    </body>
    </html>
_EOF_
```

Am adăugat o linie la începutul scriptului și am înlocuit cele două apariții ale expresiei „My System Information” cu $title.

Ceea ce am făcut este să introducem un concept fundamental care apare în fiecare limbaj de programare, variabile. Variabilele sunt zone de memorie care pot fi folosite pentru a stoca informații și sunt denumite printr-un nume. În cazul scriptului nostru, am creat o variabilă numită „title” și am plasat expresia „ My System Information” în memorie. În scriptul de aici care conține HTML-ul nostru, folosim „$title” pentru a spune shell-ului să efectueze extinderea parametrilor și să înlocuiască numele variabilei cu conținutul variabilei.

Ori de câte ori shell-ul vede un cuvânt care începe cu un „$”, încearcă să afle ce a fost atribuit variabilei și îl înlocuiește.

Pentru a crea o variabilă, puneți în script o linie care conține numele variabilei urmat imediat de un semn egal ("="). Nu sunt permise spații. După semnul egal, atribuiți informațiile de stocat.

De unde provin numele de variabile? Usor, le inventăm. Există totusi câteva reguli.

- numele trebuie să înceapă cu o literă.
- un nume nu trebuie să conțină spații încorporate (folosiți în schimb underscore).
- semnele de punctuație nu sunt permise.

Cum ne ajuta asta? Adăugarea variabilei title ne-a făcut viața mai ușoară în două moduri. În primul rând, a redus cantitatea de tastare pe care trebuia să o facem. În al doilea rând și mai important, a făcut scriptul nostru mai ușor de întreținut.

Pe măsură ce scriem din ce în ce mai multe scripturi (sau facem orice alt fel de programare), vom vedea că programele sunt rareori terminate. Ele sunt modificate și îmbunătățite de creatorii lor și/sau de alții. La urma urmei, la asta se referă dezvoltarea open source. Să presupunem că am vrut să schimbăm expresia „My System Information” în „VirtualBox System Information”. În versiunea anterioară a scenariului, ar fi trebuit să schimbăm acest lucru în două locații. În noua versiune cu variabila title, trebuie doar să o schimbăm într-un singur loc. Deoarece scriptul nostru este atât de mic, aceasta ar putea părea o chestiune banală, dar pe măsură ce scripturile devin mai mari și mai complicate, devine foarte important.

7.2. Variabile de mediu

Când începem sesiunea shell, unele variabile sunt deja setate de fișierele de pornire pe care le-am analizat data trecută. Pentru a vedea toate variabilele care se află în mediu, utilizați comanda printenv.

```
printenv
```

```
userlabso@LabSoVM:~$ printenv
SHELL=/bin/bash
SESSION_MANAGER=local/LabSoVM:@/tmp/.ICE-unix/1597,unix/LabSoVM:/tmp/.ICE-unix/1597
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
SSH_AGENT_LAUNCHER=gnome-keyring
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LC_ADDRESS=ro_RO.UTF-8
GNOME_SHELL_SESSION_MODE=ubuntu
LC_NAME=ro_RO.UTF-8
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
LC_MONETARY=ro_RO.UTF-8
GTK_MODULES=gail:atk-bridge
PWD=/home/userlabso
LOGNAME=userlabso
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=wayland
SYSTEMD_EXEC_PID=1623
XAUTHORITY=/run/user/1000/.mutter-Xwaylandauth.34B0D2
HOME=/home/userlabso
USERNAME=userlabso
IM_CONFIG_PHASE=1
LC_PAPER=ro_RO.UTF-8
LANG=en_US.UTF-8
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*
.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;
31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.
alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.b
mp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpe
g=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.webp=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=
01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:
*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
XDG_CURRENT_DESKTOP=ubuntu:GNOME
VTE_VERSION=6800
WAYLAND_DISPLAY=wayland-0
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/6966407c_593d_4b9c_9af9_b0f9eafa098b
GNOME_SETUP_DISPLAY=:1
LESSCLOSE=/usr/bin/lesspipe %s %s
XDG_SESSION_CLASS=user
TERM=xterm-256color
LC_IDENTIFICATION=ro_RO.UTF-8
LESSOPEN=| /usr/bin/lesspipe %s
USER=userlabso
GNOME_TERMINAL_SERVICE=:1.101
DISPLAY=:0
SHLVL=1
LC_TELEPHONE=ro_RO.UTF-8
QT_IM_MODULE=ibus
LC_MEASUREMENT=ro_RO.UTF-8
XDG_RUNTIME_DIR=/run/user/1000
LC_TIME=ro_RO.UTF-8
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/local/share/:/usr/share/:/var/lib/snapd/desktop
PATH=/home/userlabso/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin
GDMSESSION=ubuntu
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
LC_NUMERIC=ro_RO.UTF-8
_=/usr/bin/printenv
userlabso@LabSoVM:~$
```

O variabilă din mediul nostru conține numele de gazdă pentru sistem. Vom adăuga această variabilă la scriptul nostru astfel:

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

title="System Information for"

cat <<- _EOF_
    <html>
    <head>
        <title>
        $title $HOSTNAME
        </title>
    </head>

    <body>
    <h1>$title $HOSTNAME</h1>
    </body>
    </html>
_EOF_
```

Acum scriptul nostru va include întotdeauna numele mașinii pe care rulăm. Rețineți că, prin convenție, numele variabilelor de mediu sunt cu majuscule.

7.3. Înlocuirea comenzilor și constantele

Anterior, am învățat cum să creăm variabile și să realizăm extinderea parametrilor cu acestea. Acum, vom extinde această idee pentru a arăta cum putem înlocui rezultatele din comenzi.

Când ne-am părăsit ultima dată scriptul, ar putea crea o pagină HTML care conținea câteva rânduri simple de text, inclusiv numele gazdă al mașinii pe care l-am obținut din variabila de mediu HOSTNAME. Acum, vom adăuga o ștampilă de timp pe pagină pentru a indica când a fost actualizată ultima dată, împreună cu utilizatorul care a făcut-o.

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

title="System Information for"

cat <<- _EOF_
    <html>
    <head>
        <title>
        $title $HOSTNAME
        </title>
    </head>

    <body>
    <h1>$title $HOSTNAME</h1>
    <p>Updated on $(date +"%x %r %Z") by $USER</p>
    </body>
    </html>
_EOF_
```

După cum putem vedea, o altă variabilă de mediu, USER, este folosită pentru a obține numele de utilizator. În plus, am folosit aceasta linie ciudata:

```
$(date +"%x %r %Z")
```

Caracterele „$( )" spun shell-ului „substituiți rezultatele comenzii incluse", o tehnică cunoscută sub numele de substituție de comandă. În scriptul nostru, dorim ca shell-ul să insereze rezultatele comenzii date +"%x %r %Z" care scoate data și ora curente. Comanda date are multe caracteristici și opțiuni de formatare. Pentru a le privi pe toate, încearcă asta (apoi, pentru a iesi din less, apasati tasta q):

```
date --help | less
```

Fiți conștienți de faptul că există o sintaxă alternativă mai veche pentru „$(command)" care folosește caracterul backtick „ ` ". Această formă mai veche este compatibilă cu shell-ul original Bourne (sh), dar utilizarea sa este descurajată în favoarea sintaxei moderne. Shell-ul bash acceptă pe deplin scripturile scrise pentru sh, deci următoarele forme sunt echivalente: $(command) = `command`

7.4. Atribuirea unui rezultat al unei comenzi către o variabilă

De asemenea, putem atribui rezultatele unei comenzi unei variabile:

```
right_now="$(date +"%x %r %Z")"
```

Putem avea variabilele si "nested" (plasate una în alta), în felul următor:

```
right_now="$(date +"%x %r %Z")"
time_stamp="Updated on $right_now by $USER"
```

Un sfat important: atunci când efectuați extinderi de parametri sau înlocuiri de comenzi, este o bună practică să le încadrați între ghilimele duble pentru a preveni împărțirea nedorită a cuvintelor în cazul în care rezultatul extinderii conține caractere spații albe.

7.5. Constante

După cum sugerează numele variabilei, conținutul unei variabile este supus modificării. Aceasta înseamnă că este de așteptat ca în timpul execuției scriptului nostru, o variabilă să aibă conținutul modificat de ceva ce face scriptul.

Pe de altă parte, pot exista valori care, odată setate, nu ar trebui să fie modificate niciodată. Acestea se numesc constante. Aceasta este o idee comună în programare. Majoritatea limbajelor de programare au facilități speciale pentru a suporta valori care nu au voie să se modifice. Bash are și această facilitate, dar este rar folosit. În schimb, dacă o valoare este destinată să fie o constantă, i se dă un nume cu majuscule pentru a reaminti programatorului că ar trebui să fie considerată o constantă, chiar dacă nu este aplicată.

Variabilele de mediu sunt de obicei considerate constante, deoarece sunt rareori modificate. La fel ca constantele, variabilelor de mediu li se dau nume cu majuscule prin convenție. În scripturile care urmează, vom folosi această convenție - nume majuscule pentru constante și nume minuscule pentru variabile.

Deci, aplicând tot ce știm, programul nostru arată astfel:

```
#!/bin/bash

# sysinfo_page - A script to produce an HTML file

title="System Information for $HOSTNAME"
RIGHT_NOW="$(date +"%x %r %Z")"
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

cat <<- _EOF_
    <html>
    <head>
```

```
        <title>
        $title
        </title>
    </head>

    <body>
    <h1>$title</h1>
    <p>$TIME_STAMP</p>
    </body>
    </html>
_EOF_
```

7.6. Funcții Shell

Pe măsură ce programele devin mai lungi și mai complexe, devin mai dificil de proiectat, codificat și întreținut. Ca și în cazul oricărui efort mare, este adesea util să împărțiți o singură sarcină mare într-o serie de sarcini mai mici.

În continuare, vom începe să împărțim scriptul nostru monolitic într-un număr de funcții separate.

Pentru a ne familiariza cu această idee, să luăm în considerare descrierea unei sarcini de zi cu zi - să mergem la supermarket pentru a cumpăra alimente. Imaginați-vă că trebuie să descriem sarcina asta unui om de pe Marte.

O descriere de nivel "macroscopic" ar putea arăta astfel:

1. Pleacă din casă
2. Conduceți la supermarket
3. Parcați mașina
4. Intră in supermarket
5. Cumpărați alimente
6. Conduceți spre casa
7. Parcați mașina
8. Intră în casă

Această descriere acoperă procesul general; cu toate acestea, o persoana de pe Marte va avea nevoie probabil de detalii suplimentare. De exemplu, sarcina secundară „Parcarea mașinii" ar putea fi descrisă după cum urmează:

1. Găsiți loc de parcare
2. Conduceți mașina în spațiu
3. Opriți motorul
4. Trage frâna de mana
5. Ieșiți din mașină
6. Încuie mașina

Desigur, sarcina „Opriți motorul” are o serie de pași, cum ar fi „decuplați contactul” și „scoateți cheia din contact” și așa mai departe.

Acest proces de identificare a pașilor de nivel superior și de dezvoltare a unor vederi din ce în ce mai detaliate ale acelor pași se numește design top-down. Această tehnică ne permite să împărțim sarcini complexe mari în multe sarcini mici și simple.

Pe măsură ce scriptul nostru continuă să crească, vom folosi designul top-down pentru a ne ajuta să ne planificăm și să codificăm scriptul.

Dacă ne uităm la sarcinile de nivel superior ale scriptului nostru, găsim următoarea listă:

1. Open page
2. Open head section
3. Write title
4. Close head section
5. Open body section
6. Write title
7. Write time stamp
8. Close body section
9. Close page

Toate aceste sarcini sunt implementate, dar dorim să adăugăm mai multe. Să inserăm câteva sarcini suplimentare după sarcina 7:

7. Write time stamp
8. Write system release info
9. Write up-time
10. Write drive space
11. Write home space
12. Close body section
13. Close page

Ar fi grozav dacă ar exista comenzi care să execute aceste sarcini suplimentare. Dacă ar exista, am putea folosi înlocuirea comenzilor pentru a le plasa în scriptul nostru astfel:

```
#!/bin/bash

# sysinfo_page - A script to produce a system information HTML
file

##### Constants

TITLE="System Information for $HOSTNAME"
RIGHT_NOW="$(date +"%x %r %Z")"
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

##### Main

cat <<- _EOF_
```

```
  <html>
  <head>
      <title>$TITLE</title>
  </head>

  <body>
      <h1>$TITLE</h1>
      <p>$TIME_STAMP</p>
      $(system_info)
      $(show_uptime)
      $(drive_space)
      $(home_space)
  </body>
  </html>
_EOF_
```

Deși nu există comenzi care să facă exact ceea ce avem nevoie, le putem crea folosind funcții shell.

După cum am văzut anterior, funcțiile shell acționează ca „programe mici în cadrul programelor” și ne permit să respectăm principiile de design top-down. Pentru a adăuga funcțiile shell la scriptul nostru, îl vom schimba astfel:

```
#!/bin/bash

# sysinfo_page - A script to produce an system information HTML
file

##### Constants

TITLE="System Information for $HOSTNAME"
RIGHT_NOW="$(date +"%x %r %Z")"
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

##### Functions

system_info()
{

}


show_uptime()
{

}


drive_space()
{

}
```

```
home_space()
{

}

##### Main

cat <<- _EOF_
  <html>
  <head>
      <title>$TITLE</title>
  </head>

  <body>
      <h1>$TITLE</h1>
      <p>$TIME_STAMP</p>
      $(system_info)
      $(show_uptime)
      $(drive_space)
      $(home_space)
  </body>
  </html>
_EOF_
```

Câteva puncte importante despre funcții: În primul rând, trebuie să apară înainte de a încerca să le folosim. În al doilea rând, corpul funcției (porțiunile funcției dintre caracterele { și }) trebuie să conțină cel puțin o comandă validă. Așa cum este scris, scriptul nu se va executa fără eroare, deoarece corpurile funcției sunt goale. Modul simplu de a remedia acest lucru este să plasați o instrucțiune return în fiecare corp de funcție. După ce facem acest lucru, scriptul nostru se va executa din nou cu succes.

7.7. Păstrarea funcțională a scripturilor

Când dezvoltăm un program, este adesea o practică bună să adăugați o cantitate mică de cod, să rulăm scriptul, să adăugați mai mult cod, să rulați scriptul și așa mai departe. În acest fel, dacă introducem o greșeală în cod, va fi mai ușor de găsit și corectat.

Pe măsură ce adăugăm funcții scriptului dvs., putem folosi și o tehnică numită stubbing pentru a urmări dezvoltarea logicii script-ului nostru. Stubbing funcționează astfel: imaginați-vă că vom crea o funcție numită „system_info” dar încă nu ne-am dat seama de toate detaliile codului său. În loc să oprim dezvoltarea scriptului până când terminăm cu system_info, adăugăm doar o comandă echo ca aceasta:

```
system_info()
{
    # Temporary function stub
    echo "function system_info"
}
```

În acest fel, scriptul nostru se va executa în continuare cu succes, chiar dacă nu avem încă o funcție system_info finalizată. Vom înlocui ulterior codul temporar cu versiunea completă de lucru.

Motivul pentru care folosim o comandă echo este că primim un feedback din script pentru a indica faptul că funcțiile sunt executate.

Să mergem mai departe și să scriem stub-uri pentru noile noastre funcții și să menținem scriptul să funcționeze.

```
#!/bin/bash

# sysinfo_page - A script to produce an system information HTML
file

##### Constants

TITLE="System Information for $HOSTNAME"
RIGHT_NOW="$(date +"%x %r %Z")"
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

##### Functions

system_info()
{
    # Temporary function stub
    echo "function system_info"
}


show_uptime()
{
    # Temporary function stub
    echo "function show_uptime"
}


drive_space()
{
    # Temporary function stub
    echo "function drive_space"
}


home_space()
{
    # Temporary function stub
    echo "function home_space"
}


##### Main
```

```
cat <<- _EOF_
  <html>
  <head>
      <title>$TITLE</title>
  </head>

  <body>
      <h1>$TITLE</h1>
      <p>$TIME_STAMP</p>
      $(system_info)
      $(show_uptime)
      $(drive_space)
      $(home_space)
  </body>
  </html>
_EOF_
```

În continuare, vom dezvolta unele dintre funcţiile noastre de shell şi vom face ca scriptul nostru să producă câteva informaţii utile.

7.8. Functia show_uptime

Funcţia show_uptime va afişa rezultatul comenzii uptime. Comanda uptime returneaza cateva interesante despre sistem, inclusiv perioada de timp în care sistemul a fost în funcţiune de la ultima repornire, numărul de utilizatori şi încărcarea recentă a sistemului.

uptime

```
userlabso@LabSoVM:~$ uptime
 16:16:34 up 53 min,  1 user,  load average: 0,03, 0,11, 0,10
userlabso@LabSoVM:~$
```

Pentru a obţine rezultatul comenzii uptime în pagina noastră HTML, vom coda funcţia noastră shell astfel, înlocuind codul nostru temporar stubbing cu versiunea finală:

```
show_uptime()
{
    echo "<h2>System uptime</h2>"
    echo "<pre>"
    uptime
    echo "</pre>"
}
```

După cum putem vedea, această funcţie emite un flux de text care conţine un amestec de etichete HTML şi comenzi. Când înlocuirea comenzii are loc în corpul principal al programului nostru, outputul funcţiei noastre devine parte a scriptului.

7.9. Functia drive_space

Funcția drive_space va folosi comanda df pentru a oferi un rezumat al spațiului folosit de toate sistemele de fișiere montate.

df

```
userlabso@LabSoVM:~$ df
Filesystem     1K-blocks    Used Available Use% Mounted on
tmpfs            5017744    1572   5016172   1% /run
/dev/sda2       14311880 9929208   3633872  74% /
tmpfs           25088716       0  25088716   0% /dev/shm
tmpfs               5120       4      5116   1% /run/lock
/dev/sda4       32829200  158280  30977736   1% /home
/dev/sda1         210872  157720     36168  82% /boot
tmpfs            5017740     112   5017628   1% /run/user/1000
userlabso@LabSoVM:~$
```

În ceea ce privește structura, funcția drive_space este foarte asemănătoare cu funcția show_uptime:

```
drive_space()
{
    echo "<h2>Filesystem space</h2>"
    echo "<pre>"
    df -h
    echo "</pre>"
}
```

7.10. Functia home_space

Funcția home_space va afișa un sumar (optiunea -s pentru du) al cantitatii de spațiu folosit de fiecare utilizator în directorul său principal. Acesta va fi afișata ca o listă, sortată în ordine descrescătoare (optiunea -r pentru sort) după cantitatea de spațiu utilizată (optiunea -n pentru sort) intr-un format mai usor de citit (optiunea -h pentru du).

```
home_space()
{
    echo "<h2>Home directory space by user</h2>"
    echo "<pre>"
    echo "Bytes Directory"
    du -sh /home/* | sort -nr
    echo "</pre>"
}
```

Rețineți că, pentru ca această funcție să se execute cu succes, scriptul trebuie să fie rulat de superutilizator, deoarece comanda du necesită privilegii de superutilizator pentru a examina conținutul directorului /home.

7.11. Functia system_info

Nu suntem încă pregătiți să finalizăm funcția system_info. Între timp, vom îmbunătăți codul stubbing, astfel încât să producă HTML valid:

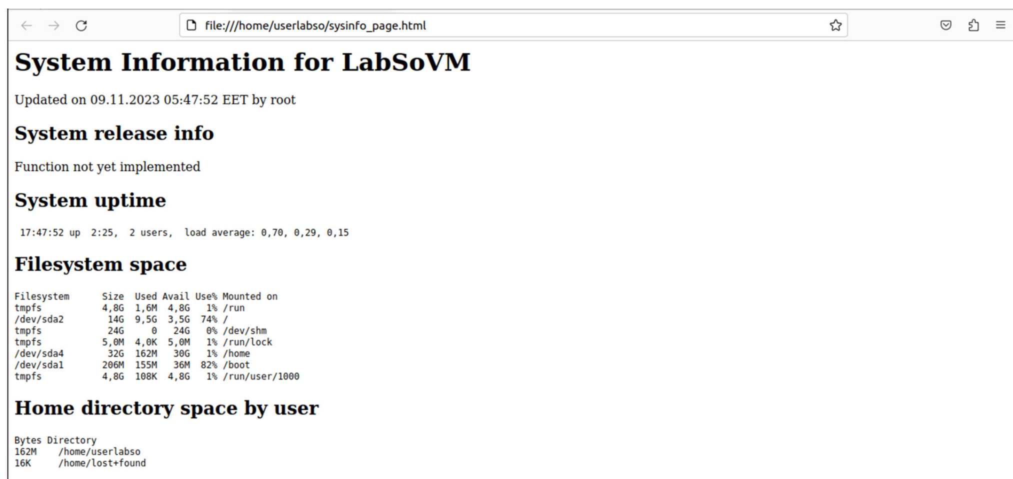
```
system_info()
{
    echo "<h2>System release info</h2>"
    echo "<p>Function not yet implemented</p>"
}
```

Acum ca am adaugat toate acestea scriptului nostru, nu ne ramane decat sa il executam si sa vedem daca totul functioneaza asa cum ne asteptam:

```
sudo ./sysinfo_page > sysinfo_page.html
```

```
userlabso@LabSoVM:~$ sudo ./sysinfo_page > sysinfo_page.html
[sudo] password for userlabso:
userlabso@LabSoVM:~$
```

După toate modificările aduse scriptului nostru, după executare ar trebui să ne producă un fișier HTML care arat așa:



8. SCRIPTURI SHELL ÎN SISTEME LINUX – CONTROLUL FLUXULUI DE EXECUȚIE: RAMIFICARE CU IF

În această lucrare vom încerca să adăugăm inteligență scripturilor noastre. Până acum, scriptul nostru a constat doar dintr-o secvență de comenzi care încep de la prima linie și continuă linie cu linie până ce ajungem la sfârșit. Majoritatea programelor fac mult mai mult decât atât - iau decizii și efectuează diferite acțiuni în funcție de condiții.

Shell-ul oferă mai multe comenzi pe care le putem folosi pentru a controla fluxul de execuție în programul nostru. În continuare, ne vom uita la următoarele:

- if
- test
- exit

if

Prima comandă la care ne vom uita este if. Comanda if este destul de simplă la suprafață; ia o decizie pe baza stării de ieșire a unei comenzi. Sintaxa comenzii if arată astfel:

```
if commands; then
    commands
[elif commands; then
    commands...]
[else
    commands]
fi
```

unde commands este o listă de comenzi. Acest lucru este puțin confuz la prima vedere. Dar înainte de a putea clarifica acest lucru, trebuie să ne uităm la modul în care shell-ul evaluează succesul sau eșecul unei comenzi.

8.1. Exit Status

Comenzile (inclusiv scripturile și funcțiile shell pe care le scriem) emit o valoare sistemului atunci când se încheie, numită exit status. Această valoare, care este un număr întreg de la 0 la 255, indică succesul sau eșecul execuției comenzii. Prin convenție, o valoare de zero indică succesul și orice altă valoare indică eșec. Shell-ul oferă un parametru pe care îl putem folosi pentru a examina starea de ieșire. Aici îl vedem în acțiune prin urmatoarele comenzi:

```
ls -d /usr/bin

echo $?

ls -d /bin/usr

echo $?
```

```
userlabso@LabSoVM:~$ ls -d /usr/bin
/usr/bin
userlabso@LabSoVM:~$ echo $?
0
userlabso@LabSoVM:~$ ls -d /bin/usr
ls: cannot access '/bin/usr': No such file or directory
userlabso@LabSoVM:~$ echo $?
2
userlabso@LabSoVM:~$
```

În acest exemplu, executăm comanda ls de două ori. Prima dată, comanda se execută cu succes. Dacă afișăm valoarea parametrului $?, vedem că este zero. Executăm comanda ls a doua oară, producând o eroare și examinăm parametrul $? din nou. De data aceasta conține un 2, indicând faptul că am întâmpinat o eroare la executarea comenzii. Unele comenzi folosesc valori diferite ale stării de ieșire pentru a oferi diagnostice pentru erori, în timp ce multe comenzi pur și simplu ies cu o valoare de 1 atunci când eșuează. Paginile de manual includ adesea o secțiune intitulată „Exit Status", care descrie codurile folosite. Cu toate acestea, un zero indică întotdeauna succes.

Shell-ul oferă două comenzi încorporate extrem de simple, care nu fac nimic decât să se încheie fie cu o stare de ieșire 0, fie cu 1. Comanda true se execută întotdeauna cu succes, iar comanda false se execută întotdeauna fără succes:

```
true
echo $?
false
echo $?
```

```
userlabso@LabSoVM:~$ true
userlabso@LabSoVM:~$ echo $?
0
userlabso@LabSoVM:~$ false
userlabso@LabSoVM:~$ echo $?
1
userlabso@LabSoVM:~$
```

Putem folosi aceste comenzi pentru a vedea cum funcționează instrucțiunea if. Ceea ce face cu adevărat declarația if este să evalueze succesul sau eșecul comenzilor:

