Ciprian Oprișa **Adrian Coleșa**

SISTEME DE OPERARE

îndrumător de laborator

Editura UTPRESS
Cluj-Napoca, 2021
ISBN 978-606-737-512-1

Editura U.T.Press
Str. Observatorului nr. 34
C.P. 42, O.P. 2, 400775 Cluj-Napoca
Tel.: 0264-401999
e-mail: utpress@biblio.utcluj.ro
https://biblioteca.utcluj.ro/editura

Director: ing. Călin Câmpean

Recenzia: prof.dr.ing. Alin Suciu
ș.l.dr.ing. Kinga Marton

ISBN 978-606-737-512-1
Bun de tipar: 03.06.2021

Cuprins

Introducere

Sistemele de operare sunt prezente pe orice sistem de calcul modern, fie că e vorba de un calculator, un telefon mobil sau chiar un bec inteligent. Acesta are atât rolul de a gestiona resursele hardware cât și cel de interacțiune cu programele utilizator.

În acest îndrumător de laborator autorii își propun să formeze deprinderile studenților de a scrie programe care interacționează cu sistemul de operare și folosesc mecanismele furnizate de acesta. Chiar dacă avem pe piață o varietate de sisteme de operare precum și o varietate și mai mare de limbaje de programare, biblioteci și framework-uri, regăsim concepte comune cum ar fi cel de fișier sau cel de proces.

S-a ales pentru studiul de față sistemul de operare Linux, împreună cu limbajul C, care permite interacțiunea cu sistemul de operare prin intermediul apelurilor de sistem.

Primul laborator prezintă câteva concepte de bază despre sistemul de operare Linux și se concentrează pe instalarea acestuia. Distribuția de Linux recomandată la acest laborator este Ubuntu, mai prietenoasă cu utilizatorii care vin din zona Windows, dar conceptele prezentate de-a lungul semestrului se aplică oricărei distribuții Linux și parțial chiar și altor sisteme de operare inspirate din Unix.

Al doilea laborator se concentrează pe lucrul în linia de comandă și scrierea de fișiere de comenzi (script-uri).

Începând cu al treilea laborator se va lucra în limbajul C. Scopul acestui laborator nu este învățarea limbajului (acesta se presupune a fi cunoscut), ci utilizarea acestuia în mediul Linux. Se prezintă compilatorul `gcc`, rularea programelor pe Linux din terminal, precum și alte unelte pentru depanarea și analiza programelor. Exemplele de cod din acest îndrumător au fost testate cu diverse versiuni de compilator, de la `gcc-7` la `gcc-9` și s-a evitat utilizarea unor elemente de sintaxă specifice unei anumite versiuni, dar care ar putea fi modificate sau înlocuite în versiuni ulterioare.

Prima componentă a sistemului de operare cu care se va interacționa prin programe va fi sistemul de fișiere. Laboratorul 4 va prezenta apelurile de sistem specifice fișierelor în timp ce laboratorul 5 se va ocupa de directoare.

Laboratorul 6 prezintă conceptele legate de procese precum și apelurile de sistem legate de utilizarea acestora. În laboratorul 7 se vor prezenta firele de execuție, numite și thread-uri. Odată cu paralelismul introdus de procese și thread-uri apar și probleme de sincronizare, tratate folosind mecanisme specifice, cum ar fi semafoare, lacăte și variabile condiționale. Aceste mecanisme de sincronizare sunt introduse în laboratoarele 8, 9 și 10.

Următoarele două laboratoare, 11 și 12, se ocupă de comunicarea între procese, prezentând concepte cum ar fi pipe-urile, maparea fișierelor în memorie și memoria partajată.

Ultimul laborator, cel de-al 13-lea, conține un set de probleme recapitulative, în vederea pregătirii pentru colocviul de laborator.

Se recomandă studenților care urmează laboratorul de Sisteme de Operare citirea în avans a lucrărilor de laborator, pentru a se familiariza cu conceptele prezentate și pentru a putea folosi într-un mod cât mai util timpul alocat laboratorului. Pentru a verifica dacă s-a înțeles conținutul, recomandăm studenților să parcurgă și întrebările recapitulative de la finalul capitolelor și să încerce să răspundă singuri la ele. Majoritatea lucrărilor de laborator se concentrează pe exemple practice, care urmează după fiecare concept teoretic prezentat. Secvențele de cod prezentate sunt explicate, încercând să expunem și motivația pentru care s-a ales abordarea respectivă. Versiunile complete ale programelor prezentate se găsesc la adresa
`https://github.com/cypryoprisa/os-lab-examples`. Problemele propuse din cadrul fiecărui

laborator ar trebui rezolvate în timpul laboratorului, sub îndrumarea profesorului coordonator. Fiecare problemă are un punctaj alocat, care în general este proporțional cu dificultatea problemei și cu efortul necesar spre a o rezolva. Așteptările de la un student de nivel mediu sunt rezolvarea unui set de probleme care împreună totalizează între 50 și 75 de puncte.

Autorii adresează mulțumiri cadrelor didactice asociate care au coordonat lucrările de laborator în ultimii ani la materia Sisteme de Operare de la Universitatea Tehnică din Cluj-Napoca, sprijinul și feed-back-ul lor îmbunătățind semnificativ conținutul îndrumătorului de față: Magda Buhu-Pop, Marius Checicheș, Mihai Feier, David Harabagiu, Andrei Mihalca, Francisc Moldovan, Mihai Moldovan, Andrei Seicean, Gergő Széles, Adrian Tirea, Ovidiu Valea și Tudor Văran.

Laborator 1

Instalarea și primii pași în Linux

1.1 Scopul lucrării

Scopul acestui laborator este familiarizarea cu mediul specific sistemului de operare Linux.

Se va instala Linux într-o mașină virtuală, urmărind configurările necesare. Se va pune apoi în evidență structura arborescentă a sistemului de fișiere și se vor studia câteva comenzi fundamentale în terminal.

1.2 Fundamente teoretice

1.2.1 Instalarea sistemului de operare într-o mașină virtuală

O mașină virtuală este echivalentul software al unui calculator real, oferit de o aplicație numită emulator. Cele mai cunoscute astfel de aplicații sunt:

- VirtualBox (rulează pe Linux, Windows, Mac);
- VMware (rulează pe Linux și Windows);
- QEMU (rulează pe Linux);
- Virtual PC (rulează pe Windows);
- Parallels (în special pentru Mac, dar rulează și pe Linux și Windows).

Atunci când vorbim despre mașini virtuale, folosim următorii termeni:

- *host* (ro. *gazdă*), care se referă la calculatorul real, exterior;
- *guest* (ro. *oaspete*), care se referă la calculatorul emulat, interior.

În cadrul acestui laborator vom instala distribuția de Linux numită Ubuntu, într-o mașină virtuală de tip VirtualBox.

Notă: Pentru a putea rezolva exercițiile și temele de laborator acasă, e necesar să aveți instalată o distribuție de Linux și pe calculatorul/laptop-ul personal. Aceasta se poate instala fie într-o mașină virtuală (recomandat dacă aveți mai mult de 4GB memorie RAM), fie direct pe sistemul real. În cazul instalării pe sistemul real, există varianta *dual boot*, adică la pornirea sistemului să fiți întrebați dacă doriți să pornească Windows sau Linux.

Pentru a instala Ubuntu, se recomandă download-ul imaginii ISO (`https://www.ubuntu.com`), versiunea pentru desktop. Imaginea ISO reprezintă conținutul unui DVD de instalare și se poate folosi ca virtual DVD, în cazul instalării într-o mașină virtuală sau se poate scrie pe un DVD sau stick USB.

Dacă utilizăm VirtualBox, primul pas este definirea mașinii virtuale prin apăsarea butonului *New* sau alegerea opțiunii *New* din meniul *Machine*. În fereastra care apare trebuie definite următoarele elemente:

- numele mașinii virtuale;
- locația unde se salvează mașina virtuală; această locație poate fi diferită de locația unde se salvează hard-disk-ul mașinii virtuale.
- tipul și versiunea de sistem de operare; se va alege *Linux* la tip și *Ubuntu (64-bit)* la versiune.
  - Observație: dacă după ce ați selectat tipul *Linux* singurele opțiuni pentru versiune sunt variantele *32-bit*, cel mai probabil procesorul vostru nu are activat suportul pentru virtualizare. Activarea se face din BIOS, numele meniului și al opțiunii fiind diferite de la un calculator la altul.
- dimensiunea memoriei RAM; se recomandă cel puțin 1 GB, dacă calculatorul vostru are o memorie generoasă (mai mult de 4GB) se poate da mai mult.
- hard disk-ul mașinii virtuale; se va salva ca un fișier pe discul real, și poate fi alocat dinamic (va crește pe măsură ce se scriu date pe el) sau de dimensiune fixă. Se recomandă cel puțin 10GB.

Observație: recomandările de specificații ale mașinii virtuale au fost făcute pe baza versiunii Ubuntu 20.04 LTS. Pentru versiuni mai noi consultați recomandările curente.

După definirea mașinii virtuale trebuie să simulăm "introducerea" discului descărcat în mașina virtuală. Acest lucru se face din meniul *Settings* la secțiunea *Storage*. Odată ce discul virtual a fost introdus, mașina virtuală poate fi pornită și se poate continua cu instalarea ghidată a sistemului de operare.

După ce sistemul de operare a fost instalat și mașina virtuală a fost repornită trebuie instalat modulul *Guest Additions*, care ajută la o mai bună integrare dintre *host* și *guest*.

1.2.2 Structura arborescentă a sistemului de fișiere

Pe Linux, datele stocate permanent sunt reprezentate sub formă de *fișiere*. Fișierele sunt organizate în *foldere* (numite și *directoare*). Un folder poate conține fișiere, precum și alte foldere. Rădăcina sistemului de fișiere este întotdeauna un folder numit "root" și notat `"/"`.

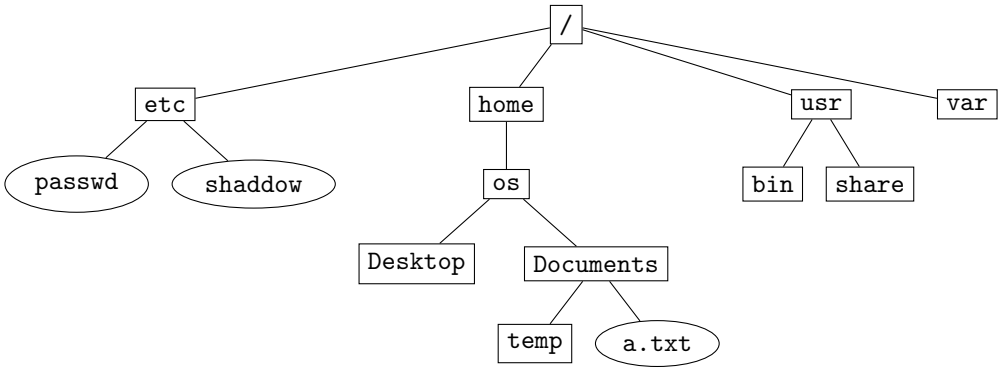


Figura 1.1: Exemplu de sistem de fișiere arborescent

Figura 1.1 prezintă un exemplu de sistem de fișiere, folderele fiind reprezentate prin dreptunghiuri, iar fișierele prin elipse.

Pentru a specifica calea completă spre un fișier, scriem toate folderele care îl conțin, de la rădăcină până la fișier. De exemplu, calea completă a fișierului `a.txt` din Figura 1.1 este `/home/os/Documents/a.txt`.

Fiecare folder conține implicit folderul "`.`", care se referă la folderul curent și folderul "`..`", care se referă la folderul părinte. De exemplu, dacă ne aflăm în folderul `/home/os/Desktop` și vrem să ne referim la fișierul `/home/os/Documents/a.txt` printr-o cale relativă, putem scrie `../Documents/a.txt`.

1.2.3 Utilizarea terminalului

Terminalul, numit și interpretor de comenzi, este un program care pune la dispoziția utilizatorului o interfață prin care se pot tasta comenzi și vizualiza rezultatul acestora.

1.2.3.1 Setarea server-ului proxy

În anumite situații, cum este de exemplu rețeaua universității, conexiunea la Internet poate trece printr-un "proxy server". În acest caz, programele care se conectează la Internet trebuie să "știe" acest lucru. Unul dintre modurile de a furniza informații programelor care rulează sub Linux este utilizarea variabilelor de mediu. Acestea funcționează similar cu niște variabile globale pentru toate programele.

Informația despre proxy server se află în variabilele `http_proxy`, `https_proxy` și `ftp_proxy` pentru protocoalele HTTP, HTTPS, respectiv FTP. Pentru a configura pentru aceste protocoale cu server-ul proxy al universității tehnice (UTCN), `proxy.utcluj.ro`, cu portul `3128`, se pot rula în termina următoarele comenzi:

Setare proxy în terminal

```
export http_proxy='http://proxy.utcluj.ro:3128/'
export https_proxy='http://proxy.utcluj.ro:3128/'
export ftp_proxy='http://proxy.utcluj.ro:3128/'
```

Notă: setarea variabilelor de mediu se face la nivel de terminal și nu este valabilă pentru alte ferestre de terminal sau după închiderea acestuia.

1.2.3.2 Instalarea de software

Distribuțiile Linux bazate pe Debian (inclusiv Ubuntu) folosesc programul *APT* (Advanced Package Tool) pentru gestiunea programelor instalate. Deoarece acest program poate aduce modificări importante sistemului instalat, nu se poate rula decât ca super-utilizator. Pentru a executa programe sau comenzi ca super-utilizator, acestea trebuie precedate de `sudo` (Super User DO).

Pentru a instala un anumit program (împreună cu eventualele dependințe), se va rula comanda:

```
sudo apt install <nume_program>
```

Pentru a dezinstala un program, rulăm:

```
sudo apt remove <nume_program>
```

Pentru a actualiza baza de date a programului APT, rulăm:

```
sudo apt update
```

1.2.3.3 Primele comenzi uzuale

man afișează "manualul de instrucțiuni" pentru comenzi de terminal, apeluri de sistem sau funcții. Ieșirea din modul manual se face folosind tasta `'q'` (quit). Detalii despre navigarea în pagina de manual se pot afla apăsând tasta `'h'` (help).

Exemple man

```
man ls
man 3 printf
man man
```

cd (Change Dir) schimbă directorul curent în cel dat ca parametru.

Exemple cd

```
cd /home/os/Documents
cd ..
```

pwd (Path to Working Directory) afișează calea spre directorul curent.

mkdir creează folderul a cărui cale (absolută sau relativă) s-a primit ca parametru.

ls afișează conținutul unui folder.

Exemple ls

```
ls
ls -l /
```

echo afișează textul primit ca parametru.

touch creează fișierul primit ca parametru.

rm șterge fișierul primit ca parametru.

cat afișează conținutul fișierelor primite ca parametru.

chmod schimbă permisiunile unui fișier sau folder.

Exemple chmod

```
chmod 0740 file1
chmod a-w file2
```

1.3 Mersul lucrării

1. Instalați Ubuntu Linux într-o mașină virtuală de tip VirtualBox.
2. Actualizați baza de date a programului APT și instalați programele `mc`, `codeblocks` și `g++`, conform indicațiilor din 1.2.3.2. Nu uitați să setați proxy-urile înainte, conform 1.2.3.1.
3. Consultați paginile de manual pentru comenzile din secțiunea 1.2.3.3.
4. Folosind IDE-ul *CodeBlocks* scrieți un program în C care afișează mesajul `"Hello World!"` pe ecran.
5. Folosind comenzile `cd` și `ls` navigați în folderul proiectului realizat anterior și afișați toate fișierele din acesta, împreună cu permisiunile lor.

Laborator 2

Interpretorul de comenzi

2.1 Scopul lucrării

Lucrarea curentă abordează interpretorul de comenzi din Linux, împreună cu cele mai uzuale comenzi. Se discută de asemenea și fișierele de comenzi (eng. *scripts*).

2.2 Fundamente teoretice

O comandă dată interpretorului are un nume și poate avea zero, unul sau mai multe argumente.

Format general comenzi

```
command_name arguments
```

În mod implicit, terminalul curent execută comanda introdusă, citind eventualele date de la tastatură și afișând eventualele rezultate pe ecran. Pe durata execuției comenzii, terminalul este blocat (nu se mai pot tasta comenzi până ce comanda curentă nu se termină).

Fiecare comandă executată returnează un cod de ieșire (eng. *exit code*). Convenția este ca o comandă care s-a executat cu succes să returneze codul 0, iar în caz contrar să returneze o valoare nenulă.

Dacă se dorește execuția comenzii fără blocarea terminalului (în *background*), la finalul liniei se scrie caracterul '`&`'.

Exemplu: Comanda de mai jos lansează editorul `gedit` pentru a edita fișierul `test.c` în background (terminalul poate accepta noi comenzi în timpul editării).

Exemplu &

```
gedit test.c &
```

Se pot scrie mai multe comenzi pe aceeași linie, separate prin caracterul '`;`'. În acest caz, comenzile se vor executa succesiv, în ordinea în care au fost date. Mai multe comenzi se pot separa și prin:

- '`|`' (operatorul *pipe*): output-ul primei comenzi este folosit ca input pentru cea de-a doua;
- '`&&`' (operatorul *AND*): a doua comandă se execută doar dacă prima comandă s-a executat cu succes (*exit code*-ul este 0, considerat TRUE);
- '`||`' (operatorul *OR*): a doua comandă se execută doar dacă prima comandă nu s-a executat cu succes (*exit code*-ul este diferit de 0, considerat FALSE).

Exemple: În prima linie din exemplele de mai jos, comanda `grep` extrage din fișierul `test.c` doar liniile care conțin subșirul "`example`". Aceste linii extrase ajung ca input la comanda `wc -l` (`wc`=*word count*, `-l`=*lines*), care numără câte astfel de linii avem. Pe ecran se va vedea doar numărul de linii. În a doua linie fișierul `test.c` se compilează folosind programul `gcc`, iar executabilul obținut se rulează doar dacă compilarea s-a efectuat cu succes.

Exemple comenzi multiple
```
grep example test.c | wc -l
gcc test.c -o test && ./test
```

Input-ul sau output-ul unei comenzi pot fi redirectate cu ajutorul operatorilor:

- '<' (redirectare *standard input*, adică ceea ce ar fi introdus de la tastatură);
- '>' (redirectare *standard output*, adică ceea ce ar fi afișat pe ecran, ca mesaj normal);
- '2>' (redirectare *standard error*, adică ceea ce ar fi afișat pe ecran, ca mesaj de eroare).

Exemplu: Prima comandă din cele de mai jos, citește linie cu linie din fișierul "file.txt" (folosit ca redirectare a intrării standard) și afișează liniile citite pe ecran. Al doilea exemplu folosește `ls` pentru a lista conținutul folderului `/etc`. În loc să se afișeze pe ecran, conținutul respectiv se scrie în fișierul `contents.txt`. Mesajele de eroare vor fi ignorate prin redirectarea acestora în fișierul `/dev/null` (ce poate fi privit ca o "gaură neagră" a sistemului de fișiere).

Exemplu redirectare
```
while read line; do echo $line; done < file.txt$
ls /etc > contents.txt 2> /dev/null
```

2.2.1 Comenzi uzuale

În laboratorul anterior s-au discutat câteva comenzi de bază, pentru familiarizarea cu terminalul. Pe lângă acestea, mai avem următoarele comenzi care deasemenea se utilizează în mod frecvent. Sintaxa completă a acestora se găsește în paginile de manual ale sistemului (rulați `man nume_comandă`).

cp (*copy*) copiază unul sau mai multe fișiere dintr-o locație în alta.

Exemplu cp
```
cp /etc/passwd my_passwd_copy.txt
```

Comanda de mai sus copiază fișierul `/etc/passwd` (cale absolută, deoarece începe cu `'/'`) în folderul curent, cu numele `my_passwd_copy.txt`. Primul parametru al comenzii este fișierul sursă, iar al doilea este fișierul destinație.

mv (*move*) mută unul sau mai multe fișiere dintr-o locație în alta.

Exemplu mv
```
mv my_passwd_copy.txt users_list.txt
```

În exemplul de mai sus, fișierul `my_passwd_copy.txt` este mutat în `users_list.txt`. Operația de mutare poate fi privită ca o operație de redenumire. La fel ca la comanda `cp`, primul parametru este sursa, iar al doilea este destinația.

less ne permite să vizualizăm în terminal un fișier text.

Exemplu less
```
less users_list.txt
```

`less` permite vizualizarea într-un mod interactiv, folosind tastele săgeți, `"Page Up"`, `"Page Down"`, etc. Pentru a încheia vizualizarea se apasă tasta `"q"`.

grep caută un anumit conținut într-un fișier.

Exemple grep
```
grep root users_list.txt
grep -v root users_list.txt
```

În primul exemplu se afișează liniile care conțin string-ul `"root"` din fișierul `users_list.txt`. În al doilea exemplu se afișează liniile care nu conțin string-ul căutat (parametrul `-v` înseamnă inversarea căutării).

wc (*word count*) numără cuvinte, linii sau caractere într-un fișier.

Exemple wc

```
wc users_list.txt
wc -l users_list.txt
wc -w users_list.txt
wc -m users_list.txt
```

Primul exemplu afișează atât numărul de linii, cât și numărul de cuvinte, respectiv caractere din fișierul `users_list.txt` primit ca parametru. Următoarele exemple folosesc opțiunile `-l`, `-w` și `-m` pentru a afișa aceste informații pe rând.

cut ajută la filtrarea informației dintr-un output.

Exemple cut

```
cut -d: -f 1,6 /etc/passwd
ls -l | cut -c 2,3,4
```

În primul exemplu, folosim comanda `cut` ca să extragem câmpurile 1 și 6 (opțiunea *fields* `-f 1,6`) de pe fiecare linie a fișierului `/etc/passwd`, considerând că separatorul de câmpuri este `":"` (opțiunea *delimiter* `-d:`).

În al doilea exemplu, output-ul comenzii `ls -l` este filtrat de `cut`, extrăgând al 2-lea, al 3-lea și al 4-lea caracter de pe fiecare linie (opțiunea *characters* `-c 2,3,4`). Aceste caractere corespund permisiunilor proprietarului pentru fiecare fișier.

find caută fișiere într-o ierarhie de directoare.

Exemple find

```
find . -name "*.txt"
find /usr/bin -type f -perm 0755
```

Primul argument reprezintă directorul în care să se facă căutarea. În primul exemplu, se face căutarea în directorul curent (`.`), iar în al doilea în directorul `/usr/bin`. În primul exemplu, căutăm toate elementele care se termină cu `".txt"`. În al doilea exemplu, căutăm doar fișiere (`-type f`) care au permisiunile `0755`.

2.2.2 Variabile

În interacțiunea cu terminalul, putem utiliza variabile pentru a reține anumite valori (de tip string) și pentru a le folosi în alte comenzi. Pentru a inițializa o variabilă, scriem numele acesteia, urmat de simbolul `"="`, urmat de valoare pe care vrem să o atribuim (fără spații înainte sau după `"="`). Pentru a folosi variabila respectivă, numele acesteia trebuie prefixat cu simbolul `"$"`. În exemplul de mai jos, variabila `NAME1` se inițializează cu string-ul `"Tom"`, apoi se folosește în comanda `echo`, împreună cu variabila `NAME2` (se va afișa `Hello Tom and Bill`).

Exemplu inițializare și utilizare variabilă

```
NAME1=Tom
export NAME2=Bill
echo "Hello $NAME1 and $NAME2"
```

Variabilele `NAME1` și `NAME2` sunt asociate doar terminalului curent. Într-un alt terminal, acestea nu mai sunt vizibile. Dacă din terminalul curent este pornit un proces nou, acesta moștenește `NAME2` (deoarece a fost exportată), dar nu și `NAME1`.

2.2.2.1 Variabile de mediu

Pe lângă variabilele definite de utilizator, terminalul mai utilizează și variabile de mediu (eng. *environment variables*), ce pot fi privite ca niște variabile globale, la nivelul terminalului.

Exemple de variabile de mediu:

- `USER` - conține numele utilizatorului curent
- `HOME` - conține calea spre folderul personal al utilizatorului curent
- `PATH` - conține o listă de căi în sistemul de fișiere, separate prin `":"`. Atunci când se rulează o comandă în terminal, fără a i se specifica calea absolută, executabilul acesteia se caută în fiecare folder din `PATH` pe rând, utilizându-se primul în care executabilul se găsește.

Pentru a vizualiza toate variabilele de mediu pentru terminalul curent, se poate rula comanda `printenv`.

2.2.3 Fișiere de comenzi

Fișierele de comenzi, numite și *script*-uri conțin o secvență de comenzi, care se execută pe rând. Principala utilizare a acestora este automatizarea unor sarcini. Pe lângă execuția de comenzi, un script mai poate conține și structuri de control (`if`, `while`, `for`) și chiar proceduri.

2.2.3.1 Crearea și execuția

Pentru a crea un script, pur și simplu creăm un fișier (prin convenție, script-urile shell au extensia `.sh`, dar acest lucru nu este obligatoriu) și scriem în el comenzi. Script-urile nu se compilează, ci se execută direct, fiind interpretate de către shell. Pentru editare, putem folosi un editor în mod text (ex. `nano`, `vim`) sau unul vizual (ex. `gedit`, `sublime`). În exemplul de mai jos se creează fișierul `myscript.sh` folosind comanda `touch`, care apoi se deschide în editorul `gedit` în mod asincron (pentru a nu ține blocat terminalul).

Crearea și editarea unui script

```
touch myscript.sh
gedit myscript.sh &
```

Ca să executăm un script, fie îl dăm ca parametru interpretorului de comenzi (`bash`), fie folosim `chmod` ca să îl facem executabil, apoi îl rulăm direct.

Execuția unui script - varianta 1

```
bash myscript.sh
```

Execuția unui script - varianta 2

```
chmod +x myscript.sh
./myscript.sh
```

Comanda `chmod` trebuie rulată o singură dată pentru acordarea dreptului de execuție, chiar dacă script-ul se modifică. La rularea directă, ar trebui să specificăm și ce interpretor trebuie folosit. Pentru asta, prima linie a script-ului poate avea o construcție specială, numită *shebang*. Linia va începe cu `"#!"` (simbolul `"#"` se folosește în mod normal pentru comentarii, în script-uri) și se va continua cu calea către interpretor, precum în exemplul de mai jos (linia 1).

```
1 #!/bin/bash
2
3 grep nologin /etc/passwd > file1.txt
4 COUNT1=`wc -l file1.txt | cut -d" " -f 1`
5 echo "We have $COUNT1 users that cannot login."
```

În linia 3 a script-ului se rulează comanda `grep`, care caută textul `"nologin"` în fișierul `/etc/passwd` și salvează liniile care îl conțin în fișierul `file1.txt`. Pe linia 4 se rulează o comandă care are ca output numărul de linii a fișierului, iar rezultatul se salvează în variabila `COUNT1`. Pentru ca variabila să rețină rezultatul comenzii, nu textul acesteia, s-a pus comand între apostroafe inverse (`"‘"` - tasta de sub *Esc*). La linia 5 se afișează un text ce conține variabila `COUNT1`.

2.2.3.2 Utilizarea argumentelor

Un script poate primi argumente în linia de comandă, pe care le poate accesa prin parametrii poziționali `$1`, `$2`, ... `${10}`, ... `$#` indică numărul de argumente primite, iar `$@` va conține toată lista de parametri.

În exemplul de mai jos se vede un exemplu de folosire a argumentelor primite în linia de comandă.

```
echo "We have $# arguments. The first one is $1."
```

2.2.3.3 Structuri de control

În fișiere de comenzi se pot folosi structuri de control, ca și în alte limbaje de programare. În continuare, se va prezenta sintaxa pentru `if` și `for`.

```
if cond1
then
    cmd_list1
elif cond2
then
    cmd_list2
else
    cmd_list3
fi
```

La `if`, clauzele `elif` și `else` sunt opționale. În exemplul de mai jos, se verifică dacă primul argument primit în linia de comandă este mai mare decât 3.

```
1 if test $1 -gt 3
2 then
3     echo "Greater"
4 else
5     echo "Smaller or equal"
6 fi
```

Pentru verificarea condiției am folosit comanda `test`. Nu putem folosi operatorul `>` pentru comparație, deoarece acesta se folosește deja la redirectarea output-ului. În schimb, vom folosi `-gt` (eng. *greater than*).

Sintaxa pentru `for` este ilustrată mai jos.

```
for name in list
do
    command_list
done
```

În exemplul de mai jos se parcurg toate fișierele cu extensia `".txt"` din directorul curent (linia 1), li se calculează dimensiunea în bytes (`-b`) folosind comanda `du` (linia 3), iar dacă aceasta este mai mare decât argumentul primit (linia 4), se afișează numele fișierului (linia 6).

```
1 for FNAME in *.txt
2 do
3     FSIZE=`du -b $FNAME | cut -f 1`
4     if test $FSIZE -gt $1
5     then
6         echo $FNAME
7     fi
8 done
```

2.3 Mersul lucrării

2.3.1 Probleme rezolvate

1. Dacă rulăm al doilea exemplu pentru comanda `cut` din secțiunea 2.2.1 observăm că prima linie nu corespunde formatului prezentat, deoarece comanda `ls -l` afișează pe prima linie un total. Modificați comanda, astfel încât să se afișeze rezultatul dorit.