```
if true; then echo "It's true."; fi
if false; then echo "It's true."; fi
```

```
userlabso@LabSoVM:~$ if true; then echo "It's true."; fi
It's true.
userlabso@LabSoVM:~$ if false; then echo "It's true."; fi
userlabso@LabSoVM:~$
```

Comanda echo „It's true." este executata când comanda care urmează if se execută cu succes și nu este executata când comanda care urmează if nu se execută cu succes.

test

Comanda test este folosită cel mai des cu comanda if pentru a efectua decizii adevărat/fals. Comanda este neobișnuită prin faptul că are două forme sintactice diferite:

```
# Prima forma
test expression
# A doua forma
[ expression ]
```

Comanda test funcționează simplu. Dacă expresia dată este adevărată, testul iese cu starea 0; altfel iese cu starea 1. Avantajul comenzii test este varietatea de expresii pe care le putem crea. Iată un exemplu:

```
if [ -f .bash_profile ]; then
    echo "Fisierul .bash_profile exista. Totul e OK."
else
    echo "Uh oh.... Fisierul .bash_profile nu exista!"
fi
```

În acest exemplu, folosim expresia „-f .bash_profile”. Această expresie întreabă: „Este .bash_profile un fișier?” Dacă expresia este adevărată, atunci testul iese cu zero (care indică adevărat) și comanda if execută comanda (comenzile) după cuvântul then. Dacă expresia este falsă, atunci testul iese cu starea unu și comanda if execută comanda (comenzile) după cuvântul else.

Iată o listă parțială a condițiilor pe care test le poate evalua. Deoarece test este încorporat în shell, utilizați „help test” pentru a vedea o listă completă.

Expression	**Description**
-d *file*	True if *file* is a directory.
-e *file*	True if *file* exists.
-f *file*	True if *file* exists and is a regular file.
-L *file*	True if *file* is a symbolic link.
-r *file*	True if *file* is a file readable by you.
-w *file*	True if *file* is a file writable by you.
-x *file*	True if *file* is a file executable by you.
file1 -nt *file2*	True if *file1* is newer than (modification time) *file2*.
file1 -ot *file2*	True if *file1* is older than *file2*.
-z *string*	True if *string* is empty.

-n *string*	True if *string* is not empty.
string1 = *string2*	True if *string1* equals *string2*.
string1 != *string2*	True if *string1* does not equal *string2*.

Înainte de a continua, trebuie să explicăm restul exemplului de mai sus, deoarece dezvăluie și alte lucruri importante.

În prima linie a scriptului, vedem comanda if urmată de comanda test, urmată de ; și, în final, cuvântul then. Majoritatea aleg să folosească forma [expresie] a comenzii test, deoarece este mai ușor de citit. Observați că sunt necesare spații între „[” și începutul expresiei. La fel, spațiul dintre sfârșitul expresiei și următorul „]”.

; este un separator de comenzi. Folosirea acestuia ne permite să punem mai mult de o comandă pe o linie. De exemplu:

```
clear; ls
```

va șterge ecranul și va executa comanda ls.

Folosim punctul și virgulă așa cum am făcut pentru a ne permite să punem cuvântul then pe aceeași linie cu comanda if, pentru că este mai ușor de citit în acest fel.

Pe a doua linie, este vechiul nostru prieten echo. Singurul lucru de notat pe această linie este indentarea. Din nou, pentru beneficiul lizibilității, este tradițional să indentați toate blocurile de cod condiționat; adică orice cod care va fi executat doar dacă sunt îndeplinite anumite condiții. Shell-ul nu necesită acest lucru; este făcut pentru a face codul mai ușor de citit.

Cu alte cuvinte, am putea scrie următoarele și obținem aceleași rezultate:

```
# Forma preferata

     if [ -f .bash_profile ]; then
         echo "You have a .bash_profile. Things are fine."
     else
         echo "Yikes! You have no .bash_profile!"
     fi

# Forma alternativa

     if [ -f .bash_profile ]
     then echo "You have a .bash_profile. Things are fine."
     else echo "Yikes! You have no .bash_profile!"
     fi
```

exit

O buna practica este setarea starii de ieșire când se termină scripturile noastre. Pentru a face acest lucru, utilizați comanda exit. Comanda de ieșire face ca scriptul să se termine imediat și setează starea de ieșire la orice valoare este dată ca argument. De exemplu:

```
exit 0
```

iese din scriptul nostru şi setează starea de ieşire la 0 (succes), în timp ce

```
exit 1
```

iese din script şi setează starea de ieşire la 1 (eşec).

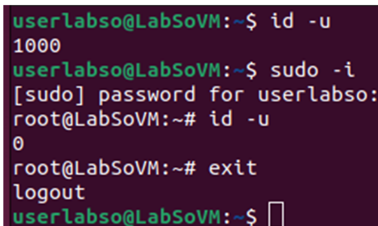
8.2. Testarea pentru Root

Când am părăsit ultima dată scriptul, am cerut ca acesta să fie rulat cu privilegii de superutilizator. Acest lucru se datorează faptului că funcția home_space trebuie să examineze dimensiunea directorului principal al fiecărui utilizator şi numai superutilizatorul are voie să facă asta.

Dar ce se întâmplă dacă un utilizator obişnuit rulează scriptul nostru? Produce o mulțime de mesaje de eroare. Oare am putea pune ceva în script pentru a-l opri dacă un utilizator obişnuit încearcă să-l ruleze?

Comanda id ne poate spune cine este utilizatorul curent. Când este executat cu opțiunea „-u", se tipăreşte ID-ul de utilizator numeric al utilizatorului curent.

```
id -u
sudo -i
id -u
exit
```



Dacă superutilizatorul execută id -u, comanda va scoate „0". Acest fapt poate sta la baza testului nostru:

```
if [ "$(id -u)" = "0" ]; then
    echo "superuser"
fi
```

În acest exemplu, dacă rezultatul comenzii id -u este egală cu şirul „0", atunci tipăriți şirul „superuser".

Deşi acest cod va detecta dacă utilizatorul este superutilizatorul, nu rezolvă încă problema. Dorim să oprim scriptul dacă utilizatorul nu este superutilizatorul, aşa că îl vom codifica astfel:

```
if [ "$(id -u)" != "0" ]; then
    echo "You must be the superuser to run this script" >&2
    exit 1
fi
```

Cu acest cod, dacă ieșirea comenzii id -u nu este egală cu „0”, atunci scriptul tipărește un mesaj de eroare descriptiv, iese și setează starea de ieșire la 1, indicând sistemului de operare că scriptul a fost executat fără succes .

Observați „>&2” de la sfârșitul comenzii echo. Aceasta este o altă formă de direcție I/O. Vom vedea adesea acest lucru în rutinele care afișează mesaje de eroare. Dacă această redirecționare nu s-ar face, mesajul de eroare va merge la ieșirea standard. Cu această redirecționare, mesajul este trimis la eroare standard. Deoarece ne executăm scriptul și redirecționăm rezultatul standard către un fișier, dorim ca mesajele de eroare să fie separate de rezultatul normal.

Am putea pune această rutină aproape de începutul scriptului nostru, astfel încât să aibă șansa de a detecta o posibilă eroare înainte ca lucrurile să înceapă, dar pentru a rula acest script ca un utilizator obișnuit, vom folosi aceeași idee și vom modifica funcția home_space pentru a testa privilegiile adecvate, astfel:

```
function home_space {
    # Only the superuser can get this information

    if [ "$(id -u)" = "0" ]; then
        echo "<h2>Home directory space by user</h2>"
        echo "<pre>"
        echo "Bytes Directory"
            du -sh /home/* | sort -nr
        echo "</pre>"
    fi

}   # end of home_space
```

În acest fel, dacă un utilizator obișnuit rulează scriptul, codul deranjant va fi trecut, mai degrabă decât executat, iar problema va fi rezolvată.

8.3. “Stay Out of Trouble”

Acum că scripturile noastre devin puțin mai complicate, să ne uităm la câteva greșeli comune în care am putea întâlni. Pentru a face acest lucru, vom crea următorul script numit trouble.bash. Asigurați-vă că îl introduceți exact așa cum este scris.

```
gedit trouble.bash &
```

```
#!/bin/bash

number=1

if [ $number = "1" ]; then
    echo " Numărul este egal cu 1"
else
    echo " Numărul nu este egal cu 1"
fi
```

```
chmod 755 trouble.bash
```

```
./trouble.bash
```

Când rulăm acest script, ar trebui să scoată linia „Numărul este egal cu 1”, deoarece, ei bine, numărul este egal cu 1. Dacă nu obținem rezultatul așteptat, trebuie să ne verificăm tastarea; am facut o greseala.

```
userlabso@LabSoVM:~$ gedit trouble.bash &
[1] 2915
userlabso@LabSoVM:~$ ls
bin        Downloads  Public        sysinfo_page.html  trouble.bash
Desktop    Music      snap          Templates          tutorial
Documents  Pictures   sysinfo_page  test               Videos
[1]+  Done                    gedit trouble.bash
userlabso@LabSoVM:~$ chmod 755 trouble.bash
userlabso@LabSoVM:~$ ls
bin        Downloads  Public        sysinfo_page.html  trouble.bash
Desktop    Music      snap          Templates          tutorial
Documents  Pictures   sysinfo_page  test               Videos
userlabso@LabSoVM:~$ ./trouble.bash
Number equals 1
userlabso@LabSoVM:~$
```

8.4. Variabile goale

Să edităm scriptul pentru a schimba linia 3 din:

```
number=1
```

in:

```
number=
```

și rulați din nou scriptul. De data aceasta ar trebui să obținem următoarele:

```
userlabso@LabSoVM:~$ gedit trouble.bash &
[1] 3006
userlabso@LabSoVM:~$ ./trouble.bash
./trouble.bash: line 5: [: =: unary operator expected
Number does not equal 1
[1]+  Done                    gedit trouble.bash
userlabso@LabSoVM:~$
```

După cum putem vedea, bash a afișat un mesaj de eroare când am rulat scriptul. Am putea crede că prin eliminarea „1” de pe linia 3 a creat o eroare de sintaxă pe linia 3, dar nu a făcut-o. Să ne uităm din nou la mesajul de eroare:

```
./trouble.bash: line 5: [: =: unary operator expected
```

Putem vedea că ./trouble.bash raportează eroarea și eroarea are legătură cu „[”. Amintiți-vă că „[” este o abreviere pentru comanda încorporata in shell test. Din aceasta putem determina că eroarea are loc pe linia 5, nu pe linia 3.

În primul rând, pentru a fi clar, nu este nimic în neregulă cu linia 3. number= este o sintaxă validă. Uneori vrem să setăm valoarea unei variabile la nimic. Putem confirma validitatea acestui lucru încercând pe linia de comandă:

```
number=
```

```
userlabso@LabSoVM:~$ number=
userlabso@LabSoVM:~$
```

Vezi, nici un mesaj de eroare. Deci, ce este în neregulă cu linia 5? A funcționat înainte.

Pentru a înțelege această eroare, trebuie să vedem ce vede shell-ul. Amintiți-vă că shell-ul își petrece o mare parte din viață extinzând textul. În linia 5, shell-ul extinde valoarea numărului unde vede $number. În prima noastră încercare (când numărul=1), shell-ul a înlocuit $number cu 1, așa:

```
if [ 1 = "1" ]; then
```

Totusi, când setăm numărul la nimic (number=), shell-ul a văzut asta după extindere:

```
if [ = "1" ]; then
```

care este o eroare. De asemenea, explică restul mesajului de eroare pe care l-am primit. „=" este un operator binar; adică se așteaptă ca două elemente să opereze - câte unul pe fiecare parte. Ceea ce shell-ul încearcă să ne spună este că există un singur element și ar trebui să existe un operator unar (cum ar fi „!”) care operează doar pe un singur articol.

Pentru a remedia această problemă, modificați rândul 5:

```
if [ "$number" = "1" ]; then
```

Acum, când shell-ul efectuează expansiunea, va vedea:

```
if [ "" = "1" ]; then
```

care exprimă corect intenţia noastră.

Acest lucru aduce două lucruri importante de reținut când scriem scripturi. Trebuie să luăm în considerare ce se întâmplă dacă o variabilă este setată la nimic și ar trebui să punem întotdeauna ghilimele duble în jurul parametrilor care sunt supuși expansiunii.

8.5. Ghilimele lipsă

Editați linia 6 pentru a elimina ghilimelele de la sfârșitul rândului:

```
echo "Number equals 1
```

și rulați din nou scriptul. Ar trebui să obținem asta:

```
trouble.bash
~/
1 #!/bin/bash
2
3 number=
4
5 if [ "$number" = "1" ]; then
6     echo "Number equals 1
7 else
8     echo "Number does not equal 1"
9 fi
```

```
userlabso@LabSoVM: ~
userlabso@LabSoVM:~$ ./trouble.bash
./trouble.bash: line 8: unexpected EOF while looking for matching `"'
./trouble.bash: line 10: syntax error: unexpected end of file
userlabso@LabSoVM:~$
```

Aici avem un alt exemplu de greșeală într-o linie care provoacă o problemă mai târziu în script. Ceea ce s-a întâmplat în acest caz a fost că shell-ul a continuat să caute ghilimelele de închidere pentru a determina unde este sfârșitul șirului, dar a ajuns la sfârșitul fișierului înainte de a-l găsi.

Aceste erori pot fi dificil de urmărit într-un script lung. Acesta este unul dintre motivele pentru care ar trebui să ne testăm frecvent scripturile în timp ce scriem, astfel încât să avem mai puțin cod nou de testat. De asemenea, utilizarea unui editor de text cu evidențiere de sintaxă face ca aceste erori să fie mai ușor de găsit.

8.6. Izolarea problemelor

Găsirea erorilor în scripturi poate fi uneori foarte dificilă și frustrantă. Iată câteva tehnici care sunt utile:

Izolați blocurile de cod „comentându-le”. Acest truc implică punerea caracterelor de comentariu la începutul liniilor de cod pentru a opri shell-ul să le citească. Putem face acest lucru unui bloc de cod pentru a vedea dacă o anumită problemă dispare. Făcând acest lucru, putem izola care parte a unui program cauzează (sau nu cauzează) o problemă.

De exemplu, atunci când căutam ghilimeaua noastră lipsă, am fi putut face asta:

```
#!/bin/bash
number=1
if [ $number = "1" ]; then
    echo "Number equals 1
#else
#   echo "Number does not equal 1"
fi
```

Comentând clauza else și rulând scriptul, am putea arăta că problema nu se afla în clauza else, chiar dacă mesajul de eroare sugera că este.

Utilizați comenzi echo pentru a verifica ipotezele. Pe măsură ce dobândim experiență în urmărirea erorilor, vom descoperi că deseori erorile nu sunt acolo unde ne așteptăm prima dată să le găsim. O problemă comună va fi aceea că vom face o presupunere falsă cu privire la performanța programului nostru. O problemă se va dezvolta la un anumit punct al programului și presupunem că problema există. Acest lucru este adesea incorect. Pentru a combate acest lucru, putem plasa comenzi echo în cod în timp ce depanăm, pentru a produce mesaje care confirmă că programul face ceea ce este așteptat. Există două tipuri de mesaje pe care le putem introduce.

Primul tip anunță pur și simplu că am ajuns la un anumit punct al programului. Am văzut acest lucru în discuția noastră anterioară despre stubbing. Este util să știm că fluxul programului evoluează așa cum ne așteptăm.

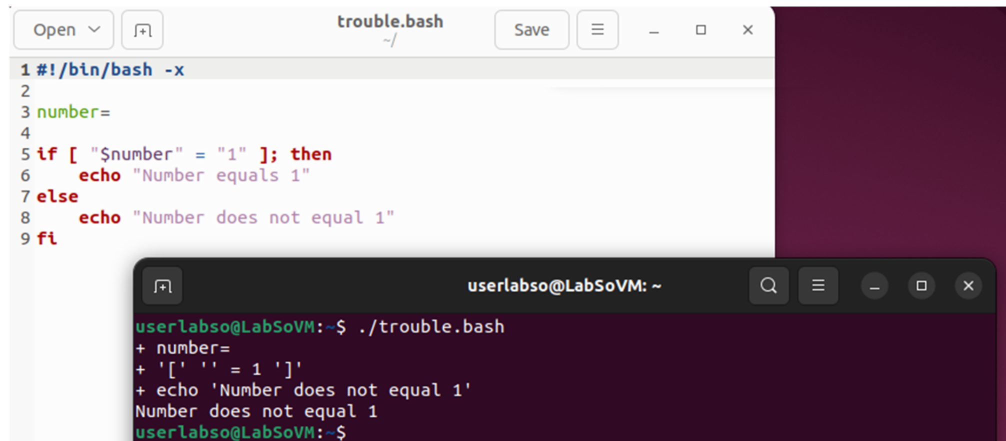
Al doilea tip afișează valoarea unei variabile (sau variabile) utilizate într-un calcul sau test. Vom descoperi adesea că o parte a unui program va eșua, deoarece ceva despre care am presupus că era corect mai devreme în program este, de fapt, incorect și provoacă eșuarea programului nostru mai târziu.

8.7. Urmărirea rulării scriptului

Este posibil ca bash să ne arate ce face atunci când rulăm scriptul. Pentru a face acest lucru, adăugați un „-x” la prima linie a scriptului, astfel:

```
#!/bin/bash -x
```

Acum, când rulăm scriptul, bash va afișa fiecare linie (cu extinderile efectuate) pe măsură ce o execută. Această tehnică se numește urmărire (tracing). Iată cum arată:



Alternativ, putem folosi comanda set din script pentru a activa și dezactiva urmărirea. Folosiți set -x pentru a activa urmărirea și set +x pentru a dezactiva urmărirea. De exemplu:

```
#!/bin/bash
number=1
set -x
if [ $number = "1" ]; then
    echo "Number equals 1"
```

```
else
    echo "Number does not equal 1"
fi
set +x
```

```
trouble.bash
~/
1 #!/bin/bash
2
3 number=1
4
5 set -x
6 if [ $number = "1" ]; then
7     echo "Number equals 1"
8 else
9     echo "Number does not equal 1"
10 fi
11 set +x
```

```
userlabso@LabSoVM: ~
userlabso@LabSoVM:~$ ./trouble.bash
+ '[' 1 = 1 ']'
+ echo 'Number equals 1'
Number equals 1
+ set +x
userlabso@LabSoVM:~$
```

8.8. Intrare de la tastatură

Până acum, scripturile noastre nu au fost interactive. Adică nu au acceptat nicio intrare de la utilizator. În continuare, vom vedea cum scripturile noastre pot pune întrebări și pot obține și utiliza răspunsuri.

read

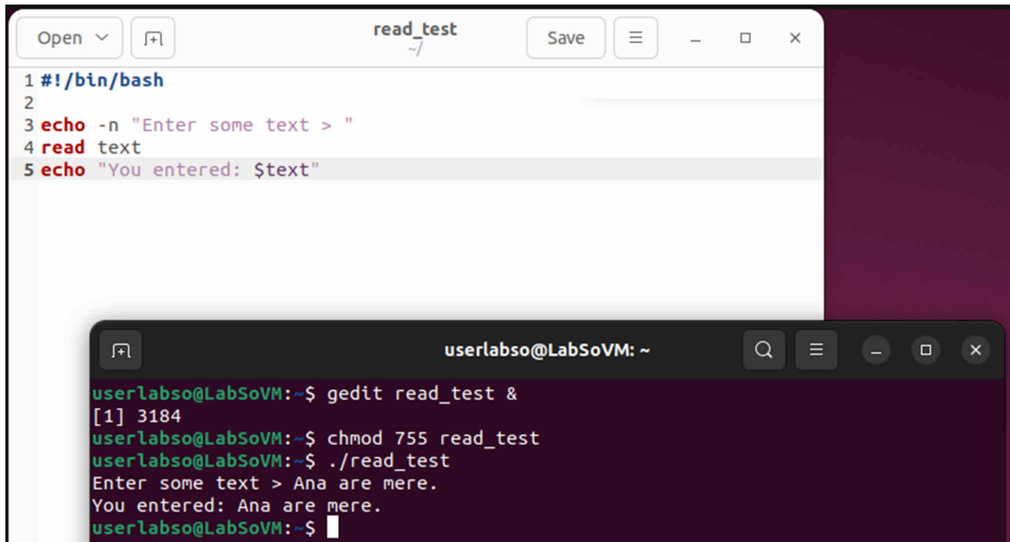
Pentru a obține intrare de la tastatură, folosim comanda read (citire). Comanda de citire preia intrarea de la tastatură și o atribuie unei variabile. Iată un exemplu:

gedit read_test &

```
#!/bin/bash

echo -n "Enter some text > "
read text
echo "You entered: $text"
```

chmod 755 read_test

./read_test



După cum putem vedea, am afișat un prompt pe linia 3. Rețineți că „-n" dat comenzii echo face ca aceasta să mențină cursorul pe aceeași linie; adică nu scoate un linefeed la sfârșitul promptului.

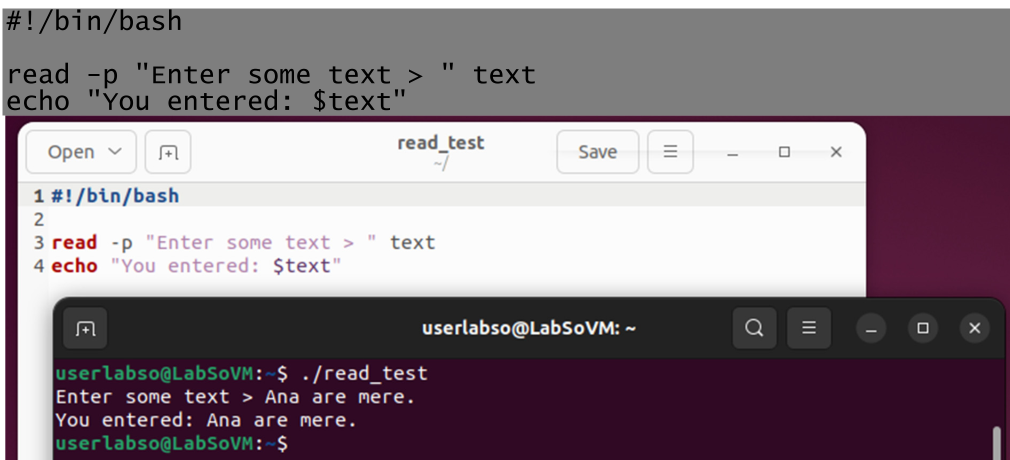
Apoi, invocăm comanda citire cu „text" ca argument. Ceea ce face aceasta este să aștepte ca utilizatorul să tasteze ceva urmat de tasta Enter și apoi să atribuie textul variabil orice a fost introdus.

Dacă nu dăm comenzii read numele unei variabile pentru a-i atribui intrarea, aceasta va folosi variabila de mediu REPLY.

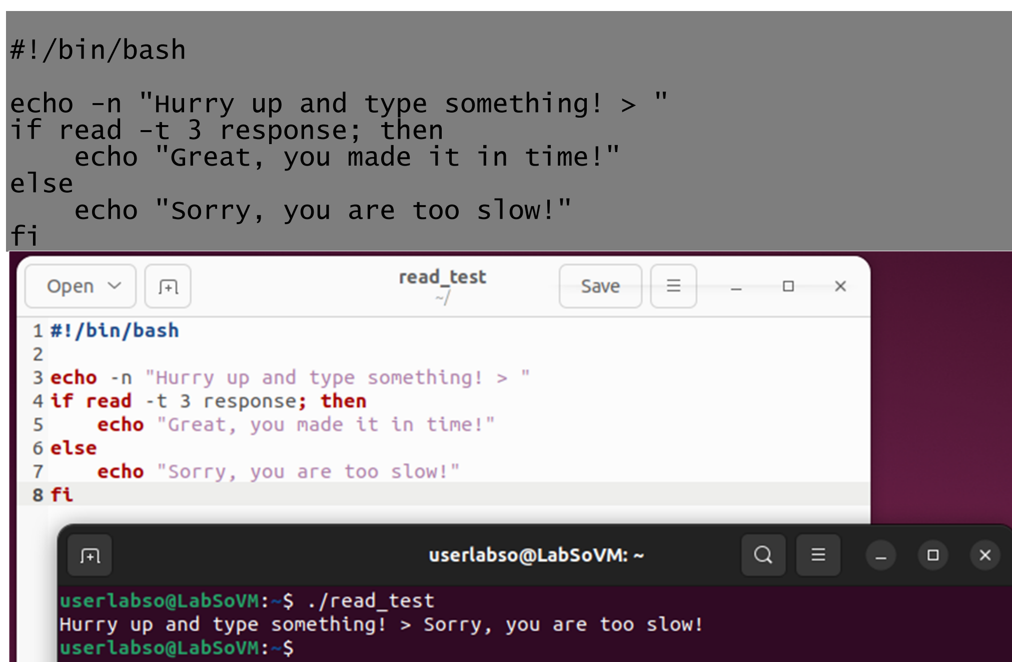
Comanda de citire are mai multe opțiuni de linie de comandă. Cele mai interesante trei sunt -p, -t și -s.

Opțiunea -p ne permite să specificăm un prompt care va precede intrarea utilizatorului. Acest lucru salvează pasul suplimentar de utilizare a unui echo pentru a solicita utilizatorului. Iată exemplul anterior rescris pentru a utiliza opțiunea -p:

```
#!/bin/bash

read -p "Enter some text > " text
echo "You entered: $text"
```



Opțiunea -t urmată de un număr de secunde oferă un timeout automat pentru comanda de citire. Aceasta înseamnă că comanda de citire va renunța după numărul specificat de secunde dacă nu a fost primit niciun răspuns de la utilizator. Această opțiune ar putea fi folosită în cazul unui script care trebuie să continue (poate apelând la un răspuns implicit) chiar dacă utilizatorul nu răspunde la solicitări. Iată opțiunea -t în acțiune:



Opțiunea -s face ca tastarea utilizatorului să nu fie afișată. Acest lucru este util atunci când îi cerem utilizatorului să introducă o parolă sau alte informații confidențiale.

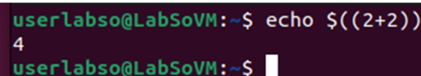
8.9. Aritmetică

Deoarece lucrăm pe un computer, este firesc să ne așteptăm că acesta poate efectua niște calcule aritmetice simple. Shell-ul oferă caracteristici pentru aritmetica numerelor întregi.

Ce este un număr întreg? Asta înseamnă numere întregi precum 1, 2, 458, -2859. Nu înseamnă numere fracționale precum 0,5, .333 sau 3,1415. Pentru a trata numerele fracționale, există un program separat numit bc care oferă un limbaj de calcul de precizie arbitrară. Poate fi folosit în scripturi shell, dar depășește scopul acestui tutorial.

Să presupunem că vrem să folosim linia de comandă ca un calculator primitiv. O putem face astfel:

```
echo $((2+2))
```



Când înconjurăm o expresie aritmetică cu paranteze duble, shell-ul va efectua o expansiune aritmetică.

Observați că spațiile albe sunt ignorate:

echo $((2+2))

echo $((2+2))

echo $((2 + 2))

```
userlabso@LabSoVM:~$ echo $((2+2))
4
userlabso@LabSoVM:~$ echo $(( 2+2 ))
4
userlabso@LabSoVM:~$ echo $(( 2 + 2 ))
4
userlabso@LabSoVM:~$
```

Shell-ul poate efectua o varietate de operații aritmetice comune (sau mai puțin comune). Iată un exemplu:

gedit aritmetic &

```
#!/bin/bash

first_num=0
second_num=0

read -p "Enter the first number --> " first_num
read -p "Enter the second number -> " second_num

echo "first number + second number = $((first_num + second_num))"
echo "first number - second number = $((first_num - second_num))"
echo "first number * second number = $((first_num * second_num))"
echo "first number / second number = $((first_num / second_num))"
echo "first number % second number = $((first_num % second_num))"
echo "first number raised to the"
echo "power of the second number = $((first_num ** second_num))"
```

chmod 755 aritmetic

./aritmetic

```
aritmetic
~/
1 #!/bin/bash
2
3 first_num=0
4 second_num=0
5
6 read -p "Enter the first number --> " first_num
7 read -p "Enter the second number -> " second_num
8
9 echo "first number + second number = $((first_num + second_num))"
10 echo "first number - second number = $((first_num - second_num))"
11 echo "first number * second number = $((first_num * second_num))"
12 echo "first number / second number = $((first_num / second_num))"
13 echo "first number % second number = $((first_num % second_num))"
14 echo "first number raised to the"
15 echo "power of the second number   = $((first_num ** second_num))"
```

```
userlabso@LabSoVM:~$ gedit aritmetic &
[1] 3405
userlabso@LabSoVM:~$ chmod 755 aritmetic
userlabso@LabSoVM:~$ ./aritmetic
Enter the first number --> 34
Enter the second number -> 2
first number + second number = 36
first number - second number = 32
first number * second number = 68
first number / second number = 17
first number % second number = 0
first number raised to the
power of the second number   = 1156
userlabso@LabSoVM:~$
```

Observați cum „$” nu este necesar pentru a face referire la variabile din interiorul expresiei aritmetice, cum ar fi „first_num + second_num”.

Încercați acest program și urmăriți cum se ocupă de diviziunea (rețineți, aceasta este diviziunea de numere întregi) și cum se ocupă de numerele mari. Numerele care devin prea mari, cum ar fi contorul de parcurs dintr-o mașină, atunci când depășesc numărul de kilometri pentru care a fost proiectat să le numere. Începe de la capăt, dar mai întâi trece prin toate numerele negative din cauza modului în care sunt reprezentate numerele întregi în memorie. Împărțirea cu zero (care este invalidă din punct de vedere matematic) provoacă o eroare.

Primele patru operații, adunarea, scăderea, înmulțirea și împărțirea, sunt ușor de recunoscut, dar a cincea poate fi necunoscută. Simbolul „%” reprezintă restul (cunoscut și ca modulo). Această operație efectuează împărțirea, dar în loc să returneze un coeficient ca o împărțire, returnează restul. Deși acest lucru ar putea să nu pară foarte util, oferă, de fapt, o mare utilitate atunci când scrieți programe. De exemplu, când o operație cu rest returnează zero, indică faptul că primul număr este un multiplu exact al celui de-al doilea. Acest lucru poate fi foarte util:

gedit odd_even &

```
#!/bin/bash

number=0

read -p "Enter a number > " number

echo "Number is $number"
if [ $((number % 2)) -eq 0 ]; then
    echo "Number is even"
else
    echo "Number is odd"
fi
```

chmod 755 odd_even

./odd_even

Sau, în acest program care formatează un număr arbitrar de secunde în ore și minute:

gedit ore_minute &

```
#!/bin/bash

seconds=0

read -p "Enter number of seconds > " seconds

hours=$((seconds / 3600))
seconds=$((seconds % 3600))
minutes=$((seconds / 60))
seconds=$((seconds % 60))

echo "$hours hour(s) $minutes minute(s) $seconds second(s)"
```

chmod 755 ore_minute

./ore_minute

9. SCRIPTURI SHELL ÎN SISTEME LINUX - CONTROLUL FLUXULUI DE EXECUȚIE: BUCLE CU WHILE / UNTIL

În lucrarea anterioară privind controlul fluxului am învățat despre comanda if și cum este folosită pentru a modifica fluxul programului pe baza statusului de ieșire a unei comenzi. În termeni de programare, acest tip de flux de program se numește ramificare (branching) deoarece este ca și cum ați traversa un arbore. Ajungem la o bifurcație în copac și evaluarea unei conditii determină ce ramură luăm.

Există un al doilea și mai complex tip de ramificare numit caz (case). Un caz este o ramură cu alegere multiplă. Spre deosebire de ramura simplă, în care luăm una dintre cele două căi posibile, un caz acceptă mai multe rezultate posibile bazate pe evaluarea unei valori.

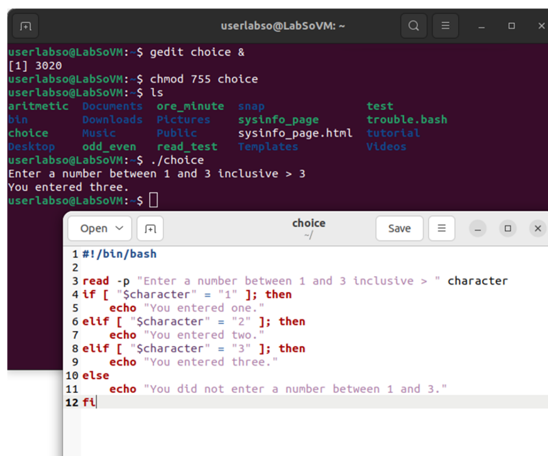
Putem construi acest tip de ramură cu mai multe instrucțiuni if. În exemplul de mai jos, evaluăm intrări de la utilizator:

gedit choice &

```
#!/bin/bash

read -p "Enter a number between 1 and 3 inclusive > " character
if [ "$character" = "1" ]; then
    echo "You entered one."
elif [ "$character" = "2" ]; then
    echo "You entered two."
elif [ "$character" = "3" ]; then
    echo "You entered three."
else
    echo "You did not enter a number between 1 and 3."
fi
```

chmod 755 choice

./choice



Nu arată foarte frumos.

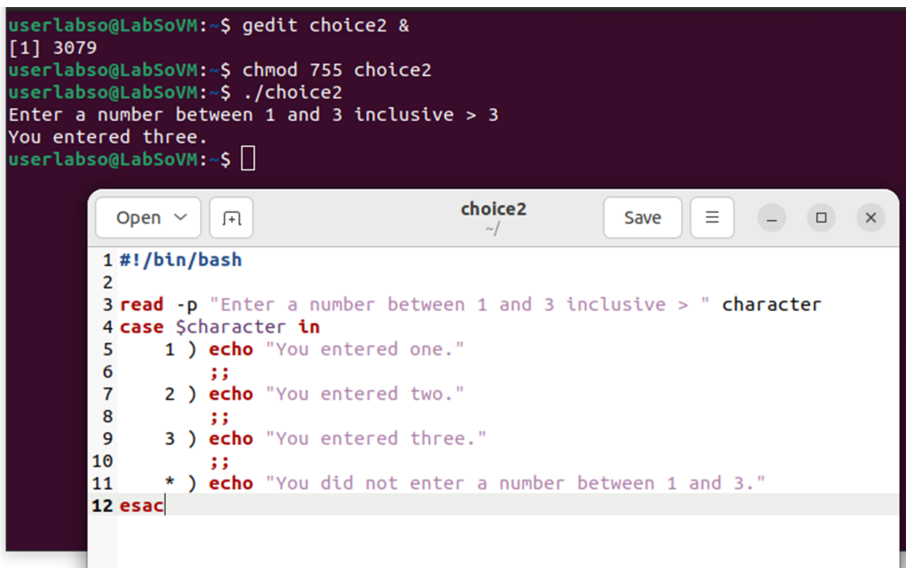
Din fericire, shell-ul oferă o soluție mai elegantă la această problemă. Oferă o comandă încorporată numită case, care poate fi folosită pentru a construi un program echivalent:

gedit choice2 &

```
#!/bin/bash

read -p "Enter a number between 1 and 3 inclusive > " character
case $character in
    1 ) echo "You entered one."
        ;;
    2 ) echo "You entered two."
        ;;
    3 ) echo "You entered three."
        ;;
    * ) echo "You did not enter a number between 1 and 3."
esac
```

chmod 755 choice2

./choice2



Comanda case are următoarea formă:

```
case word in
    patterns ) commands ;;
esac
```

case execută selectiv instrucțiuni dacă declaratia (word) se potrivește cu un model (pattern). Putem avea orice număr de modele și declarații. Modelele pot fi text literal sau wildcards. Putem avea mai multe modele separate prin caracterul „|”. Iată un exemplu mai avansat pentru a arăta cum funcționează:

gedit choice3 &

```
#!/bin/bash

read -p "Type a digit or a letter > " character
case $character in
                                # Check for letters
    [[:lower:]]  |  [[:upper:]]  )  echo  "You  typed  the  letter
$character"
                                ;;

                                # Check for digits
    [0-9]  )                        echo "You typed the digit
$character"
                                ;;

                                # Check for anything else
    * )                         echo "You did not type a letter
or a digit"
esac
```

chmod 755 choice3

./choice3

Observați modelul special „*”. Acest model se va potrivi cu orice, deci este folosit pentru a prinde cazuri care nu se potriveau cu modelele anterioare. Includerea acestui model la sfârșit este înțeleaptă, deoarece poate fi folosit pentru a detecta o intrare invalidă.

9.1. Bucle (Loops)

Ultimul tip de control al fluxului de program pe care îl vom discuta se numește buclă. Bucla este executarea în mod repetat a unei secțiuni a unui program pe baza stării de ieșire a unei comenzi. Shell-ul oferă trei comenzi pentru buclă: while, until și for.

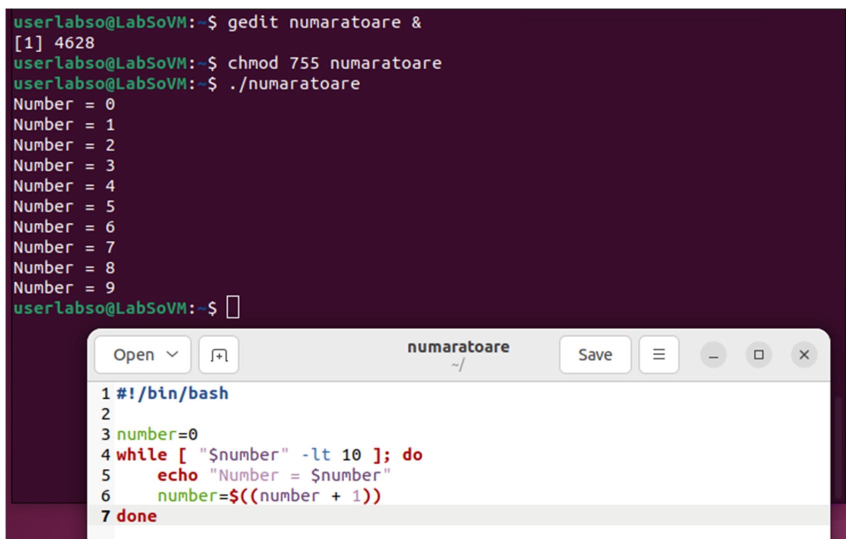
Comanda while face ca un bloc de cod să fie executat din nou și din nou, atâta timp cât starea de ieșire a unei comenzi specificate este adevărată. Iată un exemplu simplu de program care numără de la zero la nouă:

gedit numaratoare &

```
#!/bin/bash

number=0
while [ "$number" -lt 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

chmod 755 numaratoare

./numaratoare



Pe linia 3, creăm o variabilă numită Number și inițializam valoarea acesteia la 0. În continuare, începem bucla while. După cum putem vedea, am specificat o comandă care testează valoarea numărului. În exemplul nostru, testăm pentru a vedea dacă numărul are o valoare mai mică de 10.

Observați cuvântul do pe linia 4 și cuvântul done pe linia 7. Acestea includ blocul de cod care va fi repetat atâta timp cât starea de ieșire rămâne zero.

În cele mai multe cazuri, blocul de cod care se repetă trebuie să facă ceva care va schimba în cele din urmă starea de ieșire, altfel vom avea ceea ce se numește buclă fără sfârșit; adică o buclă care nu se termină niciodată.

În exemplu, blocul de cod care se repetă scoate valoarea numărului (comanda echo pe linia 5) și crește numărul cu unu pe linia 6. De fiecare dată când blocul de cod este finalizat, starea de ieșire a comenzii de testare este evaluată din nou. După a zecea iterație a buclei, numărul a fost incrementat de zece ori și comanda de testare se va termina cu o stare de ieșire diferită de zero. În acel moment, fluxul programului se reia cu instrucțiunea care urmează dupa cuvântului done. Deoarece done este ultima linie a exemplului nostru, programul se termină.

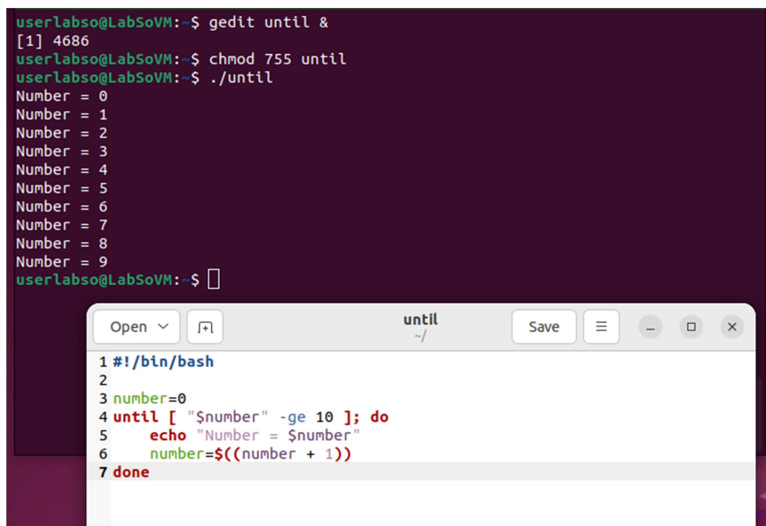
Comanda until funcționează exact în același mod, cu excepția faptului că blocul de cod este repetat atâta timp cât starea de ieșire a comenzii specificate este falsă. În exemplul de mai jos, observați cum expresia dată comenzii test a fost schimbată față de exemplul while pentru a obține același rezultat:

gedit until &

```
#!/bin/bash

number=0
until [ "$number" -ge 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

chmod 755 until

./until



9.2. Construirea unui meniu

O interfață de utilizator comună pentru programele bazate pe text este un meniu. Un meniu este o listă de opțiuni din care utilizatorul poate alege.

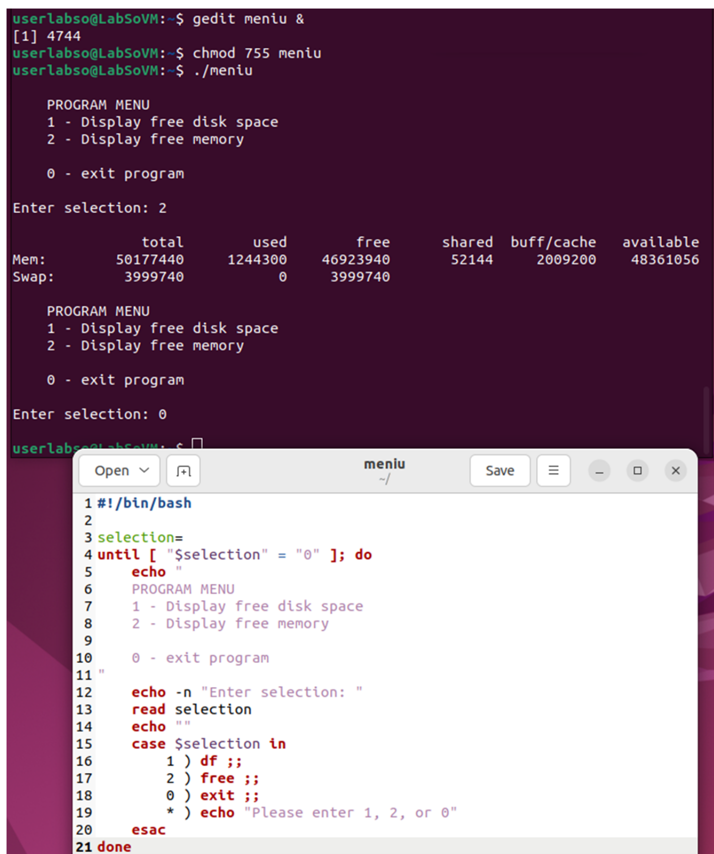
În exemplul de mai jos, folosim noile noastre cunoștințe despre bucle și cazuri pentru a construi o aplicație simplă bazată pe meniu:

gedit meniu &

```
#!/bin/bash

selection=
until [ "$selection" = "0" ]; do
    echo "
    PROGRAM MENU
    1 - Display free disk space
    2 - Display free memory

    0 - exit program
"
    echo -n "Enter selection: "
    read selection
    echo ""
    case $selection in
        1 ) df ;;
        2 ) free ;;
        0 ) exit ;;
        * ) echo "Please enter 1, 2, or 0"
    esac
done
```

chmod 755 meniu

./meniu



Scopul buclei until în acest program este de a reafișa meniul de fiecare dată când o selecție a fost finalizată. Bucla va continua până când selecția este egală cu 0, alegerea „ieșire”. Observați cum ne apărăm împotriva intrărilor de la utilizator care nu sunt alegeri valide.

Pentru ca acest program să arate mai bine atunci când rulează, îl putem îmbunătăți adăugând o funcție care cere utilizatorului să apese tasta Enter după ce fiecare selecție a fost finalizată și șterge ecranul înainte ca meniul să fie afișat din nou.

Iată exemplul îmbunătățit:

gedit meniuv2 &

```
#!/bin/bash

press_enter()
{
    echo -en "\nPress Enter to continue"
    read
    clear
}

selection=
```

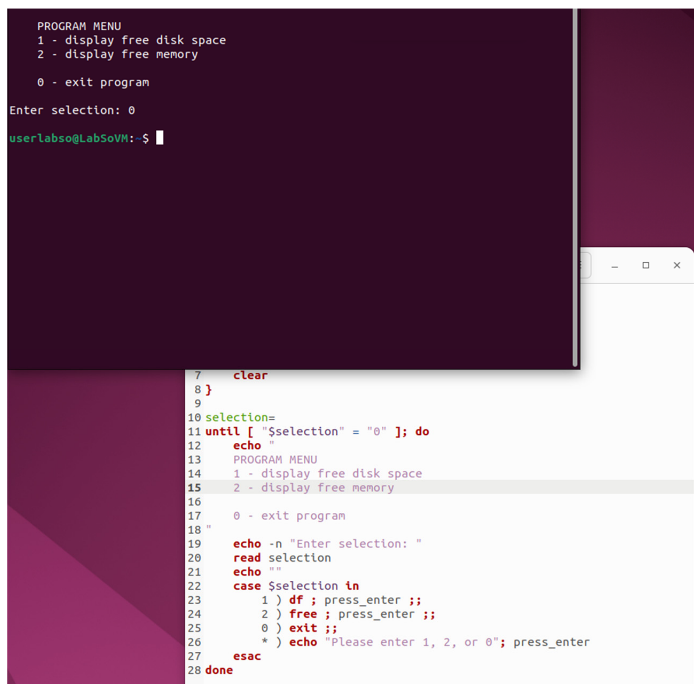
```
until [ "$selection" = "0" ]; do
    echo "
    PROGRAM MENU
    1 - display free disk space
    2 - display free memory

    0 - exit program
"
    echo -n "Enter selection: "
    read selection
    echo ""
    case $selection in
        1 ) df ; press_enter ;;
        2 ) free ; press_enter ;;
        0 ) exit ;;
        * ) echo "Please enter 1, 2, or 0"; press_enter
    esac
done
```

chmod 755 meniuv2

./meniuv2

```
userlabso@LabSoVM:~$ gedit meniuv2 &
[2] 4918
userlabso@LabSoVM:~$ chmod 755 meniuv2
[2]+  Done                    gedit meniuv2
userlabso@LabSoVM:~$ ./meniuv2

    PROGRAM MENU
    1 - display free disk space
    2 - display free memory

    0 - exit program

Enter selection: 2

              total        used        free      shared  buff/cache   available
Mem:       50177440     1231768    46903364       52596     2042308    48369444
Swap:       3999740           0     3999740

Press Enter to continue
```

```
1 #!/bin/bash
2
3 press_enter()
4 {
5     echo -en "\nPress Enter to continue"
6     read
7     clear
8 }
9
10 selection=
11 until [ "$selection" = "0" ]; do
12     echo "
13     PROGRAM MENU
14     1 - display free disk space
15     2 - display free memory
16
17     0 - exit program
18 "
19     echo -n "Enter selection: "
20     read selection
21     echo ""
22     case $selection in
23         1 ) df ; press_enter ;;
24         2 ) free ; press_enter ;;
25         0 ) exit ;;
26         * ) echo "Please enter 1, 2, or 0"; press_enter
27     esac
28 done
```



9.3. Când computerul se blochează...

Cu toții am avut experiența unei aplicații blocate. Blocarea este atunci când un program pare să se oprească brusc și să nu mai răspundă. Deși ați putea crede că programul s-a oprit, în cele mai multe cazuri, programul încă rulează, dar logica sa de program este blocată într-o buclă nesfârșită.

Imaginați-vă această situație: aveți un dispozitiv extern atașat la computer, cum ar fi un drive USB, dar ați uitat să-l porniți. Încercați să utilizați dispozitivul, dar aplicația se blochează. Când se întâmplă acest lucru, vă puteți imagina următorul dialog care se desfășoară între aplicație și interfața dispozitivului:

```
Aplicație: Ești gata?
Interfață: Dispozitivul nu este pregătit.

Aplicație: Ești gata?
Interfață: Dispozitivul nu este pregătit.