   Soluția 1: Putem insera o comandă `grep`, care să elimine liniile care încep cu `"total"`. Comanda se va modifica în:

   ```
   ls -l | grep -v "^total" | cut -c 2,3,4
   ```

   Parametrul `-v` pentru `grep` afișează liniile care nu conțin textul căutat, iar caracterul `"^"` indică potrivirea doar la începutul unei linii (nu dorim să eliminăm, de exemplu, fișierele al căror nume conține șirul de caractere "`total`").

   Soluția 2: Folosim comanda `tail` cu parametrul `-n +2`, care afișează toate liniile primite, începând de la a 2-a:

   ```
   ls -l | tail -n +2 | cut -c 2,3,4
   ```

2. Scrieți un script care afișează argumentele primite în linia de comandă, în ordine inversă.

   Soluție: Pentru a parcurge argumentele primite în lina de comandă, se folsește următoarea buclă:

   ```
   1 for ARG
   2 do
   3     echo $ARG
   4 done
   ```

   Problema este că în codul de mai sus, argumentele se parcurg în ordine, iar noi avem nevoie de ele în ordine inversă. Prin urmare, vom construi un string și vom insera fiecare argument, pe rând, în fața șirului, pentru a obține ordinea inversă. La final se afișează string-ul construit.

   ```
   1 RESULT=""
   2 for ARG
   3 do
   4     RESULT="$ARG $RESULT"
   5 done
   6 echo $RESULT
   ```

2.3.2 Probleme propuse

1. (15p) Afișați pe ecran doar linia din `/etc/passwd` care corespunde utilizatorului curent.
2. (15p) Afișați liniile dintr-un fișier care nu conțin nici string-ul `"text1"`, nici string-ul `"text2"`.
3. (20p) Scrieți o comandă care numără câte elemente se găsesc într-un director. Câte dintre acestea sunt la rândul lor directoare?
4. (25p) Scrieți un script care afișează toate fișierele cu extensia `.c` din folderul curent, care au peste 20 de linii. Modificați script-ul astfel încât să fie luate în considerare și fișierele din subdirectoare (recursiv). *Indiciu:* comanda `find` parcurge folderele recursiv.
5. (25p) Scrieți un script care primește ca argument în linia de comandă unul din cuvintele `"read"`, `"write"` sau `"execute"`. Pentru fiecare fișier din directorul curent care are permisiunea respectivă se va crea un director cu același nume, la care se adaugă extensia `.dir`. În directorul respectiv se va scrie un fișier numite `"size.txt"` ce va conține dimensiunea fișierului.

Laborator 3

Dezvoltarea și depanarea programelor C sub Linux

3.1 Scopul lucrării

În acest laborator se va discuta dezvoltarea programelor C pe Linux, precum și rularea și depanarea acestora.

3.2 Fundamente teoretice

3.2.1 Compilarea și rularea programelor

Limbajul C este un limbaj compilat, ceea ce înseamnă că pentru a rula un program C, acesta trebuie întâi compilat. În urma rulării se obține un program executabil.

Cel mai folosit compilator de C pe Linux este `gcc` (*Gnu Compiler Collection*), care poate fi apelat precum în exemplul de mai jos, pentru a compila fișierul `program.c`.

Compilarea unui program C

```
gcc -Wall program.c -o program
```

Pe lângă numele fișierului `.c` dat ca argument, se mai folosesc opțiunile :

- `-Wall` (*warnings=all*) Această opțiune activează afișarea tuturor avertismentelor (*eng. warning*) compilatorului. Deși un program se poate compila și cu avertismente, orice programator "care își merită pâinea" le va rezolva sau va configura compilatorul să le ignore explicit.
- `-o program` (*output*) Această opțiune ne permite să alegem numele programului executabil care va rezulta în urma compilării. Prin convenție, programele executabile pe Linux nu au extensie.

Observație: comanda minimală pentru a compila fișierul `program.c` este `gcc program.c`. În acest caz, programul compilat se va numi `a.out`.

Ca să rulăm programul pe care tocmai l-am compilat, din folderul curent, vom rula:

Rularea unui program din folderul curent

```
./program
```

Motivul pentru care adăugăm `./` înainte numelui programului e pentru a indica interpretorului de comenzi faptul că acesta se găsește în folderul curent. În mod normal, interpretorul de comenzi caută programele în folderele din variabila de mediu `$PATH`.

În cazul în care programul se blochează (cel mai probabil dintr-o eroare de programare), se poate opri folosind combinația de taste `Ctrl+C`.

3.2.2 Depanarea programelor

De multe ori, un program nu funcționează conform așteptărilor, din cauza unor erori de programare. Dacă prin inspecția codului nu se identifică sursa problemei, procesul de depanare (*eng. debugging*) se poate dovedi util.

3.2.2.1 Compilarea pentru depanare

Ca să beneficiem de cât mai multă informație în momentul în care depanăm un program, acesta trebuie compilat folosind opțiunea `-g`. Prin adăugarea acestei opțiuni, în program se adaugă informații legate de codul sursă, putând de exemplu să inspectăm valoarea unei variabile în timpul depanării.

Pentru depanarea propriu-zisă, se rulează depanatorul `gdb`, primind ca argument numele programului. În consola interactivă care apare se pot rula comenzi specifice depanatorului, ce vor fi prezentate în secțiunea următoare.

Depanarea unui program cu gdb

```
gdb program
```

3.2.2.2 Comenzi date la depanare

Depanatorul `gdb` încarcă programul și îl suspendă înainte de execuție, dându-ne ocazia să rulăm comenzi de depanare. Printre comenzile uzuale, suportate de `gdb`, avem:

- `break source.c:lineNo` (setarea unui *breakpoint*) - se va suspenda execuția programului atunci când se ajunge la linia `lineNo` din fișierul sursă `source.c`;
- `run` - pornește execuția programului; eventualele argumente ale programului se transmit aici;
- `continue` - reia execuția programului, care continuă până la întâlnirea unui *breakpoint*;
- `step` - execută următoarea linie de cod;
- `next` - execută următoarea linie de cod, la fel ca `step`, dar dacă aceasta este un apel de funcție, se va face tot apelul într-un singur "pas";
- `backtrace` - afișează stiva de apeluri în punctul curent al programului;
- `print var` - afișează valoarea variabilei `var`.

3.2.2.3 Depanarea post-mortem

Pentru a investiga un program care "crapă" (*eng. crash*) în timpul execuției, e util să examinăm starea programului din momentul respectiv.

La fel ca la depanarea normală, programul se compilează în prealabil cu opțiunea `-g`, pentru adăugarea de informații referitoare la codul sursă.

Pentru a activa salvarea stării unui program atunci când crapă, se va rula comanda:

Activarea core dump

```
ulimit -c unlimited
```

Comanda de mai sus are efect doar pentru terminalul curent, în care a fost rulată (se rulează o singură dată).

Se rulează apoi programul, reproducând condițiile care l-au făcut să crape. După crash, se va observa apariția unui fișier numit `core` în folderul curent, care se poate încărca în `gdb`, rulând comanda:

Încărcarea unui core dump în gdb

```
gdb program core
```

În comanda de mai sus, `program` este numele programului care a crăpat.

La încărcarea unui astfel de *core dump*, execuția programului este "înghețată". Putem examina stiva de apeluri sau examina valorile variabilelor, folosind comenzile `backtrace` sau `print`, dar nu putem avansa execuția folosind `continue`, `step` sau `next`.

În multe situații, la depanarea post-mortem putem observa o problemă tipică cum ar fi, de exemplu, dereferențierea unui pointer `NULL` sau o variabilă neinițializată (are o valoare care pare aleatoare). Rețineți totuși că linia care conține codul greșit poate să nu fie aceeași cu linia la care a crăpat programul.

3.2.2.4 Depanarea prin mesaje

Unii programatori preferă să afișeze mesaje în diverse puncte ale programului pentru a trasa execuția acestuia și a depista problemele. Deși utilizarea unui *debugger* este preferabilă, dacă totuși folosiți această metodă ar trebui să țineți cont de următoarele aspecte:

- Mesajele afișate pentru depanare ar trebui scrise într-un fișier separat, sau în `stderr`, pentru a le separa de conținutul afișat în mod normal de program.
- Apelul unor funcții de afișare cum ar fi `printf()` sau `fprintf()` scrie textul de afișat într-un buffer, dar e posibil ca acesta să ajungă pe ecran sau în fișier cu întârziere. Putem forța afișarea apelând funcția `fflush()`.
- Unele probleme se reproduc doar pe calculatorul unor clienți, nu și pe calculatorul pe care se testează. În astfel de cazuri, afișarea de informații într-un fișier jurnal (*eng. log*) poate fi singura variantă de a investiga problema.

3.2.3 Depistarea problemelor de tip *memory leak*

O problemă comună la dezvoltarea de programe într-un limbaj care nu beneficiază de un modul de colectare și eliberare automatizată a memoriei ce nu mai este folosită (*garbage collector*) este omiterea dealocării memoriei alocate. Utilitarul `valgrind` ne permite să depistăm astfel de probleme. Cel mai simplu mod de utilizare este de a rula `valgrind` având ca prim argument numele programului urmat, eventual, de argumentele acestuia.

Testarea unui program cu valgrind

```
valgrind program arg1 arg2
```

Testând precum mai sus, utilitarul `valgrind` ne anunță dacă s-au găsit probleme de tip *memory leak*. Dacă dorim să se afișeze liniile la care s-a alocat memorie care nu a mai fost eliberată, se va rula:

Testarea unui program cu valgrind

```
valgrind --leak-check=full --show-leak-kinds=all program arg1 arg2
```

Testarea unui program cu `valgrind` poate fi mai lentă (sau mult mai lentă, în special cu opțiunile de afișare a liniilor cu probleme) decât rularea normală a unui program.

Rețineți că putem avea probleme de tipul *memory leak* alocând direct memorie (cu `malloc()`) sau apelând funcții care alocă memorie intern (de exemplu la deschiderea unui fișier sau director).

3.3 Mersul lucrării

3.3.1 Întrebări recapitulative

1. Cu ce comandă se compilează un program C sub Linux?
2. De ce e nevoie să scriem `./` înaintea numelui unui program pentru a-l rula?
3. Ce este un breakpoint în contextul depanării?
4. Ce opțiune trebuie să adăugăm la comanda de compilare pentru ca executabilul să conțină și informații de debug?
5. Ce face utilitarul *valgrind*?

3.3.2 Probleme propuse

Primele patru probleme propuse presupun corectarea unui cod existent în timp ce ultimele două presupun scrierea de programe noi. Programele care trebuie corectate se găsesc la adresa `https://github.com/cypryoprisa/os-lab-examples/tree/master/l03`.

1. (20p) În *l03p1.c* se încearcă generarea tuturor submulțimilor mulțimii $\{0, 1, \ldots, N-1\}$, unde N este primit ca argument în linia de comandă, folosind operații pe biți. Reparați erorile de compilare astfel încât programul să poată fi compilat și să funcționeze corect.

2. (20p) În *l03p2.c* se citesc de la tastatură două mulțimi S_1 și S_2 ca vectori ordonați și se calculează raportul dintre cardinalul intersecției și cardinalul reuniunii, $\frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$ (numit și similaritate Jaccard). Corectați codul astfel încât rezultatul să fie corect.

3. (20p) În *l03p3.c* se citește de la tastatură un șir și se calculează suma numerelor prime din acesta. Programul crapă în timpul rulării. Utilizând tehnica post-mortem debugging prezentată mai sus, identificați și reparați problemele.

4. (25p) Programul din *l03p4.c* gestionează în memorie un vector de studenți, pe care se pot face diverse operații prin comenzi de la tastatură (*add*, *del*, *list*, *exit*). Deși funcționează corect, programul are memory leak-uri. Folosiți programul `valgrind` pentru a le identifica și reparați-le.

5. (30p) Implementați și testați funcția
   `int insert(int *v1, int n1, int c1, int *v2, int n2, int pos);`
   care inserează vectorul `v2` de lungime `n2` în vectorul `v1` de lungime `n1` și capacitate `c1`, pe poziția `pos`. Funcția va returna `0` în caz de succes și `-1` în cazul în care capacitatea `c1` nu este suficient de mare.
   Exemplu: dacă `v1`, de capacitate cel puțin 8 conține elementele $1, 2, 3, 4, 5, 6$, iar `v2` conține elementele $10, 20$, inserarea făcându-se pe poziția 3, `v1` va deveni $1, 2, 3, 10, 20, 4, 5, 6$.

6. (30p) Implementați un program care primește un întreg fără semn (`unsigned int`) ca argument în linia de comandă, îl parsează (argumentele în linia de comandă sunt string-uri, iar noi avem nevoie de numere) și inversează primul byte cu ultimul și al 2-lea cu penultimul.

Laborator 4

Fișiere

4.1 Scopul lucrării

Laboratorul curent prezintă principalele apeluri de sistem în Linux pentru manipularea fișierelor.

4.2 Fundamente teoretice

4.2.1 Conceptul de *file descriptor*

Sistemul de operare asignează fiecărui fișier deschis un descriptor sau identificator, reprezentat ca un întreg. De fiecare dată când se face un apel de sistem ce creează sau deschide un fișier, sistemul de operare returnează un astfel de descriptor. Descriptorii de fișiere sunt unici la nivel de proces (program).

Prin convenție, primii 3 descriptori de fișiere sunt standard:

- 0: *standard input* (ex.: funcția `scanf` citește din standard input);
- 1: *standard output* (ex.: funcția `printf` scrie în standard output);
- 2: *standard error* (ex.: funcția `perror` scrie în standard error).

Există cinci apeluri de sistem ce creează descriptori: `creat`, `open`, `fcntl`, `dup` și `pipe`.