Aplicație: Ești gata?
Interfață: Dispozitivul nu este pregătit.
și așa mai departe, pentru totdeauna.
```

Software-ul bine scris încearcă să evite această situație instituind un timeout. Aceasta înseamnă că bucla urmărește și numărul de încercări sau calculează timpul în care a așteptat să se întâmple ceva. Dacă numărul de încercări sau timpul permis este depășit, bucla se încheie și programul generează o eroare și iese.

9.4. Parametrii de poziție

Când am părăsit ultima dată scriptul nostru, arăta cam așa:

```
#!/bin/bash

# sysinfo_page - A script to produce a system information HTML
file

##### Constants

TITLE="System Information for $HOSTNAME"
RIGHT_NOW="$(date +"%x %r %Z")"
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

##### Functions

system_info()
{
    echo "<h2>System release info</h2>"
    echo "<p>Function not yet implemented</p>"

}   # end of system_info


show_uptime()
{
    echo "<h2>System uptime</h2>"
    echo "<pre>"
    uptime
    echo "</pre>"

}   # end of show_uptime


drive_space()
{
    echo "<h2>Filesystem space</h2>"
    echo "<pre>"
    df
    echo "</pre>"

}   # end of drive_space


home_space()
```

```
{
    # Only the superuser can get this information

    if [ "$(id -u)" = "0" ]; then
        echo "<h2>Home directory space by user</h2>"
        echo "<pre>"
        echo "Bytes Directory"
        du -s /home/* | sort -nr
        echo "</pre>"
    fi

}   # end of home_space



##### Main

cat <<- _EOF_
  <html>
  <head>
      <title>$TITLE</title>
  </head>
  <body>
      <h1>$TITLE</h1>
      <p>$TIME_STAMP</p>
      $(system_info)
      $(show_uptime)
      $(drive_space)
      $(home_space)
  </body>
  </html>
_EOF_
```

Cele mai multe lucruri funcționează, dar mai sunt câteva funcții pe care le putem adăuga:

1. Ar trebui să putem specifica numele fișierului de ieșire pe linia de comandă, precum și să setăm un nume de fișier de ieșire implicit dacă nu este specificat niciun nume.
2. Să oferim un mod interactiv care va solicita un nume de fișier și va avertiza utilizatorul dacă fișierul există și va solicita utilizatorului să-l suprascrie.
3. Desigur, dorim să avem o opțiune de help care să afișeze un mesaj de utilizare.

Toate aceste caracteristici implică utilizarea opțiunilor și argumentelor din linia de comandă. Pentru a gestiona opțiunile pe linia de comandă, folosim o facilitate din shell numită parametri de poziție. Parametrii poziționali sunt o serie de variabile speciale ($0 până la $9) care conțin conținutul liniei de comandă.

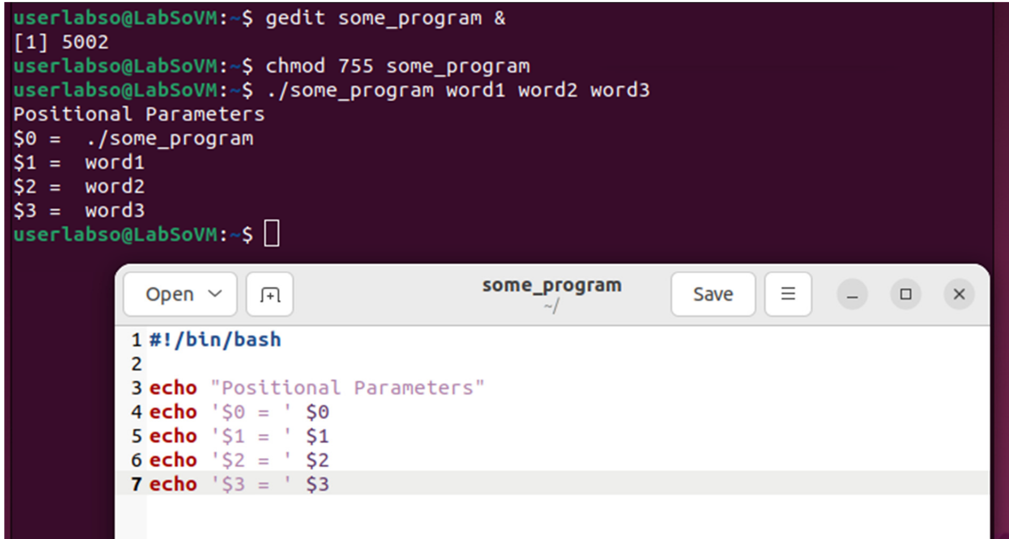
Să ne imaginăm următoarea linie de comandă:

```
gedit some_program &
```

```
#!/bin/bash

echo "Positional Parameters"
echo '$0 = ' $0
echo '$1 = ' $1
echo '$2 = ' $2
echo '$3 = ' $3
```

chmod 755 some_program

./some_program word1 word2 word3



Detectarea argumentelor liniei de comandă

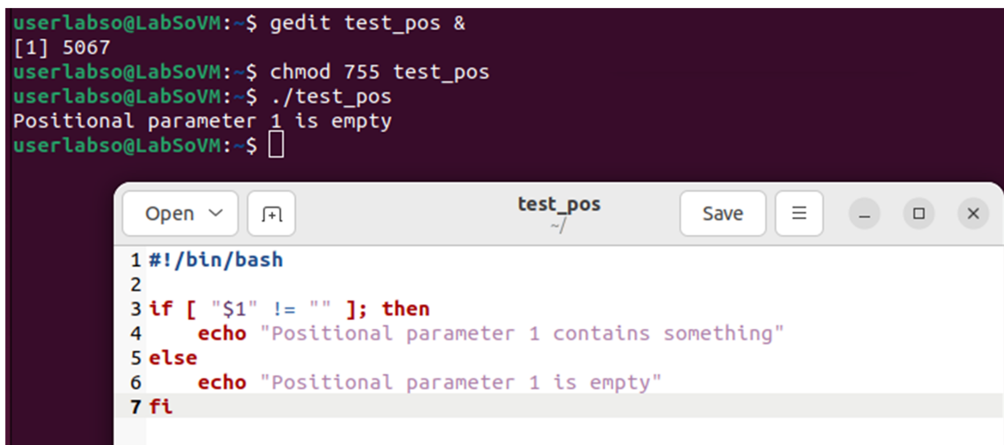
Adesea, vom dori să verificăm dacă avem argumente în linia de comandă pe baza cărora să acționăm. Există câteva moduri de a face acest lucru. În primul rând, am putea pur și simplu să verificăm dacă $1 conține ceva:

gedit test_pos &

```
#!/bin/bash

if [ "$1" != "" ]; then
    echo "Positional parameter 1 contains something"
else
    echo "Positional parameter 1 is empty"
fi
```

chmod 755 test_pos

./test_pos



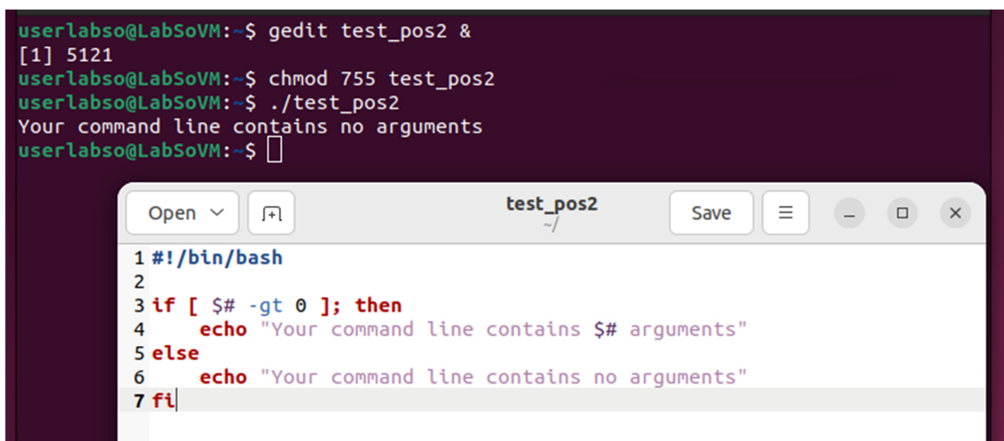
În al doilea rând, shell-ul menține o variabilă numită $# care conține numărul de elemente de pe linia de comandă în plus de numele comenzii ($0).

gedit test_pos2 &

```
#!/bin/bash

if [ $# -gt 0 ]; then
    echo "Your command line contains $# arguments"
else
    echo "Your command line contains no arguments"
fi
```

chmod 755 test_pos2

./test_pos2



9.5. Opțiuni pentru linia de comandă

După cum am discutat anterior, multe programe, în special cele din Proiectul GNU, acceptă atât opțiunile de linie de comandă scurte, cât și lungi. De exemplu, pentru a afișa un mesaj de ajutor pentru multe dintre aceste programe, putem folosi fie opțiunea „-h”, fie opțiunea mai lungă „--help”. Numele lungi ale opțiunilor sunt de obicei precedate de o liniuță dublă. Vom adopta această convenție pentru scripturile noastre.

Iată codul pe care îl vom folosi pentru a ne procesa linia de comandă (nu rulati):

```
interactive=
filename=~/sysinfo_page.html

while [ "$1" != "" ]; do
    case $1 in
        -f | --file )           shift
                                filename="$1"
                                ;;
        -i | --interactive )    interactive=1
                                ;;
        -h | --help )           usage
                                exit
                                ;;
        * )                     usage
                                exit 1
    esac
    shift
done
```

Acest cod este puțin complicat, așa că trebuie să-l explicăm.

Primele două rânduri sunt destul de ușoare. Setăm variabila interactivă să fie goală. Aceasta va indica faptul că modul interactiv nu a fost solicitat. Apoi setăm variabila nume de fișier să conțină un nume de fișier implicit. Dacă nu este specificat nimic altceva pe linia de comandă, va fi folosit acest nume de fișier.

După ce aceste două variabile sunt setate, avem setări implicite, în cazul în care utilizatorul nu specifică nicio opțiune.

În continuare, construim o buclă while care va parcurge toate elementele de pe linia de comandă și va procesa fiecare cu majuscule. Cazul va detecta fiecare opțiune posibilă și o va procesa în consecință.

Acum partea dificilă. Cum funcționează acea buclă? Se bazează pe magia shift.

shift este un alt încorporat în shell care operează pe parametrii poziționali. De fiecare dată când invocăm shift, aceasta „schimbă” toți parametrii de poziție în jos cu unul. $2 devine $1, $3 devine $2, $4 devine $3 și așa mai departe. Încearca asta:

```
gedit shift_test &
```

```
#!/bin/bash

echo "You start with $# positional parameters"

# Loop until all parameters are used up
while [ "$1" != "" ]; do
    echo "Parameter 1 equals $1"
    echo "You now have $# positional parameters"

    # Shift all the parameters down by one
    shift

done
```

chmod 755 shift_test

./shift_test

./shift_test word1 word2 word3

```
userlabso@LabSoVM:~$ gedit shift_test &
[1] 5183
userlabso@LabSoVM:~$ chmod 755 shift_test
userlabso@LabSoVM:~$ ./shift_test
You start with 0 positional parameters
userlabso@LabSoVM:~$ ./shift_test word1 word2 word3
You start with 3 positional parameters
Parameter 1 equals word1
You now have 3 positional parameters
Parameter 1 equals word2
You now have 2 positional parameters
Parameter 1 equals word3
You now have 1 positional parameters
userlabso@LabSoVM:~$
```

```
shift_test
~/
1 #!/bin/bash
2
3 echo "You start with $# positional parameters"
4
5 # Loop until all parameters are used up
6 while [ "$1" != "" ]; do
7     echo "Parameter 1 equals $1"
8     echo "You now have $# positional parameters"
9
10    # Shift all the parameters down by one
11    shift
12
13 done
```

9.6. Obținerea argumentului unei opțiuni

Opțiunea noastră „-f" necesită un nume de fișier valid ca argument. Folosim din nou shift pentru a obține următorul articol din linia de comandă și a-l atribui numelui fișierului. Mai târziu va trebui să verificăm conținutul numelui fișierului pentru a ne asigura că este valid.

9.7. Integrarea procesorului de linie de comandă în script

Va trebui să mutăm câteva lucruri și să adăugăm o funcție de utilizare pentru a integra această nouă rutină în scriptul nostru. Vom adăuga, de asemenea, un cod de test pentru a verifica dacă procesorul liniei de comandă funcționează corect. Scriptul nostru acum arată astfel:

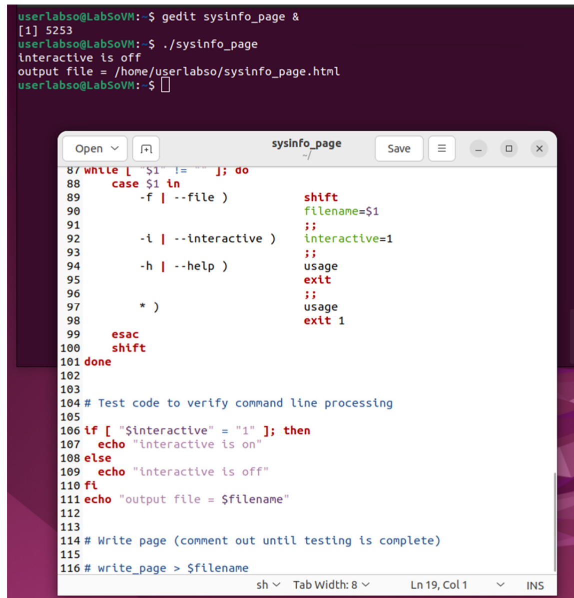
gedit sysinfo_page &

```
#!/bin/bash

# sysinfo_page - A script to produce a system information HTML file

##### Constants

TITLE="System Information for $HOSTNAME"
RIGHT_NOW="$(date +"%x %r %Z")"
TIME_STAMP="Updated on $RIGHT_NOW by $USER"

##### Functions

system_info()
{
    echo "<h2>System release info</h2>"
    echo "<p>Function not yet implemented</p>"

}   # end of system_info

show_uptime()
{
    echo "<h2>System uptime</h2>"
    echo "<pre>"
    uptime
    echo "</pre>"

}   # end of show_uptime

drive_space()
{
    echo "<h2>Filesystem space</h2>"
    echo "<pre>"
    df
    echo "</pre>"

}   # end of drive_space

home_space()
{
    # Only the superuser can get this information

    if [ "$(id -u)" = "0" ]; then
        echo "<h2>Home directory space by user</h2>"
        echo "<pre>"
        echo "Bytes Directory"
        du -s /home/* | sort -nr
        echo "</pre>"
    fi
```

```
}   # end of home_space

write_page()
{
    cat <<- _EOF_
    <html>
        <head>
        <title>$TITLE</title>
        </head>
        <body>
        <h1>$TITLE</h1>
        <p>$TIME_STAMP</p>
        $(system_info)
        $(show_uptime)
        $(drive_space)
        $(home_space)
        </body>
    </html>
_EOF_

}

usage()
{
    echo "usage: sysinfo_page [[[-f file ] [-i]] | [-h]]"
}

##### Main

interactive=
filename=~/sysinfo_page.html

while [ "$1" != "" ]; do
    case $1 in
        -f | --file )           shift
                                filename=$1
                                ;;
        -i | --interactive )    interactive=1
                                ;;
        -h | --help )           usage
                                exit
                                ;;
        * )                     usage
                                exit 1
    esac
    shift
done

# Test code to verify command line processing

if [ "$interactive" = "1" ]; then
  echo "interactive is on"
else
  echo "interactive is off"
fi
echo "output file = $filename"

# Write page (comment out until testing is complete)
```

```
# write_page > $filename
```

./sysinfo_page



9.8. Adăugarea modului interactiv

Modul interactiv este implementat cu următorul cod (inlocuiti in fisierul sysinfo_page partea corespunzătoare, și modificați param. interactiv la 1):

```
if [ "$interactive" = "1" ]; then

    response=

    read -p "Enter name of output file [$filename] > " response
    if [ -n "$response" ]; then
        filename="$response"
    fi

    if [ -f $filename ]; then
        echo -n "Output file exists. Overwrite? (y/n) > "
        read response
        if [ "$response" != "y" ]; then
            echo "Exiting program."
            exit 1
        fi
    fi
fi
```

./sysinfo_page

```
userlabso@LabSoVM:~$ ./sysinfo_page
Enter name of output file [/home/userlabso/sysinfo_page.html] > sysinfo_page.html
Output file exists. Overwrite? (y/n) > y
userlabso@LabSoVM:~$
```

```
sysinfo_page
~/
 83
 84 interactive=1
 85 filename=~/sysinfo_page.html
 86
 87 while [ "$1" != "" ]; do
 88     case $1 in
 89         -f | --file )           shift
 90                                 filename=$1
 91                                 ;;
 92         -i | --interactive )    interactive=1
 93                                 ;;
 94         -h | --help )           usage
 95                                 exit
 96                                 ;;
 97         * )                     usage
 98                                 exit 1
 99     esac
100     shift
101 done
102
103 # Test code to verify command line processing
104
105 if [ "$interactive" = "1" ]; then
106
107     response=
108
109     read -p "Enter name of output file [$filename] > " response
110     if [ -n "$response" ]; then
111         filename="$response"
112     fi
113
114     if [ -f $filename ]; then
115         echo -n "Output file exists. Overwrite? (y/n) > "
116         read response
117         if [ "$response" != "y" ]; then
118             echo "Exiting program."
119             exit 1
120         fi
121     fi
122 fi
123 # Write page (comment out until testing is complete)
124
125 write_page > $filename
sh   Tab Width: 8   Ln 125, Col 23   INS
```

În primul rând, verificăm dacă modul interactiv este activat, altfel nu avem ce face. Apoi, îi cerem utilizatorului numele fișierului. Observați modul în care este formulată solicitarea:

```
userlabso@LabSoVM:~$ ./sysinfo_page
Enter name of output file [/home/userlabso/sysinfo_page.html] > sysinfo_page.html
```

Afișăm valoarea curentă a numelui fișierului, deoarece, așa cum este codificată această rutină, dacă utilizatorul apasă doar tasta Enter, se va folosi valoarea implicită a numelui fișierului. Acest lucru se realizează în următoarele două rânduri în care se verifică valoarea răspunsului. Dacă răspunsul nu este gol, atunci numelui fișierului i se atribuie valoarea răspunsului. În caz contrar, numele fișierului rămâne neschimbat, păstrându-și valoarea implicită.

După ce avem numele fișierului de ieșire, verificăm dacă acesta există deja. Dacă o face, solicităm utilizatorului. Dacă răspunsul utilizatorului nu este „y”, renunțăm și ieșim, altfel putem continua.

Ca să verificăm dacă într-adevăr fișierul nostru a fost rescris:

ll | grep sysinfo_page.html

```
userlabso@LabSoVM:~$ ll | grep sysinfo_page.html
-rw-rw-r--  1 userlabso userlabso  933 nov 26 17:13 sysinfo_page.html
userlabso@LabSoVM:~$
```

10. SCRIPTURI SHELL ÎN SISTEME LINUX - CONTROLUL FLUXULUI DE EXECUȚIE: BUCLE CU FOR

Acum că am învățat despre parametrii poziționali, este timpul să acoperim declarația de control al fluxului rămasă, for. Ca while și until, for este folosit pentru a construi bucle, pentru lucruri de genul acesta:

```
for variable in words; do
    commands
done
```

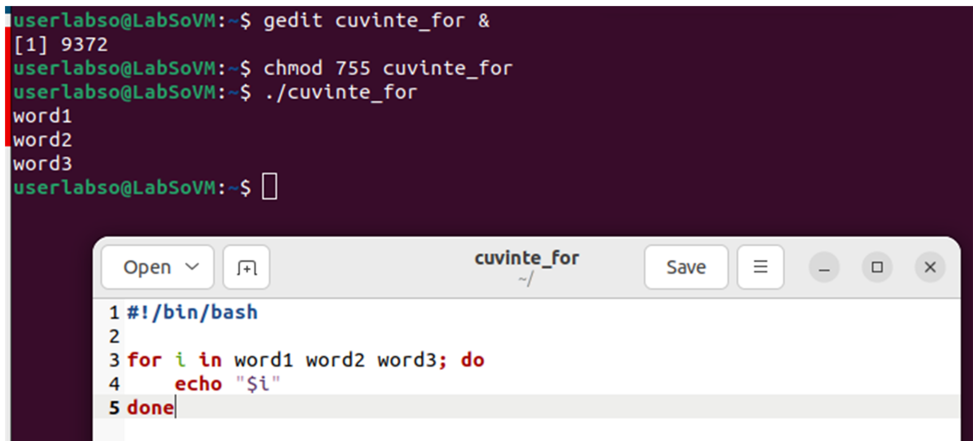
În esență, for atribuie un cuvânt din lista de cuvinte variabilei specificate, execută comenzile și repetă acest lucru până când toate cuvintele au fost epuizate. Iată un exemplu:

gedit cuvinte_for &

```
#!/bin/bash

for i in word1 word2 word3; do
    echo "$i"
done
```

chmod 755 cuvinte_for

./cuvinte_for



În acest exemplu, variabilei „i” i se atribuie șirul „word1”, apoi se execută instrucțiunea echo „$i”, apoi variabilei „i” i se atribuie șirul „word2”, iar instrucțiunea echo „$i” este executată și așa mai departe, până când toate cuvintele din lista de cuvinte au fost atribuite.

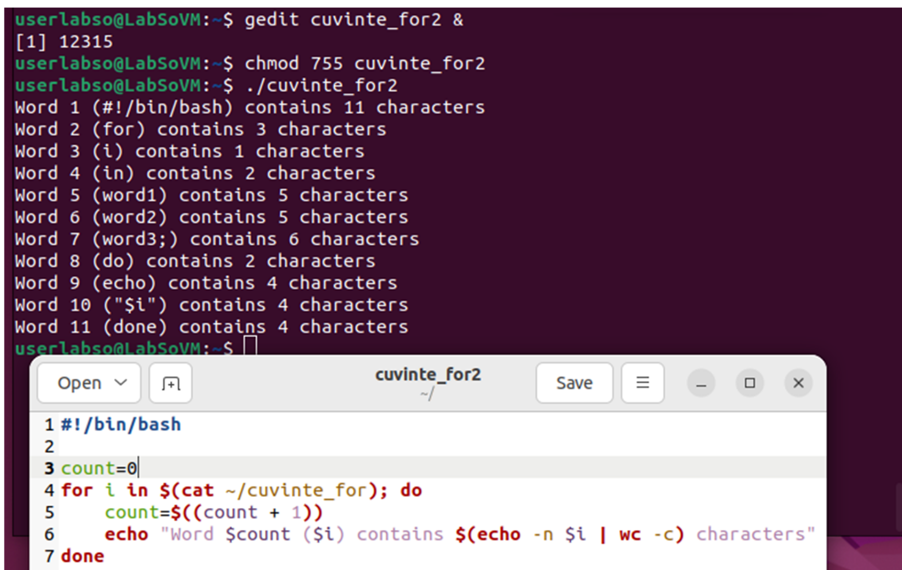
Lucrul interesant despre for este numeroasele moduri în care putem construi lista de cuvinte. Pot fi folosite tot felul de expansiuni. În exemplul următor, vom construi lista de cuvinte folosind înlocuirea comenzii:

gedit cuvinte_for2 &

```
#!/bin/bash

count=0
for i in $(cat ~/cuvinte_for); do
    count=$((count + 1))
    echo "Word $count ($i) contains $(echo -n $i | wc -c) characters"
done
```

chmod 755 cuvinte_for2

./cuvinte_for2



Aici luăm fișierul cuvinte_for și numărăm cuvintele din fișier și numărul de caractere al fiecarui cuvânt.

Deci, ce legătură are asta cu parametrii de poziție? Ei bine, una dintre caracteristicile for este faptul că poate folosi parametrii poziționali ca listă de cuvinte:

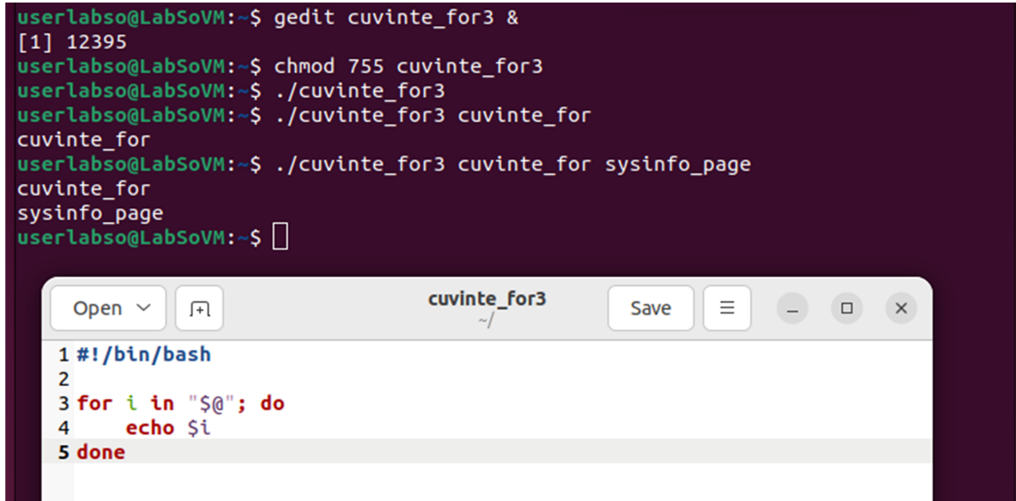
gedit cuvinte_for3 &

```
#!/bin/bash

for i in "$@"; do
    echo $i
done
```

chmod 755 cuvinte_for3

./cuvinte_for3

./cuvinte_for3 cuvinte_for sysinfo_page



Variabila shell „$@” conține lista argumentelor din linia de comandă. Această tehnică este adesea folosită pentru a procesa o listă de fișiere pe linia de comandă. Iată un alt exemplu:

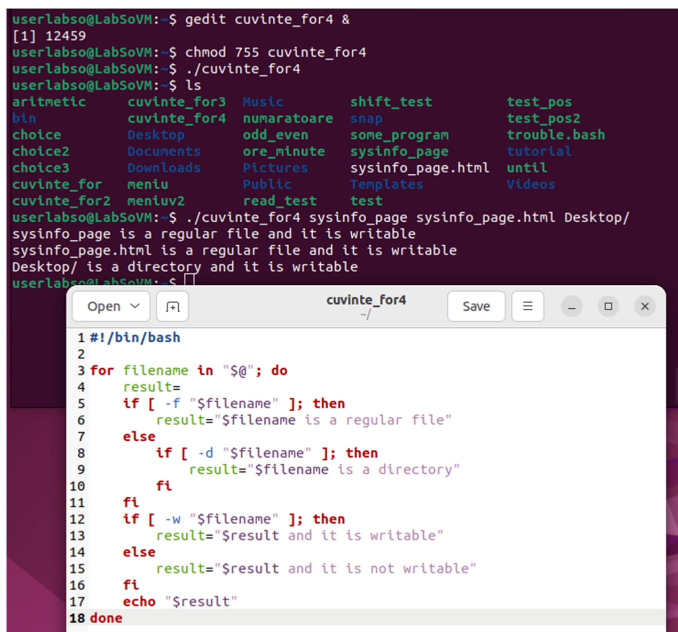
gedit cuvinte_for4 &