4.2.2 Crearea, deschiderea și închiderea fișierelor

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

int creat(const char *pathname, mode_t mode);

int close(int fd);
```

4.2.2.1 open

Pentru a deschide un fișier se folosește apelul de sistem `open`. În prima variantă (cu doi parametri), primul parametru reprezintă numele fișierului sau calea spre acesta (absolută sau relativă la folderul curent), iar al doilea indică felul în care vrem să deschidem fișierul (citire, scriere, etc.). Al treilea parametru se folosește doar la crearea unui fișier nou și indică permisiunile de acces.

Argumentul `flags` reprezintă o mască de biți (fiecare bit setat indică un anumit *flag*), ce poate fi o combinație între următoarele valori (definite în `fcntl.h`):

- `O_RDONLY` - deschide fișierul pentru citire;
- `O_WRONLY` - deschide fișierul pentru scriere;
- `O_RDWR` - deschide fișierul pentru citire și scriere;
- `O_APPEND` - deschide fișierul pentru scriere, dar dacă avem ceva scris în el, se va scrie doar după sfârșitul fișierului, prin adăugare;
- `O_CREAT` - în cazul în care fișierul nu există deja, este creat;
- `O_EXCL` - dacă folosim `O_CREAT` și fișierul există deja, apelul de sistem va eșua;
- `O_TRUNC` - dacă fișierul are deja un conținut, acesta se va șterge.

Există și alte flag-uri ce pot fi folosite la deschiderea unui fișier. Pentru o listă completă a lor, precum și pentru alte detalii legate de apelul de sistem `open`, utilizați comanda `man 2 open`.

În cazul folosirii argumentului `mode`, acesta indică permisiunile fișierului creat și este construit tot ca o mască de biți, prin combinarea (SAU pe biți) următoarelor valori (definite în `sys/stat.h`):

- `S_IRUSR`, `S_IWUSR`, `S_IXUSR` - *read*, *write*, *execute* pentru *owner*-ul fișierului;
- `S_IRGRP`, `S_IWGRP`, `S_IXGRP` - *read*, *write*, *execute* pentru membrii grupului din care face parte *owner*-ul fișierului;
- `S_IROTH`, `S_IWOTH`, `S_IXOTH` - *read*, *write*, *execute* pentru restul utilizatorilor.

Alternativ, se poate folosi pentru `mode` reprezentarea ca număr octal a permisiunilor de acces, ca de exemplu `0644`.

Rezultatul returnat este un descriptor de fișier valid, în caz de succes, sau valoarea -1, în caz de eșec.

4.2.2.2 creat

Apelul de sistem `creat` este utilizat pentru a crea un fișier și este similar cu a apela `open`, cu flag-urile `O_WRONLY | O_CREAT | O_TRUNC`. Argumentele `pathname` și `mode` au aceeași semantică și aceleași valori posibilie ca și la `open`.

Valoarea returnată este, de asemenea, un descriptor de fișier valid, în caz de succes sau -1, în caz de eșec.

4.2.2.3 close

Apelul de sistem `close` realizează închiderea fișierului al cărui descriptor a fost primit ca parametru. Va returna 0, în caz de succes și -1, în caz de eșec.

4.2.2.4 Exemplu

Dorim să scriem un program care face o copie a fișierului `/usr/include/stdlib.h` în folderul curent. Vom începe prin a deschide și închide fișierele, utilizând apelurile de sistem prezentate mai sus.

```
1  int main()
2  {
3      int fd1 = -1, fd2 = -1;
4
5      fd1 = open("/usr/include/stdlib.h", O_RDONLY);
6      if(fd1 == -1) {
7          perror("Could not open input file");
8          return 1;
9      }
10
```

```
11      fd2 = creat("copy_stdlib.h", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
12      if(fd2 == -1) {
13          perror("Could not open output file");
14          close(fd1);
15          return 1;
16      }
17
18      ...
19
20      close(fd1);
21      close(fd2);
22
23      return 0;
24  }
```

Programul începe prin a declara variabilele `fd1` și `fd2` pentru cei doi descriptori de fișier (fișierul de intrare și fișierul de ieșire), la linia 3.

La linia 5 se deschidere fișierul de intrare pentru citire, folosind flag-ul `O_RDONLY`. La linia 11 se deschide fișierul de ieșire pentru scriere. Deoarece e posibil ca acest fișier să nu existe încă, vom folosi apelul de sistem `creat` în loc de `open` (se putea deschide și cu `open`, folosind flag-urile `O_WRONLY`, `O_CREAT` și `O_TRUNC`).

După fiecare deschidere de fișier este important să verificăm valoarea descriptorului de fișier. Valoarea -1 indică eșecul. În acest caz, operația curentă nu mai poate continua. Înainte de ieșire vom afișa un mesaj de eroare folosind funcția `perror` (liniile 7 și 13), iar dacă avem deja descriptori de fișier deschiși se vor închide (linia 14).

Funcția `perror` va afișa string-ul primit ca argument, urmat de motivul ultimei erori. De exemplu, dacă fișierul de intrare nu ar exista, apelul de `open` de la linia 5 ar returna -1, iar la linia 7 s-ar afișa "`Could not open input file: No such file or directory`".

Înainte de finalul programului, este important să închidem descriptorii de fișiere utilizați, utilizând apelul de sistem `close` (liniile 20 și 21).

Pentru a realiza copierea, nu este suficient să deschidem fișierele. În secțiunea următoare vom studia apelurile de sistem necesare pentru citirea și scrierea unui fișier deschis.

4.2.3 Citirea și scrierea

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

4.2.3.1 read

Apelul de sistem `read` citește de la poziția curentă din fișierul deschis `fd` un număr `count` de octeți pe care îi va scrie în buffer-ul `buf`. După citire, poziția curentă în fișier va avansa cu numărul de octeți citiți.

Apelul `read` va returna numărul de octeți citiți în caz de succes sau valoarea -1 în caz de eroare. Se poate întâmpla ca numărul de octeți citiți să nu corespundă cu valoarea parametrului `count`. De exemplu, putem încerca să citim 8 octeți, dar până la finalul fișierului să mai fie doar 3. În acest caz, `read` va returna valoarea 3. Dacă se returnează valoarea 0, înseamnă că poziția curentă în fișier era la sfârșitul fișierului înaintea execuției apelului `read` și, prin urmare, nu s-a citit nimic.

4.2.3.2 write

Apelul de sistem `write` scrie în fișierul deschis `fd` un număr `count` de octeți din buffer-ul `buf`. După scriere, poziția curentă va avansa cu numărul de octeți scriși.

Apelul `write` va returna numărul de octeți scriși în caz de succes sau valoarea -1 în caz de eroare. Dacă se returnează o valoare mai mică decât parametrul `count`, cel mai probabil discul e plin și nu s-au putut scrie toate datele.

4.2.3.3 Exemplu

Vom continua exemplul din secțiunea 4.2.2.4, adăugând cod pentru copierea conținutului din primul fișier în al doilea.

```
1  #define BUFF_SIZE 64
2
3  int main()
4  {
5      int fd1 = -1, fd2 = -1;
6      ssize_t size = 0;
7      int err = 0;
8      unsigned char buff[BUFF_SIZE];
9
10     ... //the code for opening the files
11
12     for(;;) {
13         size = read(fd1, buff, BUFF_SIZE);
14         if(size < 0) {
15             //read error
16             perror("Could not read from input file");
17             err = 1;
18             break;
19         } else if(size == 0) {
20             //end of file
21             break;
22         } else {
23             //size may be smaller than BUFF_SIZE
24             size = write(fd2, buff, size);
25             if(size <= 0){
26                 perror("Could not write to output file");
27                 err = 2;
28                 break;
29             }
30         }
31     }
32
33     ... //the code for closing the files
34
35     if(err == 0) {
36         printf("File copied successfully.\n");
37     } else {
38         printf("Some error occured while copying.\n");
39     }
40
41     return 0;
42 }
```

Copierea propriu-zisă a conținutului se face în bucla de la liniile 12–31. La fiecare iterație, se citește un bloc de dimensiune `BUFF_SIZE` din primul fișier (linia 13), bloc care se scrie în al doilea fișier (linia 24).

La ambele apeluri de sistem, dacă valoarea returnată este negativă vom afișa un mesaj de eroare, vom seta o variabilă care să indice eroarea, apoi vom părăsi bucla (liniile 15–18 și 26–28). În mod normal, se va ieși din buclă prin `break`-ul de la linia 21, atunci când se ajunge la finalul fișierului de intrare, iar `read` returnează 0.

4.2.4 Deplasarea cursorului în fișiere

Fiecărui fișier deschis îi corespunde un cursor care indică poziția curentă, de unde se va citi la următorul `read` sau se va scrie la următorul `write`. Poziția cursorului este incrementată implicit în urma operațiilor de `read` și `write` sau poate fi modificată explicit prin apelul sistem `lseek`.

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

Apelul de sistem `lseek` primește ca argument un descriptor de fișier `fd`, un deplasament `offset` (pozitiv sau negativ) și un indicator `whence`, care spune de unde se calculează deplasamentul. Parametrul `whence` poate lua următoarele valori:

- `SEEK_SET` - deplasamentul se consideră de la începutul fișierului;
- `SEEK_CUR` - deplasamentul se consideră de la poziția curentă;
- `SEEK_END` - deplasamentul se consideră de la finalul fișierului.

Valoarea returnată reprezintă poziția absolută a cursorului în fișier.

4.2.4.1 Exemplu: determinarea dimensiunii unui fișier

Ideea de bază pentru a determina dimensiunea unui fișier (numărul de octeți din acesta) este să ne deplasăm la finalul și să aflăm poziția cursorului.

```
1    ...
2    int fd;
3    off_t size;
4
5    fd = open(...);
6    if(fd == -1) { ... }
7
8    size = lseek(fd, 0, SEEK_END);
9    printf("The file size is %ld.\n", size);
10   ...
```

Codul de mai sus începe prin a deschide fișierul și a verifica dacă deschiderea s-a efectuat cu succes (liniile 5–6). Apelul de sistem către `lseek` se face la linia 8, unde se cere deplasarea cursorului exact la finalul fișierului. Apelul returnează poziția cursorului în fișier după deplasare, adică exact poziția de la final.

Tipul de date `off_t` depinde de platformă (trebuie să fie suficient de mare ca să poată indica o poziție într-un fișier), de obicei fiind definit ca un `long int`. Din acest motiv, la afișare folosim formatul `%ld`.

4.2.4.2 Exemplu: copierea cifrelor la finalul unui fișier

Dorim să scriem un program care citește un fișier caracter cu caracter, scriind cifrele la finalul fișierului.

```
1    int fd;
2    off_t size, crtPos, i;
3    char c = 0;
4    ...
5    fd = open(..., O_RDWR);
6    if(fd == -1) { ... }
7
8    size = lseek(fd, 0, SEEK_END);
9    lseek(fd, 0, SEEK_SET);
```

```
10
11      for(i=0; i<size; i++) {
12          if(read(fd, &c, 1) != 1) {
13              perror("Reading error");
14              break;
15          }
16          if(c >= '0' && c <= '9') {
17              crtPos = lseek(fd, 0, SEEK_CUR);
18              lseek(fd, 0, SEEK_END);
19              if(write(fd, &c, 1) != 1) {
20                  perror("Writing error");
21                  break;
22              }
23              lseek(fd, crtPos, SEEK_SET);
24          }
25      }
26
27      close(fd);
```

Ca de obicei, se deschide întâi fișierul și se verifică dacă deschiderea s-a făcut cu succes (liniile 5–6). În acest exemplu vom folosi pentru deschidere flag-ul `O_RDWR`, deoarece dorim să efectuăm atât operațiile de citire, cât și cele de scriere pe același fișier.

La linia 8, determinăm dimensiunea inițială a fișierului, ca și în exemplul anterior, apoi readucem cursorul la începutul fișierului (linia 9).

Deoarece se cunoaște dimensiunea fișierului, putem realiza citirea într-o buclă cu număr fix de pași (liniile 11–25). La linia 12 se citește un singur caracter, direct în variabila `c`. Dacă caracterul citit este cifră (conform verificării de la linia 16), vom efectua următorii pași:

- salvăm poziția curentă în variabila `crtPos` (linia 17), pentru că după ce se scrie cifra la finalul fișierului, trebuie să știm unde ne vom întoarce pentru a citi următorul caracter;
- mutăm cursorul la finalul fișierului (linia 18);
- scriem cifra citită (linia 19) și verificăm dacă scrierea s-a efectuat cu succes;
- mutăm cursorul înapoi pe poziția `crtPos` (linia 23), pregătind fișierul pentru următoarea citire.

4.3 Mersul lucrării

4.3.1 Întrebări recapitulative

1. Ce apel de sistem folosește intern funcția `printf`?
2. Ce apeluri de sistem se pot folosi pentru a crea un fișier?
3. Ce returnează apelurile de sistem `read` și `write`?
4. Cum aflăm dimensiunea unui fișier deja deschis, utilizând `lseek`?
5. Presupunând că fișierul `file.txt` are dimensiunea 100 de octeți iar buffer-ul `buff` 50 de octeți, care va fi dimensiunea fișierului după ce se execută codul de mai jos?

```
1   int fd = open("file.txt", O_WRONLY);
2   lseek(fd, 80, SEEK_SET);
3   write(fd, buff, 50);
4   close(fd);
```

4.3.2 Probleme rezolvate

1. (*l04p1_copy.c*) Să se scrie un program care face o copie, în folderul curent, a fișierului `/usr/include/stdlib.h`. Programul este descris în secțiunile 4.2.2.4 și 4.2.3.3.

2. (*l04p2_size.c*) Să se scrie un program care primește ca argument numele sau calea unui fișier și afișează dimensiunea acestuia în octeți. Programul este descris în secțiunea 4.2.4.1.

3. (*l04p3_digits.c*) Să se scrie un program care primește ca argument numele sau calea unui fișier și copiază la finalul fișierului toate cifrele găsite în el. Programul este descris în secțiunea 4.2.4.2.

4.3.3 Probleme propuse

1. (25p) Implementați funcția
   `int get_line(int fd, int lineNr, char *line, int maxLength);`
   care citește linia cu numărul `lineNr` din fișierul deschis cu descriptorul `fd`.
   Pentru a delimita liniile din fișier, țineți cont că acestea se termină cu caracterul `'\n'`. Bufferul `line` în care se va scrie linia citită are dimensiunea `maxLength`, în care trebuie să încapă inclusiv terminatorul de string (un octet cu valoarea 0).
   Funcția va returna 0 în caz că citirea s-a efectuat cu succes și valori negative pentru cazurile de eroare (erori la citire, linie prea lungă, fișierul are mai puțin de `lineNr` linii, etc.).
   Scrieți de asemenea un program care primește ca argument un nume de fișier și un număr de linie și apelează funcția `get_line`.

2. (25p) Scrieți un program care citește un fișier și construiește un alt fișier cu conținutul inversat. Numele fișierului sursă și al fișierului destinație se primesc ca argumente în linia de comandă.

3. (25p) Scrieți un program care citește un fișier și construiește un alt fișier cu liniile inversate (prima linie se inversează cu ultima, a doua cu penultima, etc.). Pentru punctaj maxim trebuie ținut cont de faptul că o linie poate fi oricât de lungă. Numele fișierului sursă și al fișierului destinație se primesc ca argumente în linia de comandă.

4. (25p) Scrieți un program care inserează un string într-un fișier, pe o poziție dată. De exemplu, dacă fișierul are conținutul `abcdefghijkl` și se dorește inserarea string-ului `''XYZ''` pe poziția 4, fișierul va avea conținutul `abcdXYZefghijkl`.

Laborator 5

Directoare

5.1 Scopul lucrării

Laboratorul curent prezintă principalele apeluri de sistem în Linux pentru manipularea directoarelor.

5.2 Fundamente teoretice

5.2.1 Apeluri de sistem pentru lucrul cu directoare

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
int closedir(DIR *dirp);
struct dirent *readdir(DIR *dirp);
void rewinddir(DIR *dirp);
```

5.2.1.1 opendir și closedir

Apelul de sistem `opendir` este utilizat pentru deschiderea unui director. Parametrul `name` reprezintă calea relativă sau absolută spre directorul care se dorește deschis. Rezultatul apelului este un pointer la un *directory stream*, de tip `DIR*`. Acest stream are asociat un cursor, care ne permite să parcurgem directorul element cu element.