```
#!/bin/bash

for filename in "$@"; do
    result=
    if [ -f "$filename" ]; then
        result="$filename is a regular file"
    else
        if [ -d "$filename" ]; then
            result="$filename is a directory"
        fi
    fi
    if [ -w "$filename" ]; then
        result="$result and it is writable"
    else
        result="$result and it is not writable"
    fi
    echo "$result"
done
```

chmod 755 cuvinte_for4

./cuvinte_for4

./cuvinte_for4 sysinfo_page sysinfo_page.html Desktop/



Încercați acest script. Dați-i o listă de fișiere sau un wildcard precum „*” pentru a vedea că funcționează.

Utilizarea lui în „$@” este atât de comună încât se presupune dacă clauza in words este omisă.

Iată un alt exemplu de script. Acesta compară fișierele din două directoare și listează fișierele din primul director care lipsesc din al doilea.

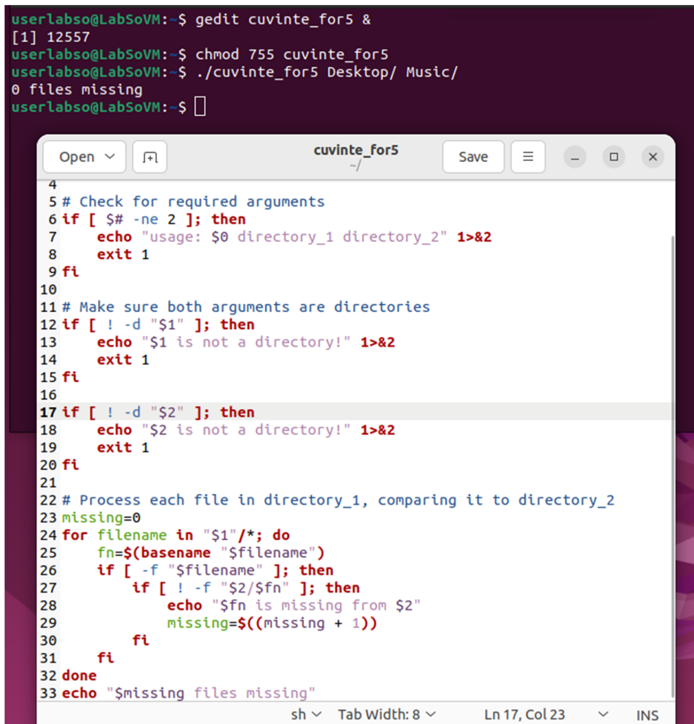
gedit cuvinte_for5 &

```
#!/bin/bash

# cmp_dir - program to compare two directories

# Check for required arguments
if [ $# -ne 2 ]; then
    echo "usage: $0 directory_1 directory_2" 1>&2
    exit 1
fi

# Make sure both arguments are directories
if [ ! -d "$1" ]; then
    echo "$1 is not a directory!" 1>&2
    exit 1
fi

if [ ! -d "$2" ]; then
    echo "$2 is not a directory!" 1>&2
    exit 1
fi

# Process each file in directory_1, comparing it to directory_2
missing=0
for filename in "$1"/*; do
```

```
    fn=$(basename "$filename")
    if [ -f "$filename" ]; then
        if [ ! -f "$2/$fn" ]; then
            echo "$fn is missing from $2"
            missing=$((missing + 1))
        fi
    fi
done
echo "$missing files missing"
```

chmod 755 cuvinte_for5

./cuvinte_for5



Acum, la scopul nostru. Vrem să îmbunătățim funcția home_space din sysinfo_page pentru a scoate mai multe informații. Amintiți-vă că versiunea noastră anterioară arăta astfel:

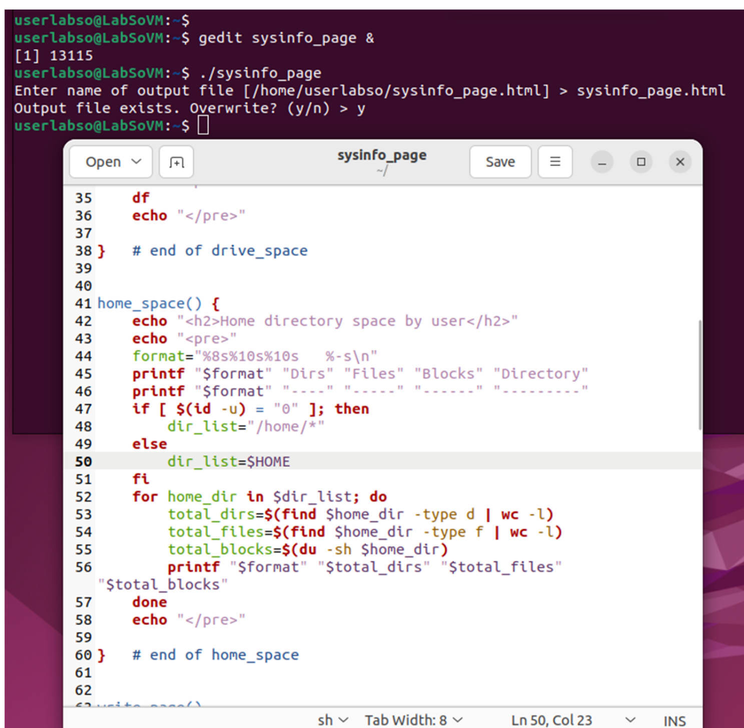
```
home_space()
{
    # Only the superuser can get this information

    if [ "$(id -u)" = "0" ]; then
        echo "<h2>Home directory space by user</h2>"
        echo "<pre>"
        echo "Bytes Directory"
        du -s /home/* | sort -nr
        echo "</pre>"
    fi
}   # end of home_space
```

Iată noua versiune:

gedit sysinfo_page &

```
home_space() {
    echo "<h2>Home directory space by user</h2>"
    echo "<pre>"
    format="%8s%10s%10s   %-s\n"
    printf "$format" "Dirs" "Files" "Blocks" "Directory"
    printf "$format" "----" "-----" "------" "---------"
    if [ $(id -u) = "0" ]; then
        dir_list="/home/*"
    else
        dir_list=$HOME
    fi
    for home_dir in $dir_list; do
        total_dirs=$(find $home_dir -type d | wc -l)
        total_files=$(find $home_dir -type f | wc -l)
        total_blocks=$(du -sh $home_dir)
        printf "$format" "$total_dirs" "$total_files" "$total_blocks"
    done
    echo "</pre>"
}   # end of home_space
```

./sysinfo_page



Această versiune îmbunătățită introduce o nouă comandă printf, care este folosită pentru a produce ieșiri formatate în funcție de conținutul unui șir de format. printf provine din limbajul

de programare C și a fost implementat în multe alte limbaje de programare, inclusiv C++, Perl, awk, java, PHP și, desigur, bash.

Introducem și comanda find. find este folosit pentru a căuta fișiere sau directoare care îndeplinesc anumite criterii. În funcția home_space, folosim find pentru a lista directoarele și fișierele obișnuite din fiecare director principal. Folosind comanda wc, numărăm numărul de fișiere și directoare găsite.

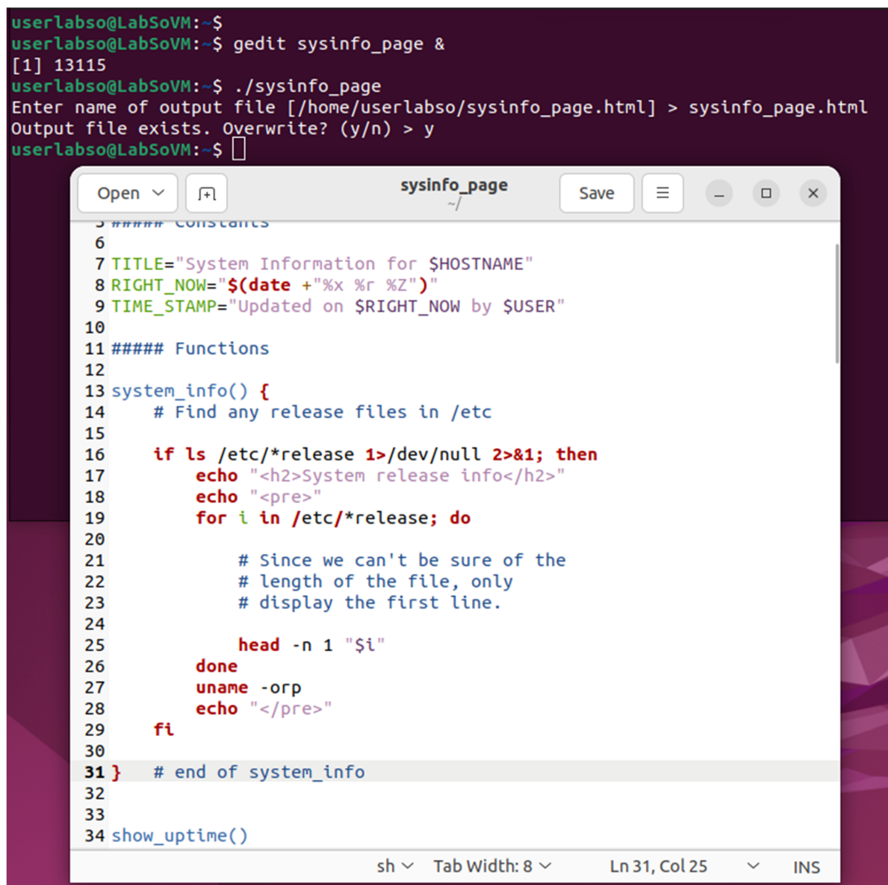
Lucrul cu adevărat interesant despre home_space este modul în care ne ocupăm de problema accesului superutilizatorului. Observați că testăm pentru superutilizatorul cu id și, în funcție de rezultatul testului, atribuim șiruri diferite variabilei dir_list, care devine lista de cuvinte pentru bucla for care urmează. În acest fel, dacă un utilizator obișnuit rulează scriptul, va fi listat doar directorul său principal.

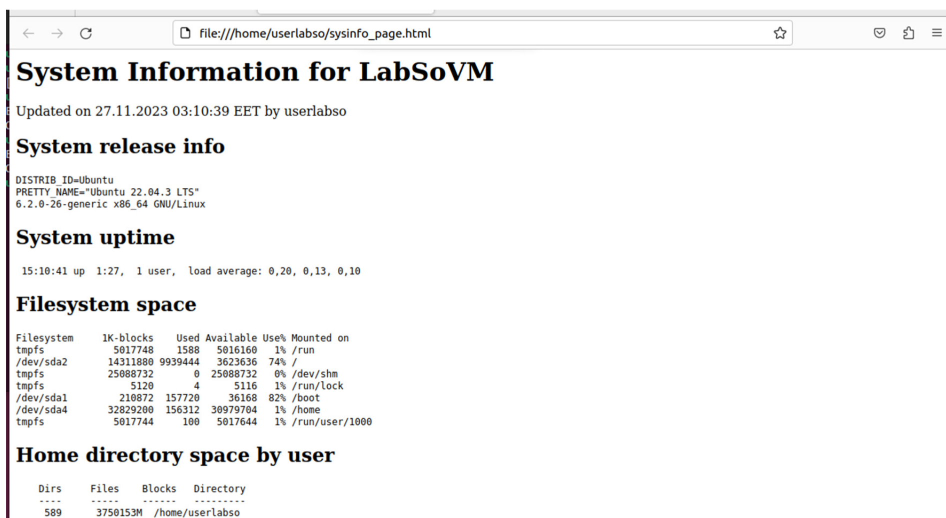
O altă funcție care poate folosi o buclă for este funcția noastră system_info neterminată. O putem construi astfel:

gedit sysinfo_page &

```
system_info() {
    # Find any release files in /etc

    if ls /etc/*release 1>/dev/null 2>&1; then
        echo "<h2>System release info</h2>"
        echo "<pre>"
        for i in /etc/*release; do

            # Since we can't be sure of the
            # length of the file, only
            # display the first line.

            head -n 1 "$i"
        done
        uname -orp
        echo "</pre>"
    fi

}   # end of system_info
```

./sysinfo_page





În această funcție, determinăm mai întâi dacă există fișiere de lansare de procesat. Fișierele de lansare conțin numele vânzătorului și versiunea distribuției. Acestea se află în

directorul /etc. Pentru a le detecta, executăm o comandă ls și aruncăm toată ieșirea acesteia. Ne interesează doar starea de ieșire. Va fi adevărat dacă se găsesc fișiere.

Apoi, scoatem HTML-ul pentru această secțiune a paginii, deoarece acum știm că există fișiere de lansare de procesat. Pentru a procesa fișierele, începem o buclă for pentru a acționa asupra fiecăruia. În interiorul buclei, folosim comanda head pentru a returna prima linie a fiecărui fișier.

În cele din urmă, folosim comanda uname cu opțiunile „o”, „r” și „p” pentru a obține câteva informații suplimentare despre sistem.

10.1. Erori, semnale și capcane

Ne vom uita la gestionarea erorilor în timpul execuției scriptului. Diferența dintre un program slab și unul bun este adesea măsurată în termeni de robustețe a programului. Adică capacitatea programului de a face față situațiilor în care ceva nu merge bine.

10.2. Stare de ieșire

După cum ne amintim din laboratoarele anterioare, fiecare program bine scris returnează o stare de ieșire când se termină. Dacă un program se termină cu succes, starea de ieșire va fi zero. Dacă starea de ieșire este altceva decât zero, atunci programul a eșuat într-un fel.

Este foarte important să verificăm starea de ieșire a programelor pe care le apelăm în scripturile noastre. De asemenea, este important ca scripturile noastre să returneze o stare de ieșire semnificativă când se termină. A fost odată un administrator de sistem Unix care a scris un script pentru un sistem de producție care conținea următoarele 2 linii de cod (NU RULATI!):

```
# Example of a really bad idea

cd "$some_directory"
rm *
```

De ce este un mod atât de prost de a face asta? Nu este, dacă totul merge bine. Cele două linii schimbă directorul de lucru spre cel conținut în $some_directory și șterge fișierele din acel director. Acesta este comportamentul intenționat. Dar ce se întâmplă dacă directorul numit în $some_directory nu există? În acest caz, comanda cd va eșua și scriptul execută comanda rm în directorul de lucru curent. Acesta nu comportamentul dorit!

Apropo, scriptul nefericitului administrator de sistem a suferit chiar acest eșec și a distrus o mare parte a unui sistem de producție important. Nu lăsa să ți se întâmple asta!

Problema cu scriptul a fost că nu a verificat starea de ieșire a comenzii cd înainte de a continua cu comanda rm.

10.3. Verificarea stării de ieșire

Există mai multe moduri prin care putem obține și răspunde la starea de ieșire a unui program. În primul rând, putem examina conținutul variabilei de mediu $?. $? va contine starea de iesire a ultimei comenzi executate. Putem vedea aceast lucru cu următoarele:

```
true; echo $?
```

false; echo $?

```
userlabso@LabSoVM:~$ true; echo $?
0
userlabso@LabSoVM:~$ false; echo $?
1
```

Înainte de a continua, pentru că vom lucra cu comanda rm, haideți să facem o copie a directorului curent:

*tar -czvf rezerva.tar.gz ./**

Comenzile true și false sunt programe care nu fac nimic decât să returneze o stare de ieșire de zero și, respectiv, unu. Folosindu-le, putem vedea cum variabila de mediu $? conține starea de ieșire a programului anterior.

Deci, pentru a verifica starea de ieșire, am putea scrie scriptul astfel:

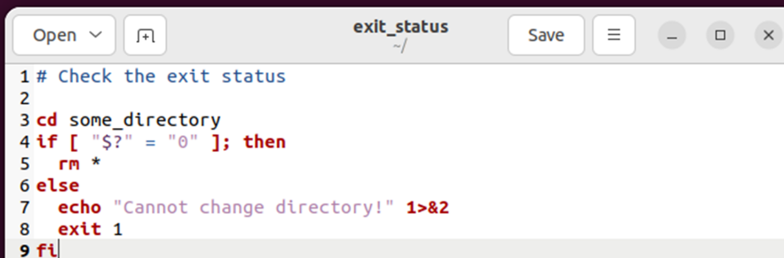
gedit exit_status &

```
#!/bin/bash
# Check the exit status

cd some_directory
if [ "$?" = "0" ]; then
  rm *
else
  echo "Cannot change directory!" 1>&2
  exit 1
fi
```

chmod 755 exit_status

./exit_status

```
userlabso@LabSoVM:~$ gedit exit_status &
[1] 14354
userlabso@LabSoVM:~$ chmod 755 exit_status
userlabso@LabSoVM:~$ ./exit_status
./exit_status: line 3: cd: some_directory: No such file or directory
Cannot change directory!
userlabso@LabSoVM:~$
```



În această versiune, examinăm starea de ieșire a comenzii cd și dacă nu este zero, tipărim un mesaj de eroare la eroare standard și încheiem scriptul cu o stare de ieșire de 1.

Deși aceasta este o soluție funcțională la problemă, există metode mai inteligente care ne vor scuti de tastarea. Următoarea abordare pe care o putem încerca este să folosim direct instrucțiunea if, deoarece evaluează starea de ieșire a comenzilor care le sunt date.

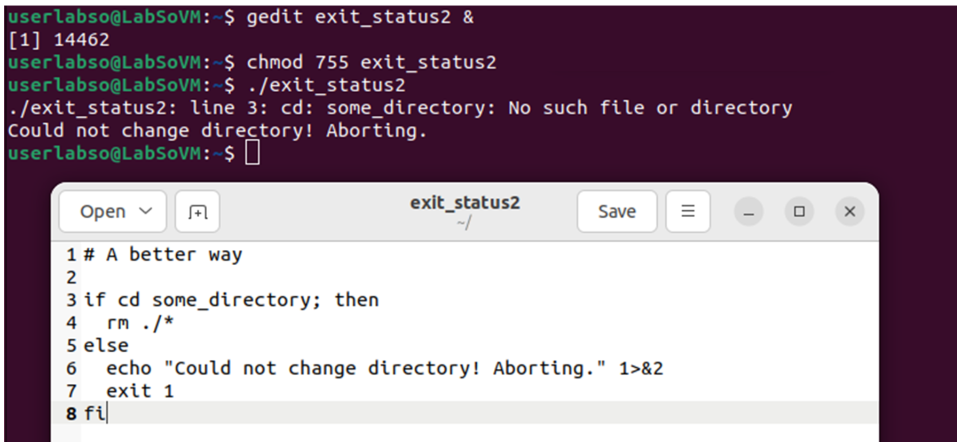
Folosind if, am putea scrie astfel:

gedit exit_status2 &

```
#!/bin/bash
# A better way

if cd some_directory; then
  rm ./*
else
  echo "Could not change directory! Aborting." 1>&2
  exit 1
fi
```

chmod 755 exit_status2

./exit_status2



Aici verificăm dacă comanda cd este executată cu succes. Abia atunci rm este executat; în caz contrar, este afişat un mesaj de eroare şi programul va ieşi cu un cod de 1, indicând că a apărut o eroare.

Observați și cum am schimbat ținta comenzii rm din „*” în „./*”. Aceasta este o măsură de siguranță. Motivul este puțin subtil și are de-a face cu modul în care sistemele Unix numesc fişierele. Deoarece este posibil să includem aproape orice caracter într-un nume de fișier, trebuie să plasăm numele de fișiere care încep cu cratime, deoarece dvs. ar putea fi interpretate ca opțiuni de comandă după extinderea wildcardului. De exemplu, dacă a existat un fișier numit -rf în director, acesta ar putea determina rm să facă lucruri neplăcute. Este o idee bună să includeți întotdeauna „./” înaintea asteriscurilor principale în scripturi.

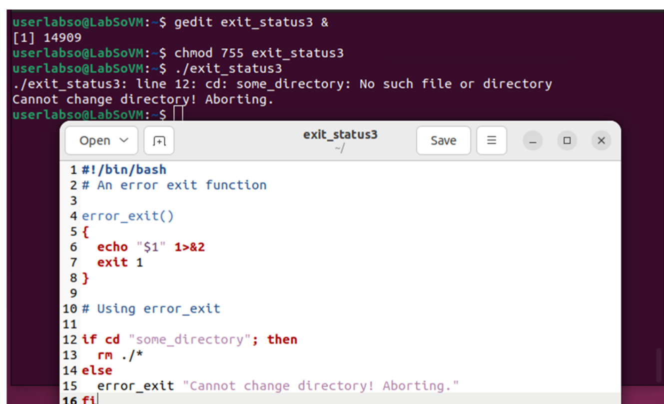
10.4. Funcții pentru ieșiri de eroare

Deoarece vom verifica frecvent erorile în programele noastre, este logic să scriem o funcție care să afișeze mesaje de eroare. Acest lucru va economisi mai multă tastare și va promova lenea.

gedit exit_status3 &

```
#!/bin/bash
# An error exit function
error_exit()
{
  echo "$1" 1>&2
  exit 1
}
# Using error_exit
if cd "some_directory"; then
  rm ./*
else
  error_exit "Cannot change directory! Aborting."
fi
```

```
chmod 755 exit_status3
./exit_status3
```



10.5. Listele AND și OR

În cele din urmă, ne putem simplifica și mai mult scriptul utilizând operatorii de control AND și OR. Pentru a explica cum funcționează, iată un citat din pagina de manual bash:

„Operatorii de control && și || denotă liste AND și, respectiv, liste OR. O listă AND are forma command1 && command2 command2 este executată dacă și numai dacă command1 returnează o stare de ieșire zero. O listă OR are forma command1 || command2 command2 este executată dacă și numai dacă command1 returnează o stare de ieșire diferită de zero. Starea de ieșire a listelor AND și OR este starea de ieșire a ultimei comenzi executate din listă."

Din nou, putem folosi comenzile true și false pentru a vedea cum funcționează:

```
true || echo "echo executed"
false || echo "echo executed"
true && echo "echo executed"
false && echo "echo executed"
```

```
userlabso@LabSoVM:~$ true || echo "echo executed"
userlabso@LabSoVM:~$ false || echo "echo executed"
echo executed
userlabso@LabSoVM:~$ true && echo "echo executed"
echo executed
userlabso@LabSoVM:~$ false && echo "echo executed"
userlabso@LabSoVM:~$
```

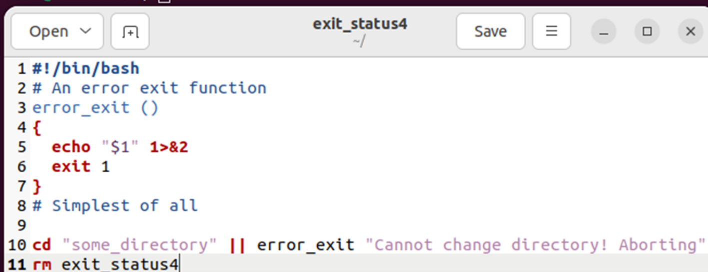
Folosind această tehnică, putem scrie o versiune și mai simplă:

gedit exit_status4 &

```
#!/bin/bash
# An error exit function

error_exit()
{
  echo "$1" 1>&2
  exit 1
}
# Simplest of all

cd "some_directory" || error_exit "Cannot change directory! Aborting"
rm exit_status4
```

chmod 755 exit_status4

./exit_status4



Dacă nu este necesară o ieșire în caz de eroare, atunci putem chiar face acest lucru:

gedit exit_status5 &

```
#!/bin/bash
# An error exit function

error_exit()
{
  echo "$1" 1>&2
```

```
  exit 1
}
# Another way to do it if exiting is not desired

cd "some_directory" && rm ./exit_status5
```

chmod 755 exit_status5

./exit_status5

```
userlabso@LabSoVM:~$ gedit exit_status5 &
[1] 15209
userlabso@LabSoVM:~$ chmod 755 exit_status5
userlabso@LabSoVM:~$ ./exit_status5
./exit_status5: line 10: cd: some_directory: No such file or directory
userlabso@LabSoVM:~$
```

```
Open   exit_status5 ~/   Save
1 #!/bin/bash
2 # An error exit function
3 error_exit ()
4 {
5   echo "$1" 1>&2
6   exit 1
7 }
8 # Another way to do it if exiting is not desired
9
10 cd "some_directory" && rm ./exit_status5
```

Trebuie să subliniem că, chiar și cu apărarea împotriva erorilor pe care am introdus-o în exemplul nostru pentru utilizarea cd, acest cod este încă vulnerabil la o eroare comună de programare, și anume, ce se întâmplă dacă numele variabilei care conține numele directorul este scris greșit? În acest caz, shell-ul va interpreta variabila ca fiind goală și cd va reuși, dar va schimba directoarele în directorul principal al utilizatorului, așa că aveți grijă!

10.6. Îmbunătățirea funcției de ieșire cu un mesaj de eroare

Există o serie de îmbunătățiri pe care le putem aduce funcției error_exit. Este util să includeți numele programului în mesajul de eroare pentru a clarifica de unde vine eroarea. Acest lucru devine mai important pe măsură ce programele noastre devin mai complexe și începem să avem scripturi care lansează alte scripturi etc. De asemenea, rețineți că includerea variabilei de mediu LINENO, care va ajuta la identificarea exactă a liniei dintr-un script în care a apărut eroarea.

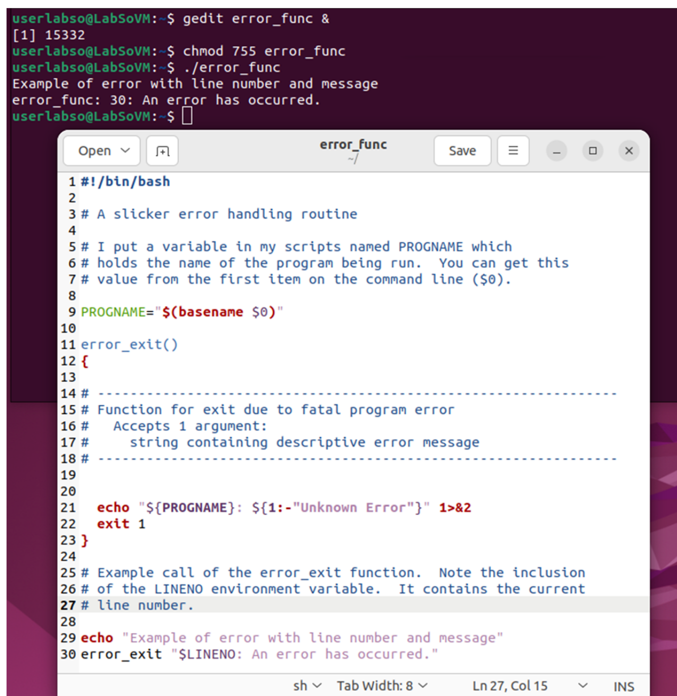
gedit error_func &

```
#!/bin/bash

# A slicker error handling routine

# I put a variable in my scripts named PROGNAME which
# holds the name of the program being run.  You can get this
# value from the first item on the command line ($0).