La deschiderea unui director, sistemul de operare alocă anumite structuri interne, care se vor elibera cu apelul de sistem `closedir`.

5.2.1.2 readdir

Pentru a citi următorul element dintr-un director se folosește apelul de sistem `readdir`, care primește ca parametru valoarea de tip `DIR*` returnată de `opendir`. Valoarea returnată este un pointer la o structură de tip `struct dirent`. Această structură poate să difere de la un sistem de operare la altul, dar standardul POSIX garantează că va conține cel puțin membrul `d_name`, indicând numele elementului.

5.2.1.3 rewinddir

Dacă dorim să mutăm cursorul asociat *directory stream*-ului la începutul unui director deschis, putem apela `rewinddir`, dând ca parametru stream-ul de tip `DIR*` returnat de `opendir`.

5.2.1.4 Exemplu

Dorim să scriem o funcție C care primește ca argument calea spre un director și îi listează conținutul.

```
1  int listDir(const char *path)
2  {
3      DIR *dir = NULL;
4      struct dirent *entry = NULL;
5
6      dir = opendir(path);
7      if(dir == NULL) {
8          perror("Could not open directory");
9          return -1;
10     }
11     while((entry = readdir(dir)) != NULL) {
12         printf("%s\n", entry->d_name);
13     }
14     closedir(dir);
15     return 0;
16 }
```

Funcția începe prin a deschide directorul, la linia 6. În cazul în care deschiderea directorului a eșuat, `opendir` returnează `NULL`. În acest caz, vom afișa motivul erorii și vom ieși din funcție.

Citirea propriu-zisă a directorului se face în bucla de la liniile 11–13. Bucla apelează `readdir` la fiecare iterație, pentru a obține următorul element din director. Atunci când se ajunge la finalul directorului, `readdir` va returna `NULL`. După citire, structura `entry` va conține printre altele, numele fișierului în câmpul `d_name`, care se va afișa.

Înainte de finalul funcției, directorul trebuie închis, apelând `closedir` (linia 14).

5.2.2 Utilizarea link-urilor

Un link sau legătură este un tip de fișier care face legătura cu un alt fișier. Pe sisteme de tip UNIX (inclusiv Linux), avem două tipuri de link-uri:

- *hard link* - un fișier obișnuit asociat aceluiași i-node. După ce este creat, un hardlink nu se mai deosebește de fișierul original. Atunci când ultima legătură se șterge, se șterge și i-node-ul, împreună cu conținutul.
- *symbolic link* - este un fișier special care indică locația altui fișier. Dacă fișierul original se șterge, symlink-ul devine invalid.

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
int symlink(const char *target, const char *linkpath);
```

Apelul de sistem `link` este utilizat pentru a crea hardlink-uri, iar `symlink` pentru a crea link-uri simbolice. Ambele apeluri de sistem primesc ca prim argument calea spre un fișier existent, iar al doilea argument reprezintă calea spre link-ul ce se dorește a fi creat.

5.2.3 Extragerea de informații despre fișiere

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *pathname, struct stat *statbuf);
```

```
int fstat(int fd, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
```

Pentru a obține metadate despre un fișier (sau director) putem folosi unul din apelurile de sistem `stat`, `fstat` sau `lstat`. Toate trei primesc în al doilea parametru un pointer la o structură de tipul `struct stat`, în care vor completa informații despre fișier. În caz de succes se va returna 0, altfel -1.

Apelurile `stat` și `lstat` primesc ca prim parametru numele sau calea fișierului, în timp ce `fstat` primește un descriptor de fișier.

Diferența dinte `stat` și `lstat` apare doar în cadrul legăturilor simbolice. Dacă avem un fișier `f1` care este de fapt o legătură simbolică spre fișierul `f2`, `stat` apelat pe `f1` va obține informațiile despre `f2`, în timp ce `lstat` va obține informațiile despre `f1`.

Printre metadatele prezente într-un `struct stat`, întâlnim:

- `off_t st_size` - pentru fișiere obișnuite, indică dimensiunea fișierului în octeți;
- `mode_t st_mode` - indică tipul fișierului și permisiunile.

Pentru a determina tipul unui fișier, putem aplica unul din macro-urile de mai jos pe câmpul `st_mode` dintr-un `struct stat`:

- `S_ISREG` - returnează o valoare nenulă dacă avem un fișier obișnuit (*regular*);
- `S_ISDIR` - returnează o valoare nenulă dacă avem un director;
- `S_ISLNK` - returnează o valoare nenulă dacă avem o legătură simbolică.

Tot cu ajutorul câmpului `st_mode` se pot identifica și permisiunile unui fișier. Mai exact, cei mai puțin semnificativi 9 biți ai acestui câmp se referă la permisiuni. Pentru a extrage doar acei biți, se poate aplica operația `&` (AND logic) cu masca `0777` (valoarea 777 în octal, care corespunde la 9 biți de 1).

5.2.3.1 Exemplu: afișarea tipului și a permisiunilor fișierelor

Vom extinde exemplul din secțiunea 5.2.1.4 prin afișarea tipului, pentru fiecare fișier listat.

```
1  ...
2  char filePath[512];
3  struct stat statbuf;
4  ...
5  while((entry = readdir(dir)) != NULL) {
6      snprintf(filePath, 512, "%s/%s", path, entry->d_name);
7      if(lstat(filePath, &statbuf) == 0) {
8          if(S_ISREG(statbuf.st_mode)) {
9              printf("[reg] ");
10         } else if(S_ISDIR(statbuf.st_mode)) {
11             printf("[dir] ");
12         } else if(S_ISLNK(statbuf.st_mode)) {
13             printf("[lnk] ");
14         } else {
15             printf("[unk] ");
16         }
17         printf("[%o] ", statbuf.st_mode & 0777);
18     } else {
19         printf("[err] ");
20     }
21     printf("%s\n", entry->d_name);
22 }
23 ...
```

Bucla de parcurgere a fișierelor (liniile 5–22) va începe prin construcția căii complete spre fișier (linia 6), prin concatenarea căii spre folderul curent cu separatorul `/` și cu numele fișierului curent. Pentru această operație vom utiliza funcția `snprintf`, care "afișează" date în mod similar cu `printf`, dar într-un string. Preferăm să folosim `snprintf` în loc de `sprintf`, deoarece prima variantă primește ca argument și lungimea string-ului destinație și garantează că nu o va depăși.

Apelul de sistem `lstat` de la linia 7 are nevoie de calea completă spre fișier și nu doar de numele acestuia. De exemplu, dacă vrem să aflăm informații despre fișierul `f1`, care se află în folderul `d1/d2` (cale relativă la folderul curent) și apelăm `lstat("f1", ...)` în loc de `lstat("d1/d2/f1", ...)`, se vor căuta metadate despre `f1` în folderul curent, în loc să se caute în `d1/d2`.

Dacă `lstat` s-a executat cu succes, va întoarce valoarea 0, caz în care ne putem baza că s-a completat câmpul `st_mode` din structura `statbuf` transmisă ca pointer. La liniile 8, 10 și 12 se folosesc macro-urile prezentate în secțiunea anterioară pentru a decide dacă fișierul este unul obișnuit, un director, sau respectiv o legătură simbolică. Cele trei tipuri nu sunt exhaustive, există și alte tipuri posibile pentru un fișier.

La linia 17 afișăm permisiunile fișierului, aplicând masca de biți `0777` pe câmpul `st_mode`. Rezultatul se afișează folosind specificatorul `%o` (octal).

5.2.3.2 Exemplu: listarea recursivă a unui folder

În exemplele anterioare s-a afișat doar conținutul direct al unui folder, fără a se intra în subfoldere. În acest exemplu vom lista recursiv fiecare subfolder al folderului dat.

```
1  void listRec(const char *path)
2  {
3      DIR *dir = NULL;
4      struct dirent *entry = NULL;
5      char fullPath[512];
6      struct stat statbuf;
7
8      dir = opendir(path);
9      if(dir == NULL) {
10         perror("Could not open directory");
11         return;
12     }
13     while((entry = readdir(dir)) != NULL) {
14         if(strcmp(entry->d_name, ".") != 0
15                 && strcmp(entry->d_name, "..") != 0) {
16             snprintf(fullPath, 512, "%s/%s", path, entry->d_name);
17             if(lstat(fullPath, &statbuf) == 0) {
18                 printf("%s\n", fullPath);
19                 if(S_ISDIR(statbuf.st_mode)) {
20                     listRec(fullPath);
21                 }
22             }
23         }
24     }
25     closedir(dir);
26 }
```

Parcurgerea recursivă a unui director are aceeași structură ca și parcurgerea obișnuită: se deschide directorul (linia 8), se citesc intrările într-o buclă (liniile 13–24), iar la final se închide (linia 25).

O diferență importantă este tratarea subfolderelor speciale `.` și `..` aflate implicit în orice folder. Aceste subfoldere ne duc la folderul curent, respectiv la folderul părinte. Dacă am apela recursiv și pe aceste subfoldere, algoritmul de parcurgere nu s-ar mai termina. Din acest motiv, în fiecare iterație a buclei se verifică dacă numele elementului curent e diferit de `.` și de `..` (liniile 14–15). Pentru simplitate s-a folosit funcția `strcmp`, care returnează 0 dacă două string-uri sunt egale.

Iterația continuă cu construcția căii complete spre fișier (linia 16), din motivul explicat în secțiunea 5.2.3.1. Această cale va fi dată ca parametru apelului de sistem `lstat` (linia 17) și va fi afișată (linia 18).

Pentru a lista și conținutul subfolderelor, fiecare intrare trebuie verificată dacă este folder sau nu (linia 19). În caz că este folder, pur și simplu vom apela recursiv funcția `listRec()` pe calea completă spre subfolder.

Observație: Exemplul de mai sus va lista doar elementele a căror cale nu depășește 512 caractere.

5.3 Mersul lucrării

5.3.1 Întrebări recapitulative

1. Ce rol au elementele `"."` și `".."` din fiecare director?
2. Ce funcții se folosesc pentru a citi conținutul unui director? Dar pentru a-i scrie conținutul?
3. Cu ce apel de sistem pentru operații pe fișiere este echivalent `rewinddir`?
4. Care este diferența dinte un *hard link* și un *symbolic link*?
5. Dându-se o cale în sistemul de fișiere, cum aflăm dacă elementul respectiv este un fișier sau director?
6. În ce câmp dintr-o structură de tip `struct stat` se găsesc permisiunile unui fișier?

5.3.2 Probleme rezolvate

1. (*l05p1_list.c*) Să se scrie un program care primește ca argument în linia de comandă calea spre un folder și listează conținutul acestuia, afișând pentru fiecare element tipul acestuia (fișier, folder sau legătură simbolică), respectiv permisiunile. Programul este descris în secțiunile 5.2.1.4 și 5.2.3.1.
2. (*l05p2_list_rec.c*) Implementați un program care primește ca argument în linia de comandă calea spre un director și afișează recursiv conținutul acestuia. Programul este descris în secțiunea 5.2.3.2.

5.3.3 Probleme propuse

1. (35p) Implementați funcția `off_t dirSize(const char* dirPath)` care primește ca argument calea spre un director și returnează dimensiunea acestuia, ca suma tuturor fișierelor obișnuite din acesta, inclusiv cele din subfoldere. Dimensiunea unui symlink se consideră 0.
   Bonus (20p): Dacă printre fișierele din director se află mai multe hard link-uri către același i-node, dimensiunea acestuia se va aduna o singură dată.
2. (35p) Scrieți un program care primește ca argument în linia de comandă calea spre un folder, numele unui fișier și un string. Programul va căuta recursiv în folder fișierele cu numele dat, ce conțin string-ul primit ca al treilea argument. Pentru fișierele găsite se vor crea legături simbolice în folderul curent, adăugând o extensie incrementală fiecărei legături (prima legătură va avea `.1` la final, a doua `.2`, ș.a.m.d.).
   Observație: Pentru punctaj parțial se poate implementa doar căutarea fișierelor după nume, fără a mai căuta string-ul dat în conținut.
3. (35p) Scrieți un program care primește ca argument numele unui folder și îl șterge împreună cu tot conținutul acestuia.
   Indicații: Consultați paginile de manual pentru apelurile de sistem `unlink` (pentru ștergerea unui fișier) și `rmdir` pentru ștergerea unui director gol.

Laborator 6

Procese

6.1 Scopul lucrării

Laboratorul curent prezintă conceptul de proces în Linux, împreună cu cele mai folosite apeluri de sistem pentru manipularea proceselor.

6.2 Fundamente teoretice

6.2.1 Conceptul de proces

Un proces este o entitate ce reprezintă un program care rulează. Pe lângă codul programului, procesul mai conține și date, descriptori de fișiere deschise, fire de execuție, precum și orice alte resurse care țin de execuția programului.

Fiecare proces are un identificator unic, numit PID (eng. *process identifier*), care poate fi folosit pentru a interacționa cu procesul.

Cu excepția procesului cu PID-ul 0, fiecare proces are un părinte (procesul care l-a creat). Relația tată-fiu generează în mod implicit o structură de arbore pentru procesele din sistem.

În cazul în care un proces (părinte) își încheie execuția în timp ce unii din copiii săi încă rulează, spunem că aceștia devin *orfani*. Copii orfani sunt "adoptați" în mod automat de procesul cu PID-ul 1.

În cazul în care un proces (copil) își încheie execuția în timp ce părintele lui încă se execută (și nu așteaptă după terminarea copilului respectiv), procesul va fi pus în starea "*zombie*", adică proces terminat, dar pentru care se păstrează încă informații. Acele informații sunt șterse și, implicit, procesul zombie șters, când părintele se termină sau cere sistemului de operare informații despre copilul terminat.

6.2.2 Crearea de procese și obținerea identificatorilor

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);

pid_t getpid(void);
pid_t getppid(void);
```

6.2.2.1 fork

În Linux, `fork` este singurul apel de sistem care dă naștere unui nou proces. La apelul acestuia se va crea un proces copil, identic cu părintele (atât din punctul de vedere al codului, cât și al resurselor), care va continua execuția imediat după apelul de sistem. Singura diferență este valoarea returnată de `fork()`. Astfel, în procesul copil se va returna 0, pe când în procesul părinte se va returna PID-ul procesului copil. În caz de eroare (nu se poate crea procesul fiu), se returnează -1.

6.2.2.2 getpid și getppid

Apelurile de sistem `getpid` și `getppid` sunt utilizate pentru a obține PID-ul propriu, respectiv PID-ul procesului părinte. Tipul returnat de ambele apeluri de sistem este `pid_t`.

6.2.2.3 Exemplu

Dorim să scriem un program care dă naștere unui proces fiu. Procesul părinte își va afișa PID-ul propriu și pe cel al copilului, în timp ce procesul fiu își va afișa PID-ul propriu și pe cel al părintelui.

```
1  int main(void)
2  {
3      pid_t pid;
4
5      pid = fork();
6      if(pid == -1) {
7          perror("Could not create child process");
8          return -1;
9      } else if(pid == 0) {
10         //this code is executed only by child
11         printf("[CHILD] My PID is %d. My parent's PID is %d.\n",
12             getpid(), getppid());
13     } else {
14         //this code is executed only by parent
15         printf("[PARENT] My PID is %d. I created a child with PID %d.\n",
16             getpid(), pid);
17         wait(NULL);
18     }
19     return 0;
20 }
```

Programul începe prin a declara variabila `pid`, de tip `pid_t` (linia 3). Acest tip, în practică, este un întreg cu semn (`int`). La linia 5 se efectuează apelul de sistem, al cărui rezultat se salvează în variabila `pid`. De aici încolo, se execută în paralel atât procesul părinte, cât și procesul copil.

Se verifică întâi dacă apelul de sistem a reușit. În caz contrar, valoarea returnată este -1 (linia 6) și se iese cu un mesaje de eroare (linia 7).

În cazul în care apelul de sistem a reușit, valoarea returnată este fie 0, caz în care ne aflăm în procesul copil, fie un număr mai mare decât 0, caz în care ne aflăm în procesul părinte.

În cazul în care ne aflăm în procesul copil, se afișează PID-ul propriu și cel al părintelui, valori obținute apelând `getpid()` și `getppid()` (liniile 11-12).

În cazul procesului părinte (liniile 14–17), PID-ul propriu se obține tot apelând `getpid()`, în timp ce PID-ul copilului a fost returnat de `fork()`, deci se găsește în variabila `pid`. Procesul părinte va aștepta și terminarea execuției copilului, apelând `wait()` (linia 17), a cărui funcționalitate va fi descrisă în secțiunea următoare.

La execuția programului se va afișa un text similar cu cel de mai jos.

```
[PARENT] My PID is 14982. I created a child with PID 14983.
[CHILD] My PID is 14983. My parent's PID is 14982.
```

Valorile PID-urilor pot fi diferite de la un sistem la altul și de la o rulare la alta. De asemenea, ordinea în care se afișează cele două linii poate să difere. În schimb, valorile de pe prima linie trebuie să fie aceleași cu valorile de pe a doua linie, inversate.

6.2.3 Terminarea proceselor și așteptarea terminării copiilor

```
#include <sys/types.h>
#include <sys/wait.h>
```

```
#include <unistd.h>
#include <stdlib.h>

void exit(int status);

pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

6.2.3.1 exit

Pentru a închide procesul curent se poate folosi apelul de sistem `exit`. Acesta primește ca parametru un număr care indică starea programului la ieșire. Valorile valide pentru parametrul `status` sunt numere de la 0 la 255. Prin convenție, starea 0 indică faptul că execuția s-a încheiat cu succes, în timp ce alte valori indică erori.

6.2.3.2 wait și waitpid

În mod normal, un proces părinte creează copii pentru a le delega unele sarcini și de obicei așteaptă terminarea execuției acestora.

Apelul de sistem `waitpid` primește ca parametru PID-ului procesului copil pentru care se așteaptă terminarea. Opțional, se primește și un pointer la o variabilă de tip `int`, prin parametrul `wstatus` în care se va salva starea cu care procesul copil s-a terminat (dacă nu avem nevoie de stare, transmitem `NULL`). Parametrul `options` se folosește pentru a indica niște condiții în care să nu se aștepte terminarea copilului (pentru mai multe detalii consultați pagina de manual) și are valoarea 0 dacă dorim să așteptăm tot timpul.

Apelul de sistem `wait` are un comportament similar cu `waitpid`, dar nu se specifică un copil anume, ci se va aștepta terminarea oricărui copil.

Pentru a folosi starea cu care s-a terminat procesul copil, nu vom folosi direct valoarea completată în parametrul `wstatus`, ci o vom extrage din acesta utilizând macro-ul `WEXITSTATUS`.

6.2.3.3 Exemplu

Un proces părinte va declara două variabile `x` și `y` pe care le va inițializa. Același proces va da naștere la 3 procese copii, fiecare din acestea efectuând una dintre operațiile *adunare*, *scădere*, respectiv *înmulțire* între cele două variabile și va transmite rezultatul părintelui folosind starea sau codul de ieșire (en. *exit status*). Înainte să facă calculul, fiecare proces copil va aștepta un timp aleator, cel mult o jumătate de secundă.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <sys/wait.h>
5  #include <unistd.h>
6
7  int main(void)
8  {
9      pid_t pid1=-1, pid2=-1, pid3=-1;
10     int x=5, y=3, z=0;
11     int status1=0, status2=0;
12
13     pid1 = fork();
14     if(pid1 == 0) {
15         //1st child
16         srand(getpid());
17         usleep(random() % 500000);
18         z = x + y;
19         exit(z);
20     }
```

```
21      pid2 = fork();
22      if(pid2 == 0) {
23          //2nd child
24          srand(getpid());
25          usleep(random() % 500000);
26          z = x - y;
27          exit(z);
28      }
29      pid3 = fork();
30      if(pid3 == 0) {
31          //3rd child
32          srand(getpid());
33          usleep(random() % 500000);
34          z = x * y;
35          exit(z);
36      }
37      waitpid(pid2, &status1, 0);
38      printf("The result of the subtraction is %d.\n", WEXITSTATUS(status1));
39      wait(&status1);
40      wait(&status2);
41      printf("The result of the other two operations are %d and %d.\n",
42          WEXITSTATUS(status1), WEXITSTATUS(status2));
43
44      return 0;
45  }
```

Părintele începe prin inițializarea variabilelor `x` și `y` (linia 10), variabile ce vor fi moștenite și de copii. Pentru a crea fiecare proces copil se apelează `fork()` (liniile 13, 21 și 29).

Fiecare din cei 3 copii trebuie să aștepte un timp aleator. Pentru a genera numere aleatoare, de obicei se inițializează generatorul de numere aleatoare folosind funcția `srand()`, o singură dată, la începutul programului. Apoi, de fiecare dată când e nevoie de un număr aleator se apelează funcția `rand()`. Numerele aleatoare generate nu sunt cu adevărat aleatoare, ci sunt determinate de starea generatorului. De obicei se apelează `srand(time(NULL))` la începutul programului, astfel la fiecare rulare a programului generatorul va fi inițializat cu altă valoare (în funcție de timpul curent). Dacă am folosi această abordare, cei 3 copii ar moșteni aceeași stare a generatorului și ar genera același timp de așteptare. Prin urmare, atunci când lucrăm cu numere aleatoare în procesele copii, ar trebui ca fiecare proces să își inițializeze generatorul de numere aleatoare. Deasemenea, nu putem folosi `time(NULL)` (timpul curent, în secunde) ca valoare de inițializare, deoarece sunt șanse ca toate cele trei procese copil să apeleze în aceeași secundă și să inițializeze cu aceeași valoare. Ar trebui să inițializăm cu ceva unic pentru fiecare proces, cum ar fi PID-ul acestuia.

După ce fiecare proces copil își execută operația, va transmite rezultatul la părinte, folosind apelul de sistem `exit()` (liniile 19, 27 și 35).

Procesul părinte așteaptă întâi după al doilea copil, apelând `waitpid()` la linia 37. Aici, procesul părinte este blocat până ce al doilea copil își termină execuția. Chiar dacă un alt copil își termină mai repede execuția, părintele nu continuă până nu primește rezultatul de la al doilea copil (cel care a efectuat scăderea).

Pentru copiii 1 și 3 se așteaptă folosind apelul de sistem `wait` (liniile 39 și 40). Deoarece nu se știe care din cei doi copii își va încheia primul execuția (sau și-a încheiat-o deja, înainte de copilul 2), rezultatele pot să vină în orice ordine, deci `status1` poate fi rezultatul adunării iar `status2` rezultatul înmulțirii, sau invers.

6.2.4 Lansarea în execuție a programelor

```
#include <unistd.h>

int execl(const char *path, const char *arg, ... /* (char *) NULL */);
```

```
int execlp(const char *file, const char *arg, ... /* (char *) NULL */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

6.2.4.1 Familia de funcții exec

Familia de funcții `exec` are rolul de a **înlocui** imaginea procesului curent cu un alt program (fișier executabil), a cărui cale este transmisă ca argument. După execuția noului program, nu se revine în codul programului apelant. Deoarece nu se creează un proces nou, ci se înlocuiește cel curent, anumite resurse, cum ar fi descriptorii de fișiere deschise, se moștenesc.

Funcția `execl()` primește un număr variabil de argumente, primul fiind calea spre un executabil, iar celelalte sunt argumentele pe care executabilul le primește în linia de comandă, inclusiv numele acestuia (ca argument 0). Ultimul argument trebuie să fie `NULL`, deoarece funcțiile variadice în C (funcții cu număr variabil de argumente) nu știu câte argumente au primit și trebuie să își poată da seama unde să se oprească.

De exemplu, dacă vrem să scriem un program care afișează permisiunile fișierului primit ca parametru, putem executa programul `ls`, ca în exemplul de mai jos:

```
1    ...
2    execl("/bin/ls", "ls", "-l", argv[1], NULL);
3    perror("execl failed");
4    ...
```

Primul argument al funcției `execl` este calea spre executabilul `ls` (`"/bin/ls"`). Urmează numele comenzii (`"ls"`), argumentele acesteia (`"-l"` și `argv[1]`), respectiv santinela `NULL`. Spre deosebire de alte apeluri de sistem, nu e nevoie să verificăm valoarea returnată de `execl`. În caz de succes, programul primit ca parametru se execută fără să se revină din acesta. În caz de eroare, se execută pur și simplu linia următoare din program, deci dacă execuția a ajuns în acel punct, știm sigur că am avut de-a face cu o eroare.

Dacă în loc să specificăm calea completă spre executabilul `ls` i-am fi scris doar numele, funcția `execl` nu l-ar fi găsit. Pentru un comportament similar cu cel al interpretorului de comenzi (executabilul să fie căutat în variabila de mediu `PATH`), vom folosi funcția `execlp()`, care primește aceleași argumente ca și `execl()`. În acest caz, exemplul de mai sus ar deveni:

```
1    ...
2    execlp("ls", "ls", "-l", argv[1], NULL);
3    perror("execlp failed");
4    ...
```

Atunci când argumentele unui program se construiesc programatic trebuie să transmitem un vector cu acestea, în loc să utilizăm o funcție variadică. Funcțiile `execv` și `execvp` funcționează ca și `execl` și `execlp`, cu diferența că primesc un vector cu argumentele. Ultimul element al vectorului trebuie deasemenea să fie `NULL`. Exemplul de mai jos are același rezultat cu cele anterioare, dar utilizează funcția `execv`.

```
1    ...
2    char *arguments[4];
3    arguments[0] = "ls";
4    arguments[1] = "-l";
5    arguments[2] = argv[1];
6    arguments[3] = NULL;
7    execv("/bin/ls", arguments);
8    perror("execv failed");
9    ...
```

6.2.4.2 Exemplu: redirectare

Dorim să listăm toate elementele aflate în directorul rădăcină `/` (nerecursiv) într-un fișier primit ca argument în linie de comandă, folosind comanda `ls`. Practic vrem să scriem un program C, care realizează operația de redirectare din comanda `ls / > file.txt`.

Ne vom folosi de faptul că programele executate cu familia de funcții `exec` moștenesc fișierele deschise și descriptorii acestora.

```
1  int main(int argc, char **argv)
2  {
3      pid_t pid;
4      int fd, status;
5
6      if(argc != 2) {
7          fprintf(stderr, "Usage: %s <result_file>\n", argv[0]);
8          return 1;
9      }
10
11     pid = fork();
12     if(pid == -1) {
13         perror("Could not create child process");
14         return -1;
15     } else if(pid == 0) {
16         fd = open(argv[1], O_CREAT | O_TRUNC | O_WRONLY, 0644);
17         if(fd < 0) {
18             perror("Could not open the result file");
19             exit(1);
20         }
21         dup2(fd, 1); //use fd as standard output
22         //ls will inherit the standard output
23         execlp("ls", "ls", "/", NULL);
24         perror("execlp failed");
25         exit(2);
26     } else {
27         wait(&status);
28         if(WEXITSTATUS(status) == 0) {
29             printf("Command completed successfuly!\n");
30         } else {
31             printf("Command failed!\n");
32         }
33     }
34     return 0;
35 }
```

La liniile 6–9 se verifică faptul că programul a primit un argument în linia de comandă. La linia 11 se apelează `fork()`, dând naștere unui proces copil. În procesul copil `fork()` a returnat 0, deci codul executat este cel din liniile 16–25. În primul rând, se deschide fișierul primit ca argument în linia de comandă (linia 16). Chiar dacă acest argument a fost primit de părinte, copilul este în acest moment o clonă a părintelui, deci va avea aceleași valori ale argumentelor.

În continuare, vrem să redirecționăm conținutul care s-ar scrie în mod normal pe ecran către fișierul pe care tocmai l-am deschis. Pentru aceasta vom folosi apelul de sistem `dup2()` (linia 21). Acest apel de sistem duplică primul descriptor de fișier primit ca prim parametru (`fd`), utilizând descriptorul de fișier primit ca al doilea parametru. În acest caz, al doilea parametru este 1, deoarece vrem să folosim fișierul deschis pe post de ieșire standard (stdout), care are tot timpul descriptorul de fișier 1. Vechiul fișier indicat de descriptorul 1 va fi închis.