PROGNAME="$(basename $0)"

error_exit()
```

```
{

# -----------------------------------------------------------------
# Function for exit due to fatal program error
#   Accepts 1 argument:
#     string containing descriptive error message
# -----------------------------------------------------------------


  echo "${PROGNAME}: ${1:-"Unknown Error"}" 1>&2
  exit 1
}

# Example call of the error_exit function.  Note the inclusion
# of the LINENO environment variable.  It contains the current
# line number.

echo "Example of error with line number and message"
error_exit "$LINENO: An error has occurred."
```

chmod 755 error_func

./error_func



Utilizarea acoladelor în funcția error_exit este un exemplu de extindere a parametrilor. Putem înconjura un nume de variabilă cu acolade (ca și în cazul ${PROGNAME}) dacă trebuie să ne asigurăm că este separat de textul din jur. Unii oameni le pun în jurul fiecărei variabile din obișnuință. Această utilizare este pur și simplu o alegere de stil. A doua utilizare, ${1:-"Unknown Error"} înseamnă că dacă parametrul 1 ($1) este nedefinit, înlocuiți șirul "Unknown Error" în locul său. Folosind extinderea parametrilor, este posibil să efectuați o serie de

manipulări utile șirurilor. Mai multe informații despre extinderea parametrilor pot fi găsite în pagina de manual bash sub subiectul „EXPANSIONS”.

Erorile nu sunt singura modalitate prin care un script se poate termina în mod neașteptat. Trebuie să ne preocupăm și cu semnale. Luați în considerare următorul program:

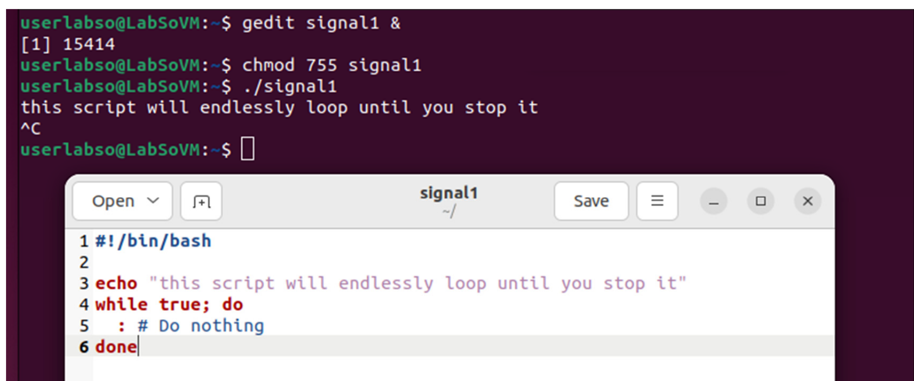
gedit signal1 &

```
#!/bin/bash

echo "this script will endlessly loop until you stop it"
while true; do
  : # Do nothing
done
```

chmod 755 signal1

./signal1

apasa Ctrl+C pentru a incheia bucla



După ce lansăm acest script, va părea să se blocheze. De fapt, ca majoritatea programelor care par să se blocheze, este într-adevăr blocat într-o buclă. În acest caz, așteaptă ca comanda adevărată să returneze o stare de ieșire diferită de zero, ceea ce nu o face niciodată. Odată pornit, scriptul va continua până când bash primește un semnal care îl va opri. Putem trimite un astfel de semnal tastând Ctrl+C, semnalul numit SIGINT (prescurtare de la SIGnal INTerrupt).

10.7. Curățenie după noi înșine

Bine, deci poate veni un semnal și poate face scriptul nostru să se termine. De ce conteaza? Ei bine, în multe cazuri nu contează și putem ignora cu siguranță semnalele, dar în unele cazuri va conta.

Să aruncăm o privire la un alt script:

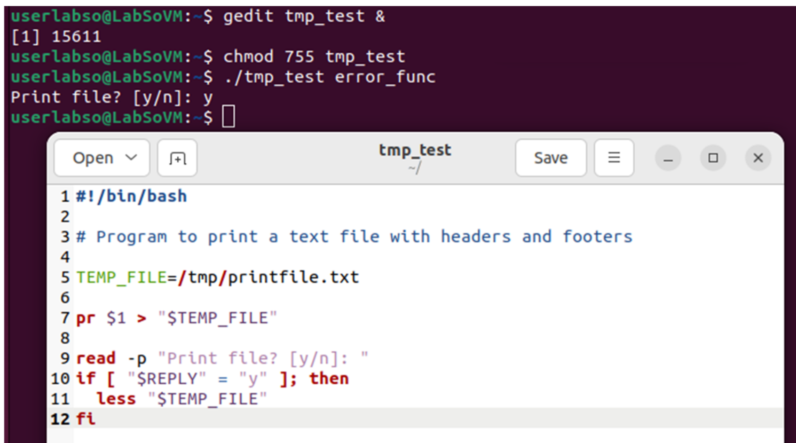
gedit tmp_test &

```
#!/bin/bash

# Program to print a text file with headers and footers

TEMP_FILE=/tmp/printfile.txt
```

```
pr $1 > "$TEMP_FILE"

read -p "Print file? [y/n]: "
if [ "$REPLY" = "y" ]; then
  less "$TEMP_FILE"
fi
```

chmod 755 tmp_test

./tmp_test error_func



Acest script procesează fișierul text specificat pe linia de comandă cu comanda pr și stochează rezultatul într-un fișier temporar. Apoi, întreabă utilizatorul dacă dorește să imprime fișierul. Dacă utilizatorul tasta „y”, atunci fișierul temporar este transmis programului less.

Desigur, acest script are o mulțime de probleme de design. Deși are nevoie de un nume de fișier transmis pe linia de comandă, nu verifică dacă a primit unul și nu verifică dacă fișierul există de fapt. Dar problema pe care vrem să ne concentrăm aici este că atunci când scriptul se termină, el lasă în urmă fișierul temporar.

Buna practică impune să ștergem fișierul temporar $TEMP_FILE atunci când scriptul se termină. Acest lucru este ușor de realizat adăugând următoarele la sfârșitul scriptului:

```
rm "$TEMP_FILE"
```

Acest lucru pare să rezolve problema, dar ce se întâmplă dacă utilizatorul tasta ctrl-c când apare promptul „Print file? [y/n]:”? Scriptul se va termina la comanda de citire și comanda rm nu este niciodată executată. În mod clar, avem nevoie de o modalitate de a răspunde la semnale precum SIGINT atunci când tasta Ctrl-c este tastata.

Din fericire, bash oferă o metodă de a efectua comenzi dacă și când sunt primite semnale.

trap

Comanda trap ne permite să executăm o comandă atunci când scriptul nostru primește un semnal. Funcționează așa:

```
trap arg signals
```

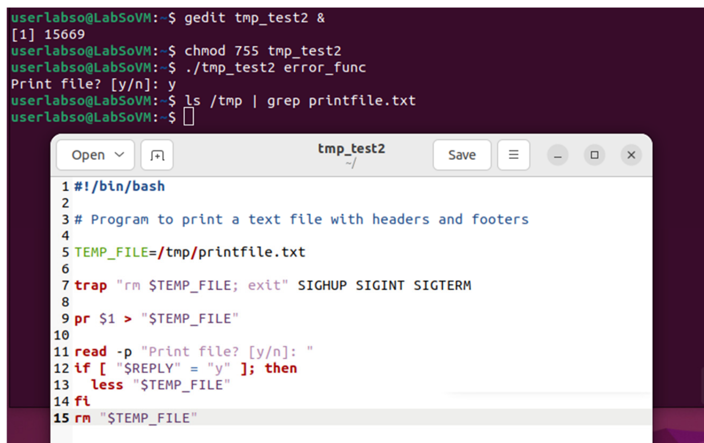
„signals" este o listă de semnale de interceptat, iar „arg" este o comandă de executat atunci când unul dintre semnale este recepționat. Pentru scriptul nostru de printare, am putea trata problema semnalului în felul acesta:

gedit tmp_test2 &

```
#!/bin/bash

# Program to print a text file with headers and footers

TEMP_FILE=/tmp/printfile.txt

trap "rm $TEMP_FILE; exit" SIGHUP SIGINT SIGTERM

pr $1 > "$TEMP_FILE"

read -p "Print file? [y/n]: "
if [ "$REPLY" = "y" ]; then
  less "$TEMP_FILE"
fi
rm "$TEMP_FILE"
```

chmod 755 tmp_test2

./tmp_test2 error_func



Aici am adăugat o comandă trap care va executa „rm $TEMP_FILE" dacă este primit oricare dintre semnalele enumerate. Cele trei semnale enumerate sunt cele mai comune pe care majoritatea scripturilor le vor întâlni, dar există multe altele care pot fi specificate. Pentru o listă completă, tastați „trap -l". Pe lângă listarea semnalelor după nume, puteți să le specificați alternativ după număr.

10.8. Semnalul 9

Există un semnal pe care nu îl puteți capta: SIGKILL sau semnalul 9. Nucleul oprește imediat orice proces caruia ii este trimis acest semnal și nu este efectuată nicio manipulare a semnalului. Deoarece va termina întotdeauna un program care este blocat sau încurcat în alt mod, este tentant să credem că este calea de ieșire ușoară atunci când trebuie să obținem ceva pentru a opri și a pleca. Există adesea referiri la următoarea comandă care trimite semnalul SIGKILL:

```
kill -9
```

Cu toate acestea, în ciuda aparentei sale ușurințe, trebuie să ne amintim că atunci când trimitem acest semnal, aplicația nu face nicio procesare. Adesea, acest lucru este OK, dar cu multe programe nu este. În special, multe programe complexe (și unele nu atât de complexe) creează fișiere de blocare pentru a preveni rularea simultană a mai multor copii ale programului. Când unui program care utilizează un fișier de blocare i se trimite un SIGKILL, nu are șansa de a elimina fișierul de blocare când acesta se încheie. Prezența fișierului de blocare va împiedica repornirea programului până când fișierul de blocare este eliminat manual. Fii avertizat. Utilizați SIGKILL ca ultimă soluție.

10.9. O funcție de curățare

În timp ce comanda trap a rezolvat problema, putem vedea că are unele limitări. Cel mai important, va accepta doar un singur șir care conține comanda care urmează să fie executată atunci când semnalul este recepționat. Am putea deveni deștepți și să folosim „;” și puneți mai multe comenzi în șir pentru a obține un comportament mai complex, dar sincer, este urât. O modalitate mai bună ar fi să creăm o funcție care este apelată atunci când dorim să realizăm orice acțiune la sfârșitul unui script. În scopurile noastre, vom numi această funcție clean_up.

gedit clean_up &

```
#!/bin/bash

# Program to print a text file with headers and footers

TEMP_FILE=/tmp/printfile.txt

clean_up() {

  # Perform program exit housekeeping
  rm "$TEMP_FILE"
  exit
}

trap clean_up SIGHUP SIGINT SIGTERM

pr $1 > "$TEMP_FILE"

read -p "Print file? [y/n]: "
if [ "$REPLY" = "y" ]; then
  less "$TEMP_FILE"
fi
clean_up
```

chmod 755 clean_up

./clean_up tmp_test

```
userlabso@LabSoVM:~$ gedit clean_up &
[1] 15964
userlabso@LabSoVM:~$ chmod 755 clean_up
userlabso@LabSoVM:~$ ./clean_up tmp_test
Print file? [y/n]: y
userlabso@LabSoVM:~$ ls /tmp | grep printfile.txt
userlabso@LabSoVM:~$
```

Open | clean_up ~/ | Save

```
1 #!/bin/bash
2
3 # Program to print a text file with headers and footers
4
5 TEMP_FILE=/tmp/printfile.txt
6
7 clean_up () {
8
9   # Perform program exit housekeeping
10   rm "$TEMP_FILE"
11   exit
12 }
13
14 trap clean_up SIGHUP SIGINT SIGTERM
15
16 pr $1 > "$TEMP_FILE"
17
18 read -p "Print file? [y/n]: "
19 if [ "$REPLY" = "y" ]; then
20   less "$TEMP_FILE"
21 fi
22 clean_up
```

Utilizarea unei funcții de curățare este o idee bună și pentru rutinele noastre de tratare a erorilor. La urma urmei, atunci când un program se termină (din orice motiv), ar trebui să curățăm după noi înșine. Iată versiunea finală a programului nostru, cu o gestionare îmbunătățită a erorilor și a semnalului:

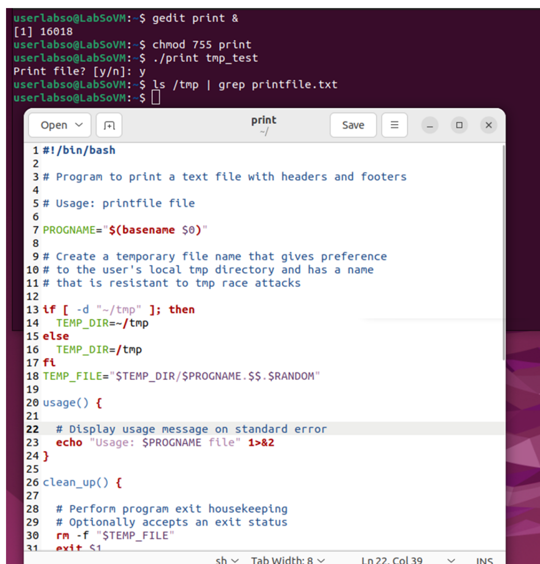
gedit print &

```
#!/bin/bash

# Program to print a text file with headers and footers

# Usage: printfile file

PROGNAME="$(basename $0)"

# Create a temporary file name that gives preference
# to the user's local tmp directory and has a name
# that is resistant to tmp race attacks

if [ -d "~/tmp" ]; then
  TEMP_DIR=~/tmp
else
  TEMP_DIR=/tmp
fi
TEMP_FILE="$TEMP_DIR/$PROGNAME.$$.$RANDOM"

usage() {
```

```
  # Display usage message on standard error
  echo "Usage: $PROGNAME file" 1>&2
}

clean_up() {

  # Perform program exit housekeeping
  # Optionally accepts an exit status
  rm -f "$TEMP_FILE"
  exit $1
}

error_exit() {

  # Display error message and exit
  echo "${PROGNAME}: ${1:-"Unknown Error"}" 1>&2
  clean_up 1
}

trap clean_up SIGHUP SIGINT SIGTERM

if [ $# != "1" ]; then
  usage
  error_exit "one file to print must be specified"
fi
if [ ! -f "$1" ]; then
  error_exit "file $1 cannot be read"
fi

pr $1 > "$TEMP_FILE" || error_exit "cannot format file"

read -p "Print file? [y/n]: "
if [ "$REPLY" = "y" ]; then
  less "$TEMP_FILE" || error_exit "cannot print file"
fi
clean_up
```

chmod 755 print

./print tmp_test



10.10. Crearea de fișiere temporare sigure

În programul de mai sus, există o serie de pași făcuți pentru a ajuta la securizarea fișierului temporar utilizat de acest script. Este o tradiție Unix să folosești un director numit /tmp pentru a plasa fișiere temporare utilizate de programe. Toată lumea poate scrie fișiere în acest director. Acest lucru duce în mod natural la unele probleme de securitate. Dacă este posibil, evitați să scrieți fișiere în directorul /tmp. Tehnica preferată este să le scriem într-un director local, cum ar fi ~/tmp (un subdirector tmp în directorul principal al utilizatorului.) Dacă fișierele trebuie scrise în /tmp, trebuie să luăm măsuri pentru a ne asigura că numele fișierelor nu sunt previzibile. Numele de fișiere previzibile pot permite unui atacator să creeze legături simbolice către alte fișiere pe care atacatorul dorește ca utilizatorul să le suprascrie.

Un nume de fișier bun va ajuta la identificarea a ceea ce a scris fișierul, dar nu va fi complet previzibil. În scriptul de mai sus, următoarea linie de cod a creat fișierul temporar $TEMP_FILE:

```
TEMP_FILE="$TEMP_DIR/$PROGNAME.$$.$RANDOM"
```

Variabila $TEMP_DIR conține fie /tmp, fie ~/tmp, în funcție de disponibilitatea directorului. Este o practică obișnuită să încorporați numele programului în numele fișierului. Am făcut asta cu constanta $PROGNAME construită la începutul scriptului. Apoi, folosim variabila shell $$ pentru a încorpora id-ul de proces (pid) al programului. Acest lucru ajută în continuare la identificarea procesului responsabil pentru fișier. În mod surprinzător, ID-ul procesului în sine nu este suficient de imprevizibil pentru a face fișierul în siguranță, așa că adăugăm variabila shell $RANDOM pentru a adăuga un număr aleator la numele fișierului. Cu această tehnică, creăm un nume de fișier care este atât ușor de identificat, cât și imprevizibil.

11. ÎNVĂȚÂND GNUPLOT - INSTALAREA, CITIREA DIN FIȘIER ȘI SALVAREA FIȘIERELOR

11.1. Invocarea gnuplot și primele grafice

```
sudo apt-get update -y
```

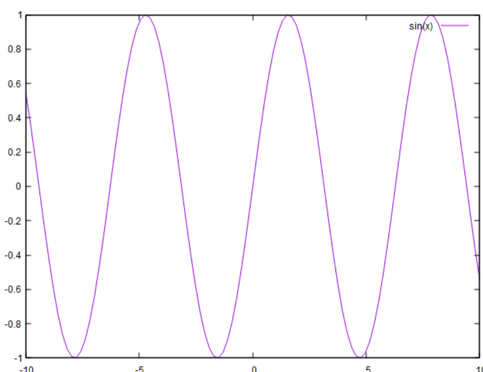
```
sudo apt-get install -y gnuplot5-x11
```

Deoarece gnuplot este un program de plot, nu ar trebui să fie surprinzător faptul că cea mai importantă comandă gnuplot este plot. Poate fi folosit pentru a reprezenta grafic ambele funcții (cum ar fi sin(x)) și date (de obicei dintr-un fișier). Comanda plot are o varietate de opțiuni și subcomenzi, prin care puteți controla aspectul graficului, precum și interpretarea datelor din fișier. Comanda plot poate chiar să efectueze transformări arbitrare asupra datelor pe măsură ce le trasați.

Probabil cea mai simplă comandă de trasare pe care o puteți emite este:

```
plot sin(x)
```

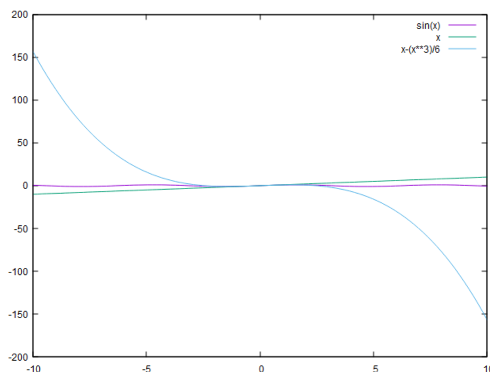
Pe Unix care rulează o interfață grafică (un manager de ferestre arbitrar care rulează pe X11), această comandă deschide o nouă fereastră care arată graficul rezultat. Observați cum gnuplot a selectat automat un interval „rezonabil" pentru valorile x (în mod implicit, de la -10 la +10) și a ajustat intervalul y în funcție de valorile funcției.



Să presupunem că doriți să adăugați mai multe funcții pentru a reprezenta un grafic împreună cu sinusul. Vă amintiți ultima comandă (folosind tasta săgeată sus sau Ctrl-P pentru „anterior") și o editați pentru a da:

```
plot sin(x), x, x-(x**3)/6
```

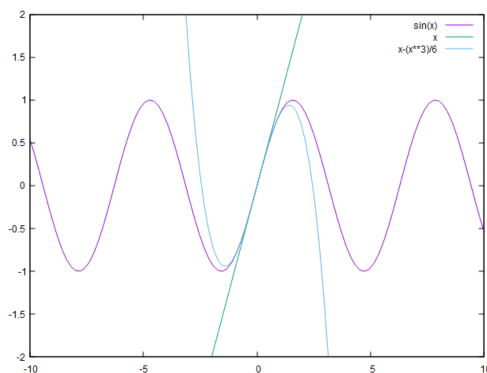
Aceasta va reprezenta graficul sinusului împreună cu funcția liniară x și polinomul de ordinul al treilea $x - \frac{1}{6}x^3$, care sunt primii termeni din dezvoltarea Taylor a sinusului. (Sintaxa Gnuplot pentru expresiile matematice este simplă și similară cu cea găsită în aproape orice alt limbaj de programare. Rețineți operatorul de exponențiere **, familiar de la Fortran sau Perl. Graficul rezultat probabil nu este ceea ce vă așteptați.



Intervalul valorilor y este mult prea mare, în comparație cu graficul anterior. Nici măcar nu mai puteți vedea mișcările funcției originale (unda sinusoidală). Gnuplot ajustează intervalul y pentru a se potrivi în toate valorile funcției, dar pentru această diagramă, sunteți interesat doar de punctele cu valori y mici. Așadar, reamintiți ultima comandă (folosind tasta săgeată sus) și definiți intervalul de grafic care vă interesează:

```
plot [][-2:2] sin(x), x, x-(x**3)/6
```

Intervalul este dat între paranteze drepte imediat după comanda plot. Prima pereche de paranteze definește intervalul de valori x (lăsați-l gol, pentru că sunteți mulțumit de valorile implicite în acest caz); al doilea restrânge intervalul de valori y afișat.



Vă puteți juca mult mai mult timp cu funcții, zonând diferite regiuni de interes și încercând diferite funcții. Dar să trecem mai departe și să discutăm pentru ce este cel mai util gnuplot: reprezentarea datelor dintr-un fișier.

11.2. Reprezentarea datelor dintr-un fișier

Gnuplot poate citi date din fișiere text. Mai întâi trebuie să stabilim directorul nostru de lucru, astfel încât să putem conecta fișierul text. Cel mai simplu mod de a-l determina este folosind următoarea comandă:

```
pwd
```

Pentru a schimba calea directorului nostru de lucru curent este folosită următoarea sintaxă:

```
cd 'C:\insert\path\here'
```

Se așteaptă ca datele să fie numerice și să fie stocate în fișier în coloane separate prin spații albe. Liniile care încep cu un semn hash (#) sunt considerate a fi linii de comentariu și sunt ignorate. Lista următoare reprezintă un fișier de date tipic care conține prețurile acțiunilor a două companii fictive, cu simbolurile ticker la fel de fictive PQR și XYZ, de-a lungul unui număr de ani.