La linia 23 apelăm `execlp()` pentru a executa comanda dorită. În cazul în care `execlp()` nu se execută cu succes, afișăm eroarea (linia 24) și închidem procesul copil (linia 25).

Procesul părinte așteaptă terminarea copilului (linia 27) și, în funcție de starea de ieșire a acestuia, afișează dacă comanda s-a executat cu succes sau nu. Mesajele afișate de părinte vor apărea pe ecran, deoarece redirectarea ieșirii standard (adică a descriptorului de fișier 1) s-a făcut doar pentru procesul copil.

6.3 Mersul lucrării

6.3.1 Întrebări recapitulative

1. Ce returnează apelul de sistem `fork`?
2. Cum poate un proces să își afle PID-ul propriu? Dar PID-ul părintelui?
3. Cum se transmite starea de terminare a unui proces copil părintelui acelui copil?
4. Ce se întâmplă dacă un proces copil își încheie execuția înainte ca părintele să apeleze `wait`? Dar invers?
5. De ce execuția liniei imediat următoare de după o funcție din familia `exec` semnifică o eroare?
6. Care este diferența dintre primul și al doilea argument al funcției `execl`?
7. Dacă dorim să executăm comanda `cat` fără să știm exact unde se află programul respectiv în sistemul de fișiere, folosim funcția `execv` sau `execvp`? Justificați.

6.3.2 Probleme rezolvate

1. (*l06p1_parent_child.c*) Programul este descris în secțiunea 6.2.2.3.
2. (*l06p2_wait.c*) Programul este descris în secțiunea 6.2.3.3.
3. (*l06p3_redirect.c*) Programul este descris în secțiunea 6.2.4.2.

6.3.3 Probleme propuse

1. (25p) Implementați ierarhia de procese din Figura 6.1, în care programul principal P_1 dă naștere la două procese copil, P_2 și P_3, fiecare din acestea dând naștere unui nou proces copil, P_4, respectiv P_5. Fiecare proces își va afișa propriul PID, PID-ul părintelui, iar în caz că are copii, va aștepta terminarea acestora.

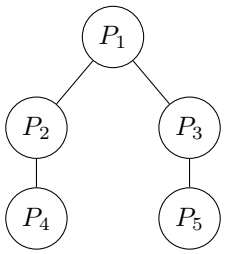


Figura 6.1: Ierarhia de procese

Pentru a vizualiza ierarhia de procese, apelați în P_4 și P_5 `sleep(60)`. Acest apel de sistem va ține procesele deschise timp de 1 minut. În acest timp, într-un alt terminal, rulați comanda `pstree -p <PID_P1>`, înlocuind `<PID_P1>` cu PID-ul afișat de procesul P_1.

2. (25p) Implementați două programe, *client.c* și *server.c*. Programul client afișează un prompt (`>`) și citește de la tastatură doi întregi și un caracter care poate fi `'+'` sau `'-'`. După citire se pornește un proces copil, care execută programul *server*, transmițându-i datele citite ca argumente în linia de comandă. Programul *server* va efectua operația primită (adunare sau scădere) pe cele două numere și va transmite rezultatul înapoi părintelui, folosind funcția `exit()` (presupunem că rezultatele sunt numere pozitive între 0 și 255). Părintele va aștepta terminarea copilului, va afișa rezultatul pe ecran și va afișa din nou prompt-ul.

3. (25p) Modificați codul de mai jos pentru a afișa numărul total de procese pornite. Fiecare copil va număra câte procese a creat (împreună cu descendenții săi) și va transmite acest număr părintelui prin exit status. Doar procesul inițial, va afișa pe ecran totalul. Propuneți o soluție care să funcționeze și pentru valori mai mari ale lui `n` (de exemplu 10).

```
1    int i, n=7;
2    for(i=0; i<n; i++) {
3        fork();
4    }
```

4. (25p) Scrieți un program care numără câte linii ce conțin secvența `"abcd"` are fișierul primit ca argument în linia de comandă, apelând `grep` și `wc`. Ar trebui să executați o comandă similară cu `grep abcd $FILENAME | wc -l`. Operatorul `|` (*pipe*) transmite output-ul primei comenzi la input-ul celei de-a doua. Vom folosi un fișier temporar pentru a salva acest output. Puteți utiliza apelul de sistem `dup2()`, ca să redirectați standard output-ul, respectiv standard input-ul unui program la un descriptor de fișier deschis.
Notă: Analogia cu operatorul *pipe* (`|`) nu este perfectă. Acest operator permite execuția în paralel a celor două programe, cel de-al doilea așteptând eventual ca primul să scrie date (se vor prezenta mai multe detalii la laboratorul 11). În cazul nostru, pentru ca numărarea liniilor să funcționeze corect trebuie ca al doilea program (`wc`) să fie rulat doar după ce primul (`grep`) își încheie execuția.

Laborator 7

Fire de execuție (thread-uri)

7.1 Scopul lucrării

Laboratorul curent prezintă conceptul de fir de execuție (eng. *thread*) în Linux. Vor fi prezentate apelurile de sistem pentru manipularea thread-urilor, precum și considerente practice în lucrul cu acestea.

7.2 Fundamente teoretice

În cadrul unui proces putem avea mai multe fire de execuție care rulează în paralel. Sistemul de operare ne permite să pornim un nou fir de execuție sau să așteptăm terminarea unuia.

7.2.1 Crearea unui thread și așteptarea terminării

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
```

Atunci când utilizăm funcții legate de thread-uri, pe lângă includerea `pthread.h`, trebuie să includem și biblioteca `pthread`, în etapa de *link*-are a programului. Pentru aceasta, la apelul compilatorului (`gcc`) vom adăuga la linia de comandă opțiunea `-pthread`. Această opțiune marchează în executabilul produs faptul că se folosește cod din această bibliotecă și cere sistemului de operare să o încarce la execuția programului.

Compilarea programelor cu pthread

```
gcc -Wall program.c -o program -pthread
```

7.2.1.1 pthread_create

Funcția `pthread_create()` ne permite să pornim un nou fir de execuție, care va executa codul funcției `start_routine`, transmisă prin al treilea argument.

Funcția executată de thread trebuie să aibă semnătura specificată, adică să primească un parametru de tip `void*`, iar rezultatul returnat să fie `void*`.

Primul argument al funcției `pthread_create()` este un pointer de tipul `pthread_t*`, de obicei adresa unei variabile de tipul respectiv, în care funcția va completa id-ul thread-ului nou creat, în caz de succes.

Al doilea argument este un pointer la o structură de tip `pthread_attr_t`, în care putem specifica atributele thread-ului. În cazul în care dorim să folosim atributele implicite, acest pointer poate fi `NULL`.

Argumentul al treilea este un pointer la funcția ce va fi executată de thread, așa cum s-a explicat anterior. Argumentul al 4-lea este argumentul transmis funcției de thread. Pentru a putea transmite orice date funcției, argumentul este un pointer generic (`void*`), putând să fie adresa oricărei structuri definite de programator.

În caz de succes, funcția `pthread_create()` returnează 0.

7.2.1.2 pthread_join

Funcția `pthread_join()` ne permite să așteptăm terminarea execuției unui thread. Primul argument va fi id-ul thread-ului, obținut prin apelul funcției `pthread_create()`. Al doilea argument va fi un pointer de tip `void**`, care va fi modificat pentru a indica spre rezultatul funcției thread-ului. Acest argument e un pointer la un pointer de tip `void*`, deoarece funcția thread-ului returnează un rezultat de tip `void*`, de obicei adresa unei variabile (sau structuri) alocate dinamic, în care sunt stocate rezultatele funcției.

7.2.1.3 Exemplu

Vom începe cu un exemplu de program în care se creează un thread și se așteaptă terminarea lui.

```
1  void *threadFn(void *unused)
2  {
3      printf("Secondary thread\n");
4      return NULL;
5  }
6
7  int main()
8  {
9      pthread_t tid = -1;
10
11     if(pthread_create(&tid, NULL, threadFn, NULL) != 0){
12         perror("Error creating thread");
13         return 1;
14     }
15     printf("Main thread\n");
16     pthread_join(tid, NULL);
17     return 0;
18 }
```

Programul începe prin a defini funcția thread-ului `threadFn`. Așa cum s-a explicat mai sus, o funcție care poate fi folosită ca funcție de thread trebuie să primească un argument de tipul `void*` și să returneze `void*`. În acest exemplu argumentul nu se folosește și se returnează tot timpul `NULL`.

În funcția principală, se apelează `pthread_create` pentru a porni thread-ul. Primul argument este un pointer la variabila `tid`, declarată la linia 9, în care se va stoca identificatorul thread-ului. Acest identificator poate fi folosit ulterior pentru operații pe thread, cum ar fi așteptarea terminării acestuia (la linia 16). Al treilea argument este funcția de thread, care va fi executată pe un fir de execuție separat, după cum se poate observa în Figura 7.1.

Firul principal de execuție merge de la începutul până la sfârșitul programului. Firul secundar începe de la crearea thread-ului și continuă până la terminarea execuției funcției thread-ului. Terminarea execuției nu trebuie neapărat să coincidă cu apelul funcției `pthread_join` pe thread-ul principal. În Figura 7.2 se pot observa două scenarii:

- În Figura 7.2(a), funcția thread-ului își încheie execuția înainte să se apeleze `pthread_join`. În acest caz, la join rezultatul va fi disponibil imediat și thread-ul principal poate continua.
- În Figura 7.2(b), funcția thread-ului își încheie execuția după ce se apelează `pthread_join`. În acest caz, thread-ul principal este blocat, întrerupându-și execuția până la încheierea funcției thread-ului.

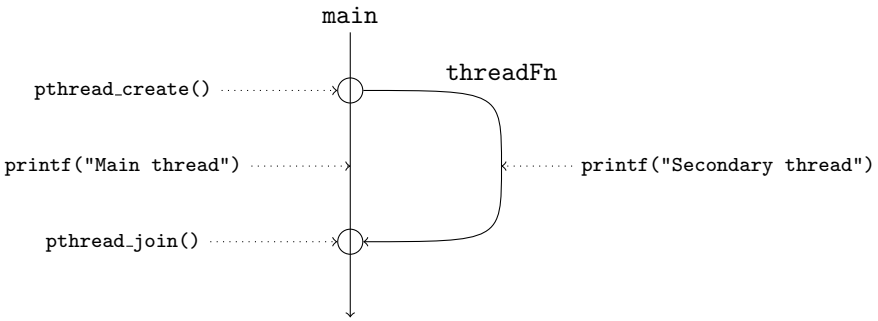
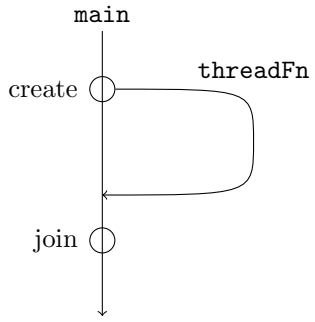
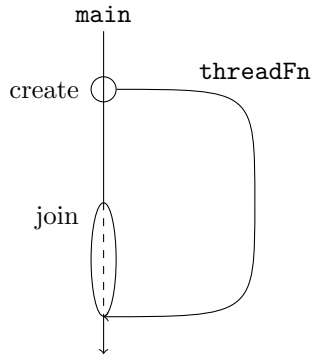


Figura 7.1: Ilustrarea execuției unui thread



(a) Thread-ul secundar își încheie execuția înainte ca thread-ul principal să apeleze `pthread_join`



(b) Thread-ul secundar își încheie execuția după ce thread-ul principal apelează `pthread_join`

Figura 7.2: Terminarea unui thread înainte sau după ce se apelează `pthread_join`

7.2.2 Transmiterea argumentelor unui thread

O funcție de thread va avea tot timpul prototipul `void *threadFn(void *)`, adică va primi întotdeauna un argument de tipul `void*`, iar rezultatul va fi tot de tipul `void*`. Acest tip oferă suficientă flexibilitate, deoarece un pointer de tip `void*` se poate converti în orice alt tip de pointer, inclusiv către o structură cu orice câmpuri am avea nevoie.

7.2.2.1 Exemplu

Dorim să scriem un program care generează un vector aleator, cu numere întregi și determină valoarea maximă din acesta folosind thread-uri.

Ideea de bază e să împărțim vectorul în mai multe bucăți de dimensiuni (aproximativ) egale, apoi fiecare thread să calculeze maximul dintr-o anumită bucată. Thread-ul principal va aștepta terminarea tuturor thread-urilor și va afișa valoarea maximă dintre maximele calculate de thread-uri.

7.2.2.2 Funcția de thread și parametri

```
1 typedef struct {
2     int *arr;
3     int from;
4     int to;
5 } TH_STRUCT;
6
7 void *thread_fn_max(void *param)
```

```
8  {
9      TH_STRUCT *s = (TH_STRUCT*)param;
10     int i, max;
11     max = s->arr[s->from];
12     for(i=s->from; i<=s->to; i++){
13         if(s->arr[i] > max){
14             max = s->arr[i];
15         }
16     }
17     return (void*)(long)max;
18 }
```

Funcția unui thread trebuie să primească cel puțin următoarele informații:

- `arr` - pointer spre vectorul pe care se lucrează;
- `from` - indicele de început în vector pentru bucata alocată thread-ului;
- `to` - indicele de final în vector pentru bucata alocată thread-ului.

Cum astfel de funcții primesc un singur parametru de tipul `void*`, vom defini o structură care conține informațiile dorite (liniile 1–5), iar funcția va primi ca parametru un pointer spre o astfel de structură.

În prima linie din funcție (linia 9) se face conversia de tip (eng. cast) a parametrului la pointer spre o structură `TH_STRUCT`, astfel încât să poată fi dereferențiat și să se poată extrage informațiile utile. În continuare, se parcurge vectorul între indicii `s->from` și `s->to`, calculându-se valoarea maximă în variabila `max`, de tip `int` (liniile 11–16). Această variabilă trebuie returnată de către funcție.

Funcția thread-ului returnează tipul `void*`, care pe 64 bit ocupă 8 octeți, în timp ce pe 32 bit ocupă 4 octeți. În ambele cazuri, un întreg (care ocupă 32 bit pe majoritatea platformelor) încape în capacitatea de reprezentare, trebuie doar să îi facem cast. Cast-ul de la linia 17 este unul dublu: `int`-ul se transformă întâi într-un `long`, apoi în `void*`. Dacă am fi transformat direct din `int` în `void*`, compilatorul ar fi afișat un warning de genul `"cast to pointer from integer of different size"`, deoarece pe 64 bit, un întreg și un pointer au dimensiuni diferite.

7.2.2.3 Generarea vectorului aleator