```
# Average PQR and XYZ stock price (in dollars per share) per calendar
year

2000     49    162

2001     52    144

2002     67    140

2003     53    122

2004     67    125

2005     46    117

2006     60    116

2007     50    113

2008     66     96

2009     70    101

2010     91     93

2011    133     92

2012    127     95

2013    136     79

2014    154     78

2015    127     85

2016    147     71

2017    146     54

2018    133     51

2019    144     49

2020    158     43
```

Modul canonic de a gândi la acest lucru este că valoarea x este în coloana 1 și valoarea y este în coloana 2. Dacă există valori y suplimentare care corespund fiecărei valori x, acestea sunt listate în coloanele ulterioare. (Veți vedea mai târziu că prima coloană nu are nimic special. De fapt, orice coloană poate fi reprezentată fie de-a lungul axei x, fie de-a lungul axei y.)

Acest format, oricât de simplu este, s-a dovedit a fi extrem de util – atât de mult încât utilizatorii gnuplot de lungă durată generează de obicei date în acest fel pentru început. În special, capacitatea de a păstra seturi de date asociate în același fișier este de mare ajutor (deci nu trebuie să păstrați prețul acțiunilor PQR într-un fișier separat de cel al lui XYZ - deși ați putea dacă doriți).

Deși datele numerice separate prin spații albe sunt ceea ce se așteaptă gnuplot în mod nativ, gnuplot poate analiza și interpreta abateri semnificative de la această normă, inclusiv coloane de text (cu spațiu alb încorporat dacă sunt incluse între ghilimele duble), date lipsă și o varietate de reprezentări textuale pentru datele calendaristice, precum și date binare.

Reprezentarea datelor dintr-un fișier este simplă. Presupunând că fișierul afișat anterior se numește prices.txt și se află în directorul de lucru curent, puteți tasta:

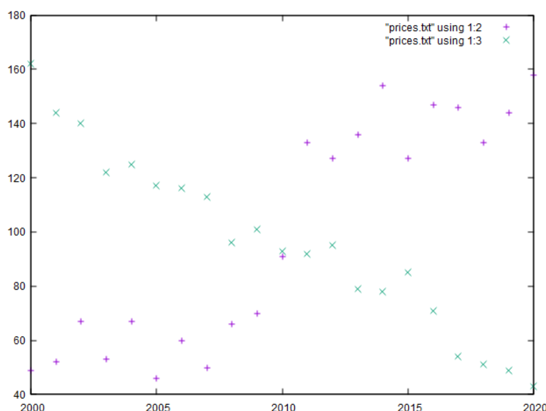
```
plot "prices"
```

Deoarece fișierele de date conțin de obicei multe seturi de date diferite, de obicei veți dori să selectați coloanele care vor fi utilizate ca valori x și y. Acest lucru se realizează prin directiva **using** la comanda plot:

```
plot "prices.txt" using 1:2
```

Aceasta reprezinta grafic prețul acțiunilor PQR în funcție de timp: primul argument al directivei **using** specifică coloana din fișierul de intrare care urmează să fie reprezentată de-a lungul axei orizontale (x), iar al doilea argument specifică coloana pentru axa verticala (y). Dacă doriți să reprezentați prețul acțiunilor XYZ în aceeași diagramă, puteți face acest lucru cu ușurință:

```
plot "prices.txt" using 1:2, "prices.txt" using 1:3
```



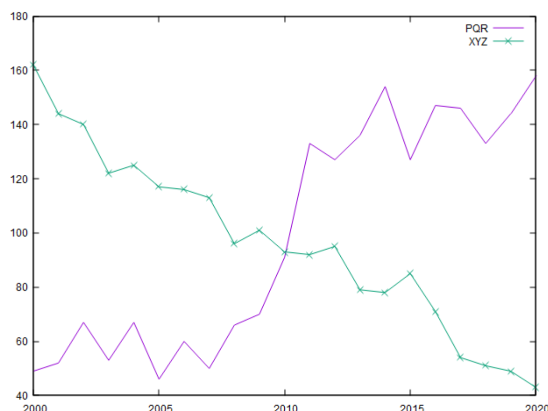
În mod implicit, punctele de date dintr-un fișier sunt reprezentate folosind simboluri neconectate. Adesea, acest lucru nu este ceea ce doriți, așa că trebuie să spuneți lui gnuplot ce stil să folosească pentru date. Aceasta se face folosind directiva **with**. Sunt disponibile multe stiluri diferite. Printre cele mai utile se numără **with linespoints**, care reprezintă fiecare punct de date ca simbol și, de asemenea, conectează punctele ulterioare, și **with lines**, care doar trasează liniile de legătură, omițând simbolurile individuale.

```
plot "prices.txt" using 1:2 with lines, \
     "prices.txt" using 1:3 with linespoints
```

Acum arată mai bine, dar nu este clar din grafic care linie este care. Gnuplot oferă automat o legendă (**key**), care arată un eșantion al tipului de linie sau simbol utilizat pentru fiecare set de date împreună cu o descriere text, dar descrierea implicită nu este foarte semnificativă în acest caz. Arată mult mai bine incluzând un titlu (**title**) pentru fiecare set de date ca parte a comenzii **plot**:

```
plot "prices.txt" using 1:2 title "PQR" with lines, \
     "prices.txt" using 1:3 title "XYZ" with linespoints
```

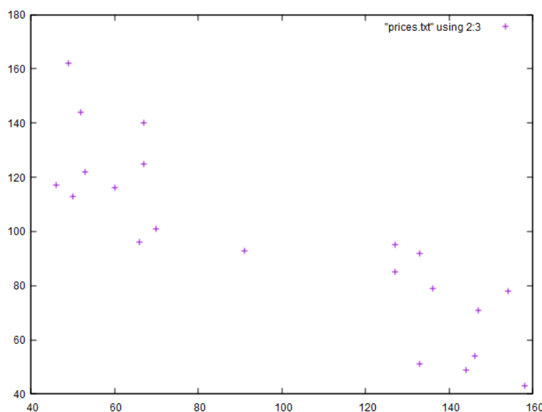
Aceasta schimbă textul din legendă cu șirul dat ca titlu. Titlul trebuie să vină după directiva **using** din comanda **plot**. O modalitate bună de a memora această ordine este să vă amintiți că trebuie să specificați mai întâi setul de date pentru a reprezenta un grafic și apoi să furnizați descrierea: mai întâi definiți-o, apoi descrieți ceea ce ați definit.



Doriți să vedeți cum se corelează prețul PQR cu XYZ? Nici o problemă; reprezentați un grafic unul împotriva celuilalt, folosind prețul acțiunilor PQR pentru valorile x și XYZ pentru valorile y, așa:

```
plot "prices.txt" using 2:3 with points
```

Vedeți aici că nu este nimic special la prima coloană. Orice coloană poate fi reprezentată fie pe axa x, fie pe axa y; alegeți orice combinație de care aveți nevoie prin directiva using. Deoarece nu are sens să conectăm punctele de date din ultimul grafic, am ales stilul cu puncte (**with points**), care reprezintă un simbol pentru fiecare punct de date, dar fără linii de legătură.



Un astfel de grafic este cunoscut sub numele de **scatter plot** și poate arăta corelații între două seturi de date. În acest grafic, puteți vedea o corelație negativă clară: pe măsură ce prețul acțiunilor PQR crește, prețul XYZ scade.

Acum că ați văzut cele mai importante comenzi de bază, să facem un pas înapoi pentru un moment și să introducem rapid câteva conforturi pe care gnuplot le oferă utilizatorului mai experimentat.

11.3. Abrevieri și valori implicite

Gnuplot este bun la încurajarea analizei datelor intr-o manieră iterativă și exploratorie. Ori de câte ori finalizați o comandă, graficul rezultat este afișat imediat și toate modificările intră în vigoare deodată. Scrierea comenzilor nu este o activitate diferită de generarea de grafice și nu este nevoie de un program de vizualizare separat. (Graficurile sunt, de asemenea, create aproape instantaneu; numai pentru seturile de date care includ milioane de puncte există vreo întârziere vizibilă.) Comenzile anterioare pot fi rechemate, modificate și reemise, facilitând continuarea jocului cu datele.

Gnuplot oferă încă două funcții pentru utilizatorul mai priceput: abrevieri și valori implicite. Orice comandă și subcomandă sau opțiune poate fi abreviată la cea mai scurtă, non-ambiguă formă. Deci comanda:

```
plot "prices.txt" using 1:2 with lines, \
     "prices.txt" using 1:3 with linespoints
```

este mai probabil să fie emisă ca:

```
plot "prices.txt" u 1:2 w l, "prices.txt" u 1:3 w lp
```

Acest stil compact este util atunci când lucrați interactiv și ar trebui să îl stăpâniți. În continuare, îl vom folosi din ce în ce mai mult.

Dar aceasta nu este încă cea mai compactă formă posibilă. Ori de câte ori o parte a unei comenzi nu este dată în mod explicit, gnuplot încearcă mai întâi să interpoleze valorile lipsă cu valorile furnizate de utilizator; în caz contrar, revine la valori implicite sensibile. Ați văzut deja

cum gnuplot setează implicit intervalul de valori x la [-10:10], dar ajustează intervalul y pentru a include toate punctele de date.

Ori de câte ori lipsește un nume de fișier, cel mai recent nume de fișier este interpolat. Puteți utiliza acest lucru pentru a abreviera și mai mult comanda anterioară:

```
plot "prices.txt" u 1:2 w l, "" u 1:3 w lp
```

Rețineți că al doilea set de ghilimele trebuie să fie acolo.

În general, orice intrare de utilizator (sau o parte din intrarea utilizatorului) rămâne neafectată până când este anulată în mod explicit de intrarea ulterioară. Modul în care numele fișierului este interpolat în exemplul precedent este un bun exemplu al acestui comportament. Mai târziu vom vedea cum pot fi construite opțiunile pas cu pas, furnizând ulterior valori pentru diferite subopțiuni. Gnuplot ajută la menținerea comenzilor scurte prin amintirea comenzilor anterioare cât mai mult posibil.

Un ultim exemplu se referă la directiva **using**. Dacă lipsește în întregime și fișierul de date conține mai multe coloane, gnuplot grafică a doua coloană față de prima (acest lucru este echivalent cu **using 1:2**). Dacă este dată o directivă **using**, dar listează doar o singură coloană, gnuplot folosește această coloană pentru valorile y și furnizează valorile x ca numere întregi începând cu zero. Acest lucru se întâmplă și atunci când nu se oferă niciun **using** și fișierul de date conține doar o singură coloană.

11.4. Salvarea comenzilor și exportarea graficelor

Există două modalități de a vă salva munca în gnuplot: puteți salva comenzile gnuplot utilizate pentru a genera un grafic, astfel încât să puteți regenera graficul mai târziu. Sau puteți exporta graficul într-un fișier într-un format standard de fișier grafic, astfel încât să îl puteți imprima sau să îl includeți în pagini web, documente sau prezentări.

Dacă salvați într-un fișier comenzile pe care le-ați folosit pentru a genera un grafic, le puteți încărca din nou ulterior și puteți regenera graficul de unde ați rămas. Comenzile Gnuplot pot fi salvate într-un fișier folosind comanda **save**:

```
save "graph.plt"
```

Aceasta salvează valorile curente ale tuturor opțiunilor, precum și cea mai recentă comandă de **plot**, în fișierul specificat. Acest fișier poate fi încărcat din nou folosind comanda **load**:

```
load "graph.plt"
```

Efectul încărcării unui fișier este același cu emiterea tuturor comenzilor conținute (inclusiv comanda **plot**) la promptul gnuplot.

Fișierele cu comenzi sunt fișiere text simple, care conțin de obicei exact o comandă pe linie. Mai multe comenzi pot fi combinate pe o singură linie, separându-le cu punct și virgulă (;) - aceasta funcționează și la linia de comandă interactivă. Pentru a scrie o comandă pe mai

multe linii, folosim (, \) pentru a-i spune lui gluplot să aștepte restul comenzii pe următoarea linie. Semnul hash (#) este interpretat ca un caracter de comentariu: restul liniei care urmează semnului hash este ignorată. Semnul hash nu este interpretat ca un caracter de comentariu atunci când apare între ghilimele.

Extensia de fișier recomandată pentru fișierele de comenzi gnuplot este .gp, dar este posibil să găsiți și persoane care folosesc .plt. Deoarece fișierele de comenzi sunt fișiere text simple, ele pot fi editate folosind un editor de text obișnuit. Uneori este util să le creați manual și să le încărcați în gnuplot și sunt, de asemenea, folosite pentru operațiuni și configurații în serie.

Comanda **save** pe care tocmai am introdus-o salvează comenzile necesare pentru a genera (sau regenera) graficul într-un fișier text, dar nu salvează graficul în sine. Cum obțineți graficul pe care îl vedeți în interfața grafică într-un fișier? Există trei moduri de a face acest lucru: ușor (cu mai puțin control), indirect (cu puțin control, dar încă nu suficient) și complicat (la prima utilizare, dar din ce în ce mai ușor odată configurat cu avantajul de a oferi grafice de cea mai bună calitate).

Folosind butonul GUI

În versiunile recente de gnuplot, majoritatea terminalelor contemporane (inclusiv terminalele wxt și qt) includ un buton GUI pe care îl puteți utiliza pentru a exporta un grafic într-un fișier. Butonul deschide un dialog standard de selectare a fișierului în care selectați numele dorit al fișierului țintă și formatul fișierului (PNG, PDF sau SVG).

Folosind clipboard-ul

O modalitate indirectă de a salva graficul într-un fișier este să faceți o captură de ecran a graficului, folosind utilitarul dvs. de captură de ecran preferat și să salvați rezultatul ca fișier GIF sau PNG. La urma urmei, fereastra grafică de pe ecran nu este altceva decât o imagine bitmap, iar o captură de ecran o salvează într-un fișier. Această metodă poate părea puțin nesofisticată, dar are două avantaje semnificative: fluxul de lucru este simplu, iar graficul rezultat corespunde exact graficului care a fost pe ecran.

Folosind terminale

Atât butonul GUI, cât și clipboard-ul sunt metode relativ noi pentru exportul unui grafic. Sunt ușor de utilizat, dar nu permit multă flexibilitate în aspectul graficului generat. Pentru a obține controlul deplin asupra procesului de export, trebuie să vă familiarizați cu facilitatea terminalului gnuplot.

În limbajul gnuplot, un terminal este un dispozitiv de ieșire capabil de grafică. În mod tradițional, aceasta poate fi o piesă hardware specifică (cum ar fi un plotter cu creion și cerneală), dar astăzi un terminal gnuplot este doar o referință la biblioteca grafică de bază.

Terminalele contemporane vin în două variante: interactive și bazate pe fișiere. Terminalele interactive creează graficele pe care le vedeți pe ecran. Pe Linux, aveți de ales între trei terminale interactive, fiecare construit folosind un set de widget-uri diferit: terminalul **qt** folosește Qt, terminalul **wxt** folosește wxWidgets, iar vechiul terminal **x11** este o aplicație pură Xlib. (Atât terminalele **qt**, cât și **wxt** există și pe Windows și Mac OS X, împreună cu terminale specifice platformei.)

Dar pentru a exporta un grafic într-un fișier, trebuie să utilizați un terminal bazat pe fișiere care poate genera rezultate în formatul de ieșire dorit (PNG, PDF, SVG și așa mai departe). Majoritatea terminalelor bazate pe fișiere acceptă un număr mare de opțiuni prin care puteți controla diferite aspecte (cum ar fi dimensiunea) graficului rezultat.

Toate acestea sunt simple. Dar există o piatră de poticnire: exportul unui grafic printr-un terminal bazat pe fișiere necesită mai mulți pași. Pentru a exporta un grafic cu un terminal gnuplot, mai multe comenzi trebuie să fie executate într-o secvență adecvată. Acest lucru poate fi o surpriză, deoarece „exportarea într-un fișier” este o operațiune unică, atomică în majoritatea celorlalte aplicații din zilele noastre. Gnuplot este diferit.

Secvența completă a pașilor este următoarea. Să trecem prin ea:

- Începeți cu o comandă **plot** arbitrară: `plot exp(-x**2)`
- Selectați terminalul bazat pe fișiere dorit, folosind comanda **set terminal**: `set terminal pngcairo`
- Specificați numele fișierului de ieșire, folosind comanda **set output**: `set output “graph.png”`
- Regenerați ultimul plot, trimițându-l de data aceasta către terminalul bazat pe fișiere și fișierul numit: `replot`
- Restabiliți terminalul interactiv, utilizând atât comanda **set terminal**, cât și comanda **set output**: `set terminal wxt` `set output`

Este prima dată când întâlniți comanda **set** a lui gnuplot, vom vorbi despre ea mai detaliat în laboratorul următor. Pentru moment, este suficient să înțelegeți că setează un parametru (cum ar fi terminalul) la o valoare. Dar, iar acest lucru este adesea uitat, nu generează un plot! Singurele comenzi care fac acest lucru sunt **plot**, **splot** (care este folosit pentru grafice tridimensionale) și **replot** și **refresh** (ambele repetă cea mai recentă comandă de **plot** sau **splot**).

Atâta timp cât țineți cont de aceste trei elemente, crearea fișierelor prin terminale nu va crea dificultăți. (Mergând mai departe, veți vedea cum să utilizați scripturi și macrocomenzi pentru a facilita acest proces.)

Un sfat: salvați întotdeauna comenzile folosite pentru a genera un plot într-un fișier de comandă înainte de a exporta într-un format imprimabil. Întotdeauna. Este aproape garantat că veți dori să regenerați graficul pentru a face o modificare minoră mai târziu (cum ar fi remedierea unei greșeli de tipar într-o legendă, adăugarea unui set de date sau ajustarea ușoară a intervalului de grafic). Acest lucru se poate face numai din comenzile salvate în fișier folosind **save**, nu din ploturi exportate într-un fișier grafic. Vom reveni la acest subiect de mai multe ori.

11.5. Gestionarea opțiunilor cu set și show

Gnuplot are relativ puține comenzi (cum ar fi comenzile **plot**, **save** și **load** pe care le-ați întâlnit deja), dar un număr mare de **opțiuni**. Aceste opțiuni sunt folosite pentru a controla totul, de la formatul punctului zecimal până la numele fișierului de ieșire. Sunt disponibile peste 100 de astfel de opțiuni, împreună cu nenumărate subopțiuni pentru fiecare. Toate aceste opțiuni pot fi găsite cu ușurință în manuale sau doar făcând căutări de bază pe google.

Cele trei comenzi folosite pentru a manipula opțiunile individuale sunt următoarele:

- **show** - Afișează valoarea curentă a unei opțiuni
- **set** - Modifică valoarea unei opțiuni
- **unset** - Dezactivează o anumită opțiune sau o readuce la valoarea implicită

Există, de asemenea, o a patra comandă, **reset**, care readuce toate opțiunile la valorile implicite. Singurele opțiuni care nu sunt afectate de **reset** sunt cele care influențează direct generarea ieșirii: **terminal** și **output**.

Sintaxa tuturor celor trei comenzi este simplă. Pentru a atribui o nouă valoare unei opțiuni, utilizați cuvântul cheie **set** urmat de numele opțiunii și noua valoare. Comanda **show** ia doar un singur argument: numele opțiunii pe care doriți să o inspectați.

Comanda **show** este, de asemenea, folosită în general pentru a afișa tot felul de informații despre starea internă gnuplot, nu doar opțiunile care pot fi modificate folosind **set**. De exemplu, **show variables** afișează toate variabilele care sunt definite în sesiunea curentă împreună cu valorile lor și **show functions** listează toate funcțiile definite de utilizator.

12. ÎNVĂȚÂND GNUPLOT - VARIABILE, FUNCȚII DEFINITE DE UTILIZATOR ȘI TUTORIALE

12.1 Variabile și funcții definite de utilizator

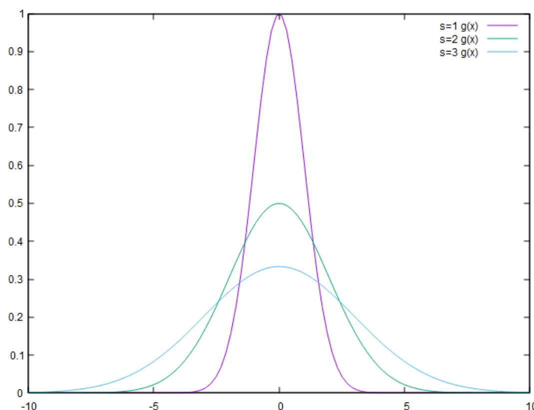
Este ușor să definiți variabile noi prin atribuirea unei expresii unui nume. De exemplu, ați putea dori să definiți câteva constante matematice utile. (Nu trebuie să definiți pi, deoarece este deja definit de gnuplot.)

Funcțiile pot avea până la 12 variabile și pot conține alte funcții și operatori. Le puteți folosi așa cum ați folosi orice altă funcție. Presupunând că ați emis definițiile, ați putea apoi să scrieți **plot sin(x), f(x)**.

În mod implicit, gnuplot presupune că variabila dummy independentă, care este înlocuită automat cu un interval de valori x atunci când trasează o funcție, este etichetată x (ca în: plot sin(x)), dar puteți modifica acest lucru folosind comanda **set dummy**. De exemplu, **set dummy t** face t variabila independentă, așa că acum ați spune **plot sin(t)**.

Toate celelalte variabile care apar într-o definiție a funcției (parametri) trebuie să li se fi atribuit valori înainte de a putea reprezenta (adică, evalua) funcția. Pentru comoditate, puteți aloca valori parametrilor ca parte a comenzii plot. De exemplu, puteți face următoarele (observați că nu este necesară nicio virgulă între alocarea variabilei și funcție):

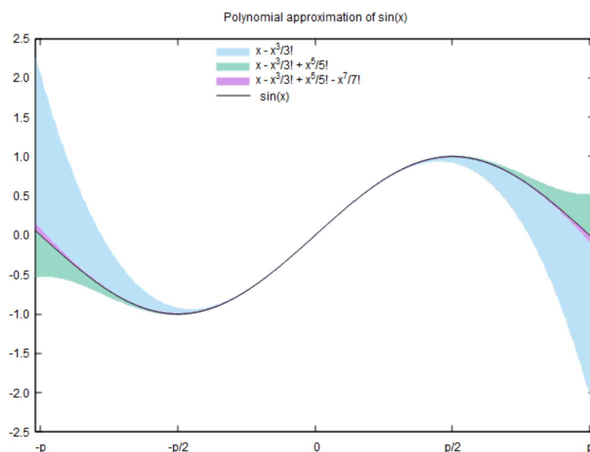
```
g(x) = exp(-0.5*(x/s)**2)/s
plot s=1 g(x), s=2 g(x), s=3 g(x)
```



12.2. Exerciții Gnuplot

12.2.1. Aproximarea polinomială a sinusului

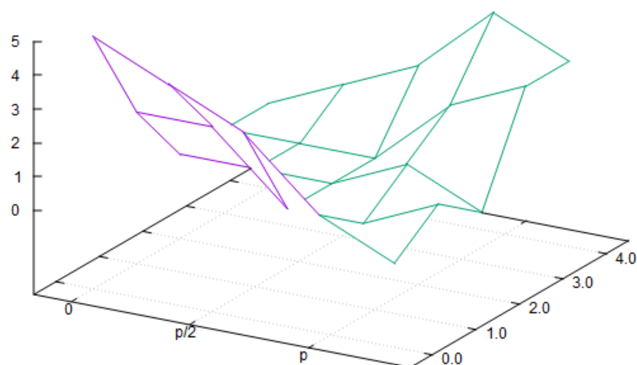
```
approx_1(x) = x - x**3/6
approx_2(x) = x - x**3/6 + x**5/120
approx_3(x) = x - x**3/6 + x**5/120 - x**7/5040
label1 = "x - {x^3}/3!"
label2 = "x - {x^3}/3! + {x^5}/5!"
label3 = "x - {x^3}/3! + {x^5}/5! - {x^7}/7!"
set termoption enhanced
save_encoding = GPVAL_ENCODING
set encoding utf8
set title "Polynomial approximation of sin(x)"
set key Left center top reverse
set xrange [ -3.2 : 3.2 ]
set xtics ("-π" -pi, "-π/2" -pi/2, 0, "π/2" pi/2, "π" pi)
set format y "%.1f"
set samples 500
set style fill solid 0.4 noborder

plot '+' using 1:(sin($1)):(approx_1($1)) with filledcurve title label1 lt
3, \
     '+' using 1:(sin($1)):(approx_2($1)) with filledcurve title label2 lt
2, \
     '+' using 1:(sin($1)):(approx_3($1)) with filledcurve title label3 lt
1, \
     sin(x) with lines lw 1 lc rgb "black"
```



12.2.2. Suprafață 3D dintr-o matrice de valori Z

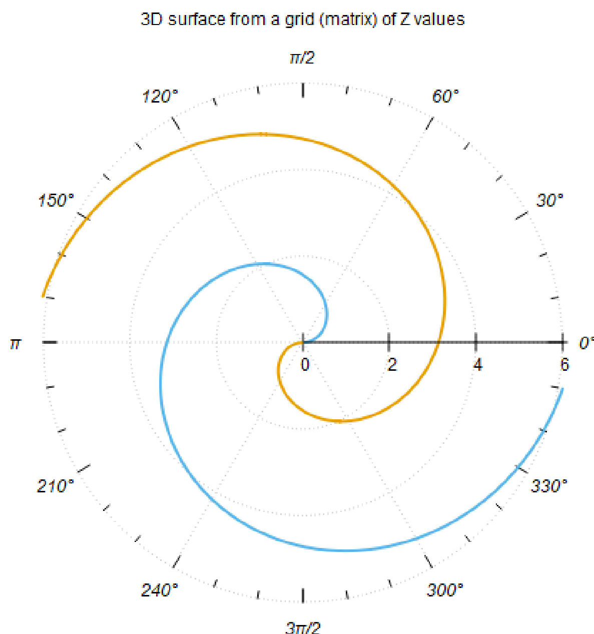
```
set title "3D surface from a grid (matrix) of Z values"
set xrange [-0.5:4.5]
set yrange [-0.5:4.5]
set grid
set hidden3d
$grid << EOD
5 4 3 1 0
2 2 0 0 1
0 0 0 1 0
0 0 0 2 3
```

```
0 1 2 4 3
EOD
splot '$grid' matrix with lines notitle
```



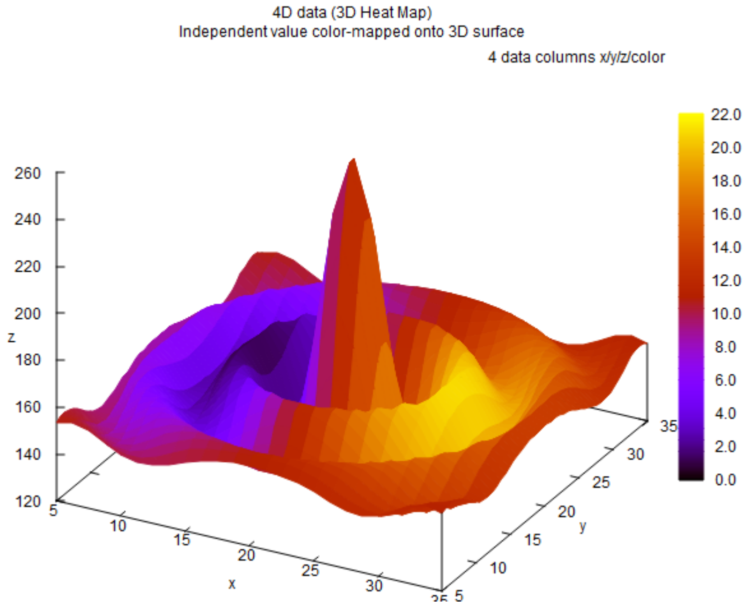
12.2.3. Grafice unghiulare

```
set title "Angle labels (ttics) for polar plots" offset 0,1
set polar
set ttics 0,30 format "%g".GPVAL_DEGREE_SIGN font ":Italic"
set mttics 3
set grid r polar 60
unset xtics
unset ytics
set border 0
set size square
unset key

set rrange [0:6.1]
if (GPVAL_ENCODING eq "utf8") {
    set ttics add ("π" 180, "π/2" 90, "3π/2" 270)
} else {
    set ttics add ("pi" 180, "pi/2" 90, "3pi/2" 270)
}

plot t lt 3 lw 2, -t lt 4 lw 2
```



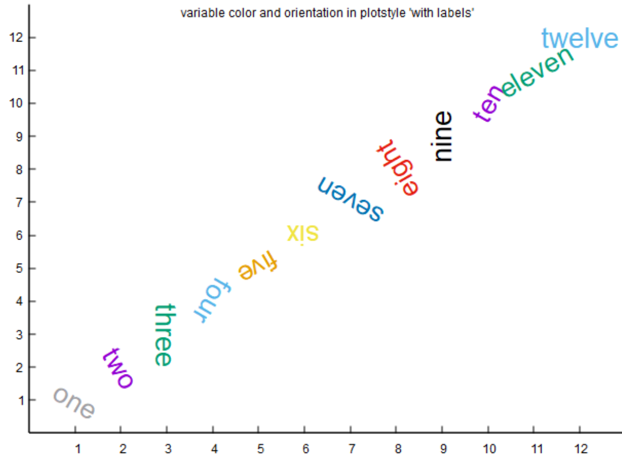
12.2.4. Valoare independentă mapată în culori pe suprafața 3D

```
set view 49, 28, 1, 1.48
set urange [ 5 : 35 ] noreverse nowriteback
set vrange [ 5 : 35 ] noreverse nowriteback
# set zrange [ 1.0 : 3.0 ] noreverse nowriteback
set ticslevel 0
set format cb "%4.1f"
set colorbox user size .03, .6 noborder
set cbtics scale 0
set samples 25, 25
set isosamples 50, 50
set title "4D data (3D Heat Map)"\
          ."\nIndependent value color-mapped onto 3D surface"  offset 0,1
set xlabel "x" offset 3, 0, 0
set ylabel "y" offset -5, 0, 0
set zlabel "z" offset 2, 0, 0
set pm3d implicit at s
Z(x,y) = 100. * (sinc(x,y) + 1.5)
sinc(x,y) = sin(sqrt((x-20.)**2+(y-20.)**2))/sqrt((x-20.)**2+(y-20.)**2)
color(x,y) = 10. * (1.1 + sin((x-20.)/5.)*cos((y-20.)/10.))
splot '++' using 1:2:(Z($1,$2)):(color($1,$2)) with pm3d title "4 data
columns x/y/z/color"
```



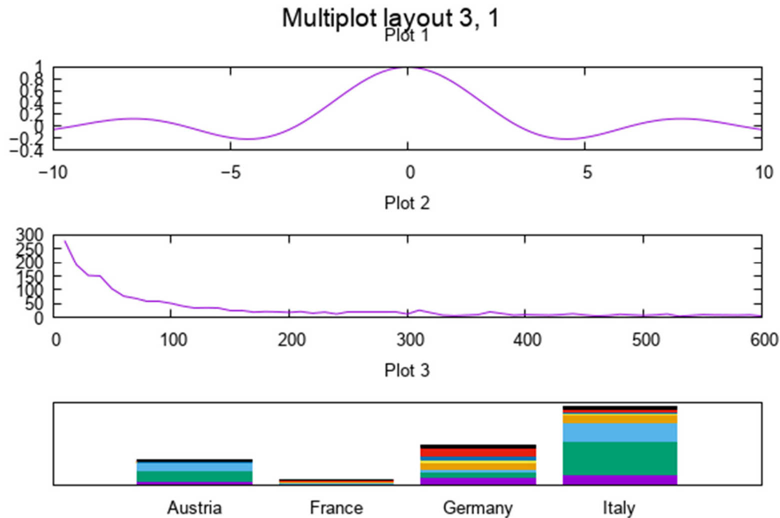
12.2.5. Culoare și orientare variabile în stilul de plot „with labels”

```
$Data <<EOD
1 one -30
2 two -60
3 three -90
4 four -120
5 five -150
6 six -180
7 seven -210
8 eight -240
9 nine -270
10 ten -300
11 eleven -330
12 twelve -360
EOD
set angle degrees
unset key
set title "variable color and orientation in plotstyle 'with
labels'" offset 0,-2
set xrange [0:13]
set yrange [0:13]
set xtics 1,1,12 nomirror
set ytics 1,1,12 nomirror
set border 3
plot $Data using 1:1:2:3:0 with labels rotate variable tc variable
font ",20"
```



12.2.6. Grafice multiple

```
set multiplot layout 3, 1 title "Multiplot layout 3, 1" font ",14"
set tmargin 2
set title "Plot 1"
unset key
plot sin(x)/x
#
set title "Plot 2"
unset key
plot 'silver.dat' using 1:2 ti 'silver.dat'
#
set style histogram columns
set style fill solid
set key autotitle column
set boxwidth 0.8
set format y "    "
set tics scale 0
set title "Plot 3"
plot 'immigration.dat' using 2 with histograms, \
     '' using 7  with histograms , \
     '' using 8  with histograms , \
     '' using 11 with histograms
#
unset multiplot
```

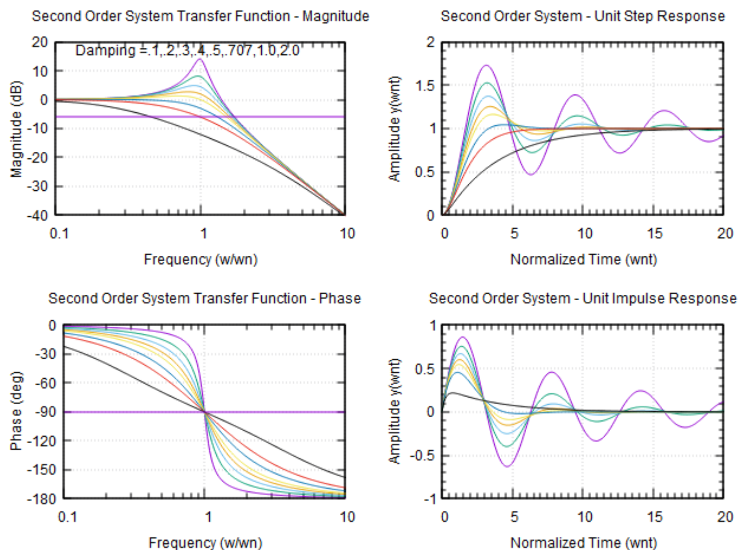


12.2.7. Sisteme de ordinul al 2-lea

```
D**2 + 2*zeta*wn*D + (wn**2)y = (wn**2)*x

#    x            input variable
#    y            output variable
#    w            frequency ratio (w/wn)
#    wn           natural frequency
#    wd           damped natural frequency
#    zeta         damping ratio
#    mag(w)       amplitude response
#    phi(w)       phase response
#    wdwn         damped natural frequency ratio
#    wnt          normalized time
#
#    Plots:
#    Frequency domain    magnitude response
#                            phase response
#
#    Time domain         unit step response
#                        unit impulse response
set style function lines
set size 1.0, 1.0
set origin 0.0, 0.0
set multiplot
set size 0.5,0.5
set origin 0.0,0.5
set grid
unset key
set angles radians
set samples 250
#
#   Plot Magnitude Response
set title "Second Order System Transfer Function - Magnitude"
mag(w) = -10*log10( (1-w**2)**2 + 4*(zeta*w)**2)
set dummy w
```

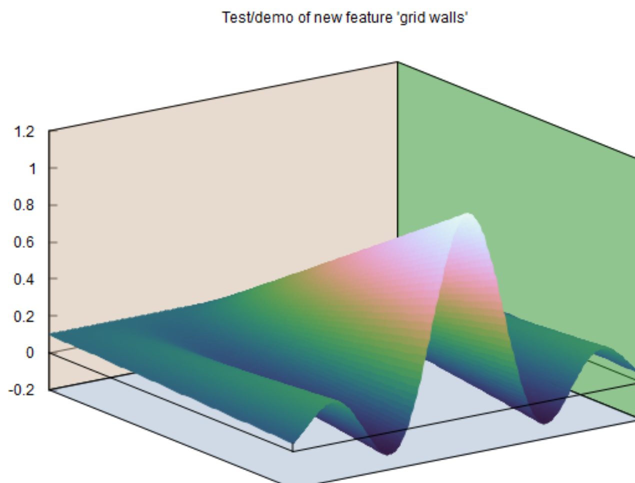
```
set logscale x
set xlabel "Frequency (w/wn)"
set ylabel "Magnitude (dB)" offset 1,0
set label 1 "Damping =.1,.2,.3,.4,.5,.707,1.0,2.0" at .14,17
set xrange [.1:10]
set yrange [-40:20]
plot \
  zeta=.1,mag(w), \
  zeta=.2,mag(w), \
  zeta=.3,mag(w), \
  zeta=.4,mag(w), \
  zeta=.5,mag(w), \
  zeta=.707,mag(w), \
  zeta=1.0,mag(w), \
  zeta=2.0,mag(w),-6
#
#  Plot Phase Response
set size 0.5,0.5
set origin 0.0,0.0
set title "Second Order System Transfer Function - Phase"
set label 1 ""
set ylabel "Phase (deg)" offset 1,0
set ytics -180, 30, 0
set yrange [-180:0]
tmp(w) = (-180/pi)*atan( 2*zeta*w/(1-w**2) )
#
# Fix for atan function wrap problem
tmp1(w)= w<1?tmp(w):(tmp(w)-180)
phi(w)=zeta==1?(-2*(180/pi)*atan(w)):tmp1(w)
plot \
  zeta=.1,phi(w), \
  zeta=.2,phi(w), \
  zeta=.3,phi(w), \
  zeta=.4,phi(w), \
  zeta=.5,phi(w), \
  zeta=.707,phi(w), \
  zeta=1,phi(w), \
  zeta=2.0,phi(w), \
  -90
#
#  Plot Step Response
set size 0.5,0.5
set origin 0.5,0.5
set dummy wnt
unset logscale x
set title "Second Order System - Unit Step Response"
set ylabel "Amplitude y(wnt)" offset 1,0
set xlabel "Normalized Time (wnt)"
set xrange [0:20]
set xtics 0,5,20
set yrange [0:2.0]
set ytics 0, .5, 2.0
set mytics 5
set mxtics 10
wdwn(zeta)=sqrt(1-zeta**2)
```

```
shift(zeta) = atan(wdwn(zeta)/zeta)
alpha(zeta)=zeta>1?sqrt(zeta**2-1.0):0
tau1(zeta)=1/(zeta-alpha(zeta))
tau2(zeta)=1/(zeta+alpha(zeta))
c1(zeta)=(zeta + alpha(zeta))/(2*alpha(zeta))
c2(zeta)=c1(zeta)-1
y1(wnt)=zeta==1?1 - exp(-wnt)*(wnt + 1):0
y2(wnt)=zeta<1?(1 - (exp(-zeta*wnt)/wdwn(zeta))*sin(wdwn(zeta)*wnt +
shift(zeta))):y1(wnt)
y(wnt)=zeta>1?1-c1(zeta)*exp(-wnt/tau1(zeta))+c2(zeta)*exp(-
wnt/tau2(zeta)):y2(wnt)
plot \
  zeta=.1,y(wnt), \
  zeta=.2,y(wnt), \
  zeta=.3,y(wnt), \
  zeta=.4,y(wnt), \
  zeta=.5,y(wnt), \
  zeta=.707,y(wnt), \
  zeta=1,y(wnt), \
  zeta=2,y(wnt)
#
#  Plot Impulse Response
set origin .5,0.
set title "Second Order System - Unit Impulse Response"
y(wnt)=exp(-zeta*wnt) * sin(wdwn(zeta)*wnt) / wdwn(zeta)
set yrange [-1. :1.]
set ytics -1,.5,1.
plot \
  zeta=.1,y(wnt), \
  zeta=.2,y(wnt), \
  zeta=.3,y(wnt), \
  zeta=.4,y(wnt), \
  zeta=.5,y(wnt), \
  zeta=.707,y(wnt), \
  zeta=1,y(wnt), \
  zeta=2,y(wnt)
unset multiplot
```



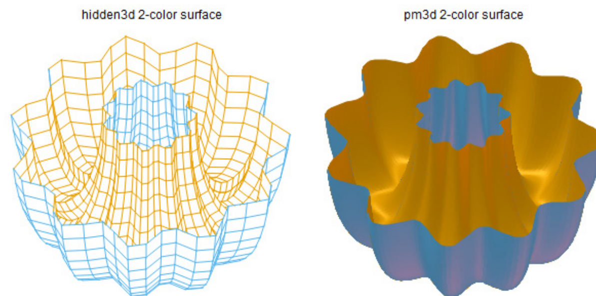
12.2.8. Caracteristica pereţi de reţea

```
set title "Test/demo of new feature 'grid walls'"

set wall z0 fs transparent solid 0.5 border -1 fc "slategray"
set wall x0 fs transparent solid 0.5 border -1 fc "forest-green"
set wall y0 fs transparent solid 0.5 border -1 fc "bisque"

set xrange [0:1]
set yrange [0:1]
set xyplane at 0
set isosample 25,25
set palette cubehelix
set pm3d interpolate 2,2
unset colorbox
unset key
unset xtics
unset ytics

set view 65, 145

sinc(x) = sin(x)/x
splot .1 + y* sinc(20*(x-0.5)) with pm3d
```



12.2.9. Suprafață în 2 culori (Hidden3d/pm3d)

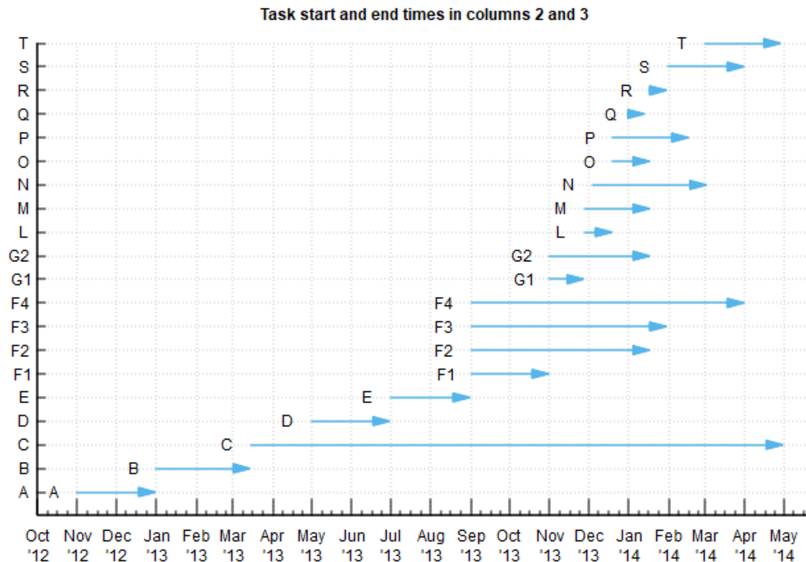
```
unset border
set dummy u, v
unset key
set view 225, 206, 1.25, 0.50
unset xtics
unset ytics
unset ztics
unset colorbox
set parametric
set dummy u,v
set urange [ -3.14159 : 3.14159 ] noreverse nowriteback
set vrange [ 0.250000 : 3.14159 ] noreverse nowriteback
set isosamples 50, 20
set multiplot layout 1,2 margins .05,.95,.2,.8 spacing 0.05
set title "hidden3d 2-color surface"
set hidden3d back offset 1

splot (cos(u)+.5*cos(u)*cos(v))*(1.+sin(11.*u)/10.), \
      (sin(u)+.5*sin(u)*cos(v))*(1.+sin(11.*u)/10.), \
      0.5*sin(v) with lines lt 3
set title "pm3d 2-color surface"
set pm3d depthorder
set pm3d interpolate 1,1 # border linecolor rgb "#a0a0f0"  linewidth 0.500
set pm3d lighting primary 0.33 specular 0.2 spec2 0.3
set isosamples 200,200
splot (cos(u)+.5*cos(u)*cos(v))*(1.+sin(11.*u)/10.), \
      (sin(u)+.5*sin(u)*cos(v))*(1.+sin(11.*u)/10.), \
      0.5*sin(v) with pm3d fc ls 3

unset multiplot
```



12.2.10. Diagrama Gantt simplă

```
$DATA << EOD
#Task start      end
A     2012-11-01 2012-12-31
B     2013-01-01 2013-03-14
C     2013-03-15 2014-04-30
D     2013-05-01 2013-06-30
E     2013-07-01 2013-08-31
F1    2013-09-01 2013-10-31
F2    2013-09-01 2014-01-17
F3    2013-09-01 2014-01-30
F4    2013-09-01 2014-03-31
G1    2013-11-01 2013-11-27
G2    2013-11-01 2014-01-17
L     2013-11-28 2013-12-19
M     2013-11-28 2014-01-17
N     2013-12-04 2014-03-02
O     2013-12-20 2014-01-17
P     2013-12-20 2014-02-16
Q     2014-01-05 2014-01-13
R     2014-01-18 2014-01-30
S     2014-01-31 2014-03-31
T     2014-03-01 2014-04-28
EOD

set xdata time
timeformat = "%Y-%m-%d"
set format x "%b\n'%y"

set yrange [-1:]
OneMonth = strptime("%m","2")
set xtics OneMonth nomirror
set xtics scale 2, 0.5
set mxtics 4
set ytics nomirror
set grid x y
unset key
set title "{/=15 Simple Gantt Chart}\n\n{/:Bold Task start and end times in
columns 2 and 3}"
set border 3
```

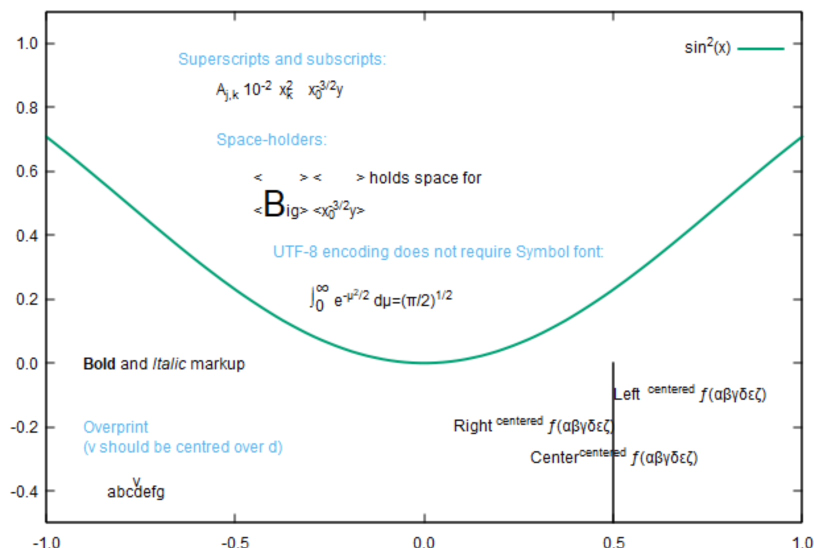
```
T(N) = timecolumn(N,timeformat)

set style arrow 1 filled size screen 0.02, 15 fixed lt 3 lw 1.5

plot $DATA using (T(2)) : ($0) : (T(3)-T(2)) : (0.0) : yticlabel(1) with
vector as 1, \
     $DATA using (T(2)) : ($0) : 1 with labels right offset -2
```



12.2.11. Modul text îmbunătățit folosind un singur font codificat UTF-8

```
set termoption enhanced
save_encoding = GPVAL_ENCODING
set encoding utf8
#
set title "Demo of enhanced text mode using a single UTF-8 encoded
font\nThere is another demo that shows how to use a separate Symbol
font"
set xrange [-1:1]
set yrange [-0.5:1.1]
set format xy "%.1f"
set arrow from  0.5, -0.5 to  0.5, 0.0 nohead
#
set label 1 at -0.65, 0.95
set label 1 "Superscripts and subscripts:" tc lt 3

set label 3 at -0.55, 0.85
set label 3 'A_{j,k} 10^{-2}  x@^2_k    x@_0^{-3/2}y'

set label 5 at -0.55,  0.7
set label 5 "Space-holders:" tc lt 3
set label 6 at -0.45, 0.6
set label 6 "<&{{/=20 B}ig}> <&{x@_0^{-3/2}y}> holds space for"
set label 7 at -0.45, 0.5
```

```
set label 7 "<{{/=20 B}ig}> <{x@_0^{-3/2}y}>"

set label 8 at -0.9, -0.2
set label 8 "Overprint\n(v should be centred over d)" tc lt 3
set label 9 at -0.85, -0.4
set label 9 " ~{abcdefg}{0.8v}"


set label 10 at  -.40, 0.35
set label 10 "UTF-8 encoding does not require Symbol font:" tc lt 3
set label 11 at -.30, 0.2
set label 11 "{/*1.5 ∫@_{/=9.6 0}^{/=12 ∞}} {e^{-{μ}^2/2}
d}{μ=(π/2)^{1/2}}"

set label 21 at 0.5, -.1
set label 21 "Left  ^{centered} ƒ(αβγδεζ)" left
set label 22 at 0.5, -.2
set label 22 "Right ^{centered} ƒ(αβγδεζ)" right
set label 23 at 0.5, -.3
set label 23 "Center^{centered} ƒ(αβγδεζ)" center

set label 30 at -.9, 0.0 "{/:Bold Bold} and {/:Italic Italic}
markup"
#
set key title " "
plot sin(x)**2 lt 2 lw 2 title "sin^2(x)"
```



BIBLIOGRAFIE

1. J. Bacon - Concurrent Systems, Addison Wesley 1997.
2. A. Silberschatz, J. Peterson and P. Galvin - Operating Systems Concepts (5th Ed.), Addison Wesley 1998.
3. The Design and Implementation of the 4.3BSD UNIX Operating System Leffler S J, Addison Wesley 1989.
4. D. Solomon and M. Russinovich - Windows Internals (4th Ed), Microsoft Press 2005.
5. A. Tanenbaum, H. Bos - Modern operating systems (5th Ed.), Pearson Education 2022.
6. R. Deaconescu, R. Rughinis, M. Carabaș, A. Radovici - Utilizarea sistemelor de operare, Printech 2021.
7. R. Deaconescu, R. Rughinis, G. Milescu, M. Bardac - Introducere în sisteme de operare, Printech 2009.
8. W. R. Stevens – Advanced Programming in the Unix Environment, Addison-Wesley 1992.
9. E.S. Raymond – The Art of Unix Programming, Addison Wesley 2003.
10. S. Hunger – Debian GNU/Linux Bible, Wiley 2001.
11. M. G. Graff and K. R. Van Wyk – Secure Coding: Principles and Practices, O'Reilly 2003.
12. J. Fusco – The Linux Programmer's Toolbox, Prentice Hall 2007.
13. T. Adelstein and B. Lubanovic – Linux System Administration, O'Reilly Media 2007.
14. W. Shotts – The Linux Command Line (5th Ed.), No Starch Press 2019.
15. P. K. Janert – Gnuplot in action (2nd Ed.), Manning Publications 2016.

ISBN 978-606-35-0600-0