```
1  #define SIZE_ARRAY 1000
2
3  void generate_random_array(int *v, int size)
4  {
5      int i;
6      for(i=0; i<size; i++){
7          v[i] = rand() % 1000000;
8      }
9  }
10
11 int main()
12 {
13     int v[SIZE_ARRAY];
14     ...
15
16     srand(time(NULL));
17     generate_random_array(v, SIZE_ARRAY);
18     ...
19 }
```

În funcția `main` se apelează `srand(time(NULL))` pentru inițializarea generatorului de numere aleatoare (linia 16), apoi se apelează funcția `generate_random_array` care completează valori

aleatoare în elementele vectorului.

7.2.2.4 Împărțirea vectorului între thread-uri

```
1  #define SIZE_ARRAY 1000
2  #define NR_THREADS 6
3
4  int main()
5  {
6      ...
7      TH_STRUCT params[NR_THREADS];
8      int i;
9      ...
10
11     for(i=0; i<NR_THREADS; i++){
12         params[i].arr = v;
13         if(i == 0){
14             params[i].from = 0;
15         }else{
16             params[i].from = params[i-1].to + 1;
17         }
18         params[i].to = params[i].from + SIZE_ARRAY / NR_THREADS - 1;
19         if(i < SIZE_ARRAY % NR_THREADS){
20             params[i].to++;
21         }
22
23         ... //create thread
24     }
25     ...
26 }
```

Dorim să împărțim munca între thread-uri într-un mod cât mai echitabil. Dacă vectorul are `SIZE_ARRAY` elemente și avem `NR_THREADS` thread-uri, numărul de elemente procesate de fiecare thread ar trebui să fie `SIZE_ARRAY / NR_THREADS`. În cazul în care împărțirea nu se face exact (în exemplul curent, cu 1000 de elemente și 6 thread-uri, rămân 4 elemente neîmpărțite). Cel mai echitabil ar fi ca `SIZE_ARRAY % NR_THREADS` thread-uri să proceseze câte un element în plus.

În funcția `main` se declară vectorul `params`, ce va conține structurile cu parametri pentru fiecare thread. În bucla de la liniile 11–24 se setează valorile acestor parametri. Câmpul `arr` din fiecare structură va pointa spre vectorul `v` (linia 12). Primul thread va procesa elementele din vector începând cu indicele 0 (linia 14), iar celelalte thread-uri vor continua de la ultimul indice procesat de thread-ul anterior (linia 16). Capătul din dreapta al intervalului de indici va fi calculat adunând `SIZE_ARRAY / NR_THREADS` capătului din stânga (linia 18). În cazul în care suntem la primele `SIZE_ARRAY % NR_THREADS` thread-uri, adăugăm un element în plus la interval (liniile 19–21).

7.2.2.5 Lansarea și așteptarea thread-urilor

```
1  int main()
2  {
3      pthread_t tid[NR_THREADS];
4      TH_STRUCT params[NR_THREADS];
5      int i;
6      void *result;
7      int thMax, max = 0;
8      ...
9
10     for(i=0; i<NR_THREADS; i++){
```

```
11          ... // fill params[i]
12          pthread_create(&tid[i], NULL, thread_fn_max, &params[i]);
13      }
14      for(i=0; i<NR_THREADS; i++){
15          pthread_join(tid[i], &result);
16          thMax = (int)(long)result;
17          if(thMax > max){
18              max = thMax;
19          }
20      }
21      printf("The maximum value is %d.\n", max);
22
23      return 0;
24  }
```

Pentru a gestiona thread-urile cu care lucrăm, vom păstra identificatorul fiecăruia în vectorul `tid` declarat la linia 3. Structurile cu parametrii fiecărui thread au fost deja completate în secțiunea anterioară. Putem să creăm thread-urile în aceeași buclă în care se completează parametrii (liniile 10–13).

Funcția `pthread_create` primește ca prim argument un pointer spre `tid[i]` unde se va stoca id-ul thread-ului, `NULL` pentru atributele implicite, numele funcției de thread și un pointer la structura cu parametri `params[i]`.

Atunci când transmitem informații diferite thread-urilor, este foarte important ca fiecare thread să aibă parametrii într-o structură diferită, deoarece thread-urile primesc doar o referință la structură, nu o copie, și nu ar fi în regulă să transmiteam adresa unei structuri pe care o vom modifica.

După ce s-au creat thread-urile trebuie să se aștepte după ele, folosind `pthread_join` (linia 15). Rezultatul thread-ului este de tipul `void*`, așa că trebuie să folosim un pointer la o variabilă de acest tip. Ulterior, pointer-ul respectiv poate fi transformat într-un întreg (linia 16) și comparat cu maximul existent.

Este foarte important ca așteptarea după thread-uri să se facă într-o buclă separată față de cea în care se creează. În Figura 7.3(a) se ilustrează execuția thread-urilor din codul de mai sus, în paralel. În prima buclă se creează toate thread-urile, apoi în a doua buclă se așteaptă terminarea execuției tuturor. O variantă greșită ar fi ca imediat după ce se creează un thread, să apelăm `pthread_join` pentru el, în aceeași buclă *for*. În acest caz, comportamentul ar fi cel ilustrat în Figura 7.3(b), obținându-se o performanță similară cu apelul direct al funcțiilor, fără să se profite de paralelism.

7.2.3 Încheierea execuției unui thread

```
#include <pthread.h>

void pthread_exit(void *retval);
int pthread_cancel(pthread_t thread);

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

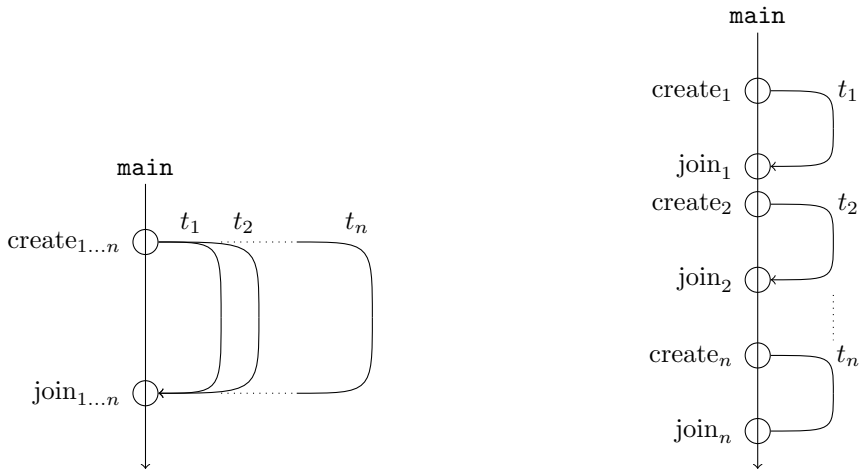
7.2.3.1 pthread_exit

Un thread își încheie execuția atunci când funcția acestuia se termină, sau prin apelul `pthread_exit`, care primește ca parametru valoarea returnată de thread.

7.2.3.2 pthread_cancel

Un thread poate fi anulat (închis) și de către alt thread din același proces, apelând `pthread_cancel`, cu identificatorul thread-ului ca parametru.

În general, nu se recomandă această închidere "forțată" a unui thread și se preferă, pe cât posibil ca thread-urile să își încheie singure execuția. Atunci când un thread este închis din exterior, acesta



(a) Thread-urile se execută în paralel, folosind bucle separate pentru `pthread_create` și `pthread_join`

(b) Thread-urile se execută secvențial, ca și cum s-ar apela funcția direct

Figura 7.3: Importanța utilizării buclelor separate pentru `pthread_create` și `pthread_join`

poate lăsa resurse folosite într-o stare inconsistentă (de exemplu un fișier deschis și neînchis, sau un vector cu valori completat doar parțial). Totuși există și situații când un thread trebuie să poată fi închis de către alt thread.

Atunci când se apelează `pthread_cancel`, thread-ul țintă nu se va încheia instant, ci doar în momentul în care acesta atinge un punct de terminare (eng. *cancellation point*), în anumite apeluri de sistem, cum ar fi `open`, `close`, `read`, `write`, `sleep` (pentru lista completă puteți consulta pagina de manual).

7.2.3.3 pthread_setcancelstate și pthread_setcanceltype

În mod implicit, un thread poate fi anulat de alte thread-uri din același proces. Totuși, el își poate modifica starea, apelând `pthread_setcancelstate`. Parametrul `state` poate fi una din valorile:

- `PTHREAD_CANCEL_ENABLE` - thread-ul curent devine anulabil (implicit);
- `PTHREAD_CANCEL_DISABLE` - thread-ul curent nu se va putea anula prin apelul `pthread_cancel`.

Tot în mod implicit, dacă un thread este anulabil, închiderea acestuia se va face doar când se ajunge la un *cancellation point*. Se poate modifica acest tip de anulabilitate, apelând funcția `pthread_setcanceltype`. Parametrul `type` poate lua una din valorile:

- `PTHREAD_CANCEL_DEFERRED` - thread-ul curent se poate închide doar când atinge un punct de anulare (implicit);
- `PTHREAD_CANCEL_ASYNCHRONOUS` - thread-ul curent se poate închide imediat ce se apelează `pthread_cancel`.

7.3 Mersul lucrării

7.3.1 Întrebări recapitulative

1. Cum se compilează în linia de comandă programele care folosesc thread-uri?
2. Ce primește ca argument și ce returnează o funcție de thread?
3. Cum procedăm atunci când funcția de thread trebuie să primească mai multe argumente?

4. Ce este greșit din punct de vedere al logicii programului în codul de mai jos?

```
pthread_t tid;
int i;
for(i=0; i<NR_THREADS; i++) {
    pthread_create(&tid, NULL, thread_fn, NULL);
    pthread_join(tid, NULL);
}
```

5. Când își încheie execuția un thread pentru care s-a apelat `pthread_cancel`?

7.3.2 Probleme rezolvate

1. (*l07p1_thread.c*) Programul este descris în secțiunea 7.2.1.3.

2. (*l07p2_max.c*) Programul este descris în secțiunea 7.2.2.1.

7.3.3 Probleme propuse

1. (35p) Scrieți un program care numără câte cifre de '1' se găsesc în numerele de la 1 la n, folosind mai multe thread-uri. Pentru punctaj maxim, "munca" trebuie împărțită în mod echitabil între thread-uri (țineți cont că numerele mici au mai puține cifre ca numerele mari).

2. (10p) Pentru problema anterioară (dacă ați reușit să echilibrați volumul de calcule între thread-uri), identificați numărul optim de thread-uri pentru a minimiza timpul de execuție (pentru o valoare mare a lui n). Ce relație există între acel număr și numărul de nuclee pe care le are procesorul vostru? Pentru această problemă se va completa un fișier text care conține timpii de rulare pentru diverse valori ale numărului de thread-uri, pentru același n.

3. (25p) Scrieți un program care verifică care este numărul maxim de thread-uri care pot fi create simultan de un program (până ce `pthread_create` eșuează). Pentru ca numărătoarea să fie validă, thread-urile create nu trebuie să se închidă.

4. (30p) Scrieți un program care creează 9 thread-uri, fiecare din acestea executând o buclă infinită în care așteaptă un timp aleator între 1 și 5 secunde, apoi își afișează pe ecran numărul (de la 1 la 9). Pe thread-ul principal se citesc cifre de la tastatură. Atunci când se introduce cifra corespunzătoare unui thread, acesta va fi anulat. După ce s-au închis toate cele 9 thread-uri (trebuie să se și aștepte terminarea lor), se va închide și thread-ul principal.

Laborator 8

Semafoare

8.1 Scopul lucrării

Laboratorul curent prezintă conceptul de semafor, împreună cu cele mai importante apeluri de sistem Linux pentru manipularea semafoarelor.

8.2 Fundamente teoretice

8.2.1 Conceptul de semafor

Semaforul este un mecanism de sincronizare a execuției mai multor procese ce folosesc aceeași resursă. Zona din codul unui program în care resursa comună este utilizată se numește *regiune critică*. De obicei, accesul unui proces la regiunea critică trebuie restricționat, astfel încât resursa comună să se găsească mereu într-o stare consistentă.

Componentele principale ale unui semafor sunt:

- un întreg `perm`, ce reprezintă numărul de permisiuni asociate cu semaforul respectiv;
- o coadă `q`, ce reprezintă lista de procese ce așteaptă să primească permisiuni de la semaforul respectiv.

Cele două componente nu vor fi manipulate direct, ci prin intermediul următoarelor primitive:

- *wait* (denumită și *down* sau P)
  Această primitivă încearcă să decrementeze numărul de permisiuni, dacă acest număr este o valoare pozitivă. Altfel, procesul curent este pus în coada de așteptare și suspendat.

```
while (perm == 0) {
    q.push(crt_proc);
    crt_proc.suspend();
}
perm = perm - 1;
```

- *post* (denumită și *up* sau V)
  Se incrementează numărul de permisiuni, iar primul proces din coadă este trezit pentru a-și continua execuția.

```
perm = perm + 1;
if (!q.empty()) {
    proc = q.pop();
    proc.resume();
}
```

Se consideră că pseudocodul pentru cele două operații de mai sus se execută în mod atomic.

8.2.2 Crearea, deschiderea și închiderea semafoarelor

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);

int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

Pentru a compila programe ce folosesc aceste funcții, `gcc` trebuie să primească în plus opțiunea `-pthread`.

8.2.2.1 sem_open

Deschiderea unui semafor (cu nume) pe Linux se face în mod similar cu deschiderea unui fișier. Dacă mai multe procese apelează `sem_open` pe același nume, vor primi referințe la același semafor.

Parametrul `oflag` reprezintă o mască pe biți ce conține valori privind modul de deschidere:

- `O_CREAT` - în cazul în care semaforul nu există deja, este creat;
- `O_EXCL` - dacă folosim `O_CREAT` și semaforul există deja, apelul de sistem va eșua.

În cazul în care folosim flag-ul `O_CREAT`, va trebui să specificăm încă doi parametri, `mode` și `value`.

Parametrul `mode` reprezintă permisiunile semaforului, care se specifică în mod similar cu cele pentru un fișier la apelul de sistem `open`.

Parametrul `value` reprezintă numărul de permisiuni inițiale ale semaforului.

8.2.2.2 sem_close

Apelul de sistem `sem_close` este utilizat pentru a închide referința spre un semafor. Semaforul va continua să existe în sistem și va putea fi re-deschis ulterior.

8.2.2.3 sem_unlink

Acest apel de sistem va șterge semaforul din sistem. Dacă mai există referințe spre el, semaforul se va șterge doar când ultima referință se închide, dar semaforul nu mai poate fi deschis între timp.

8.2.2.4 Exemplu

Dorim să scriem o funcție care adaugă la finalul unui fișier un mesaj dat (*logging*). Această funcție trebuie să poată fi apelată din programe diferite pentru a scrie în același fișier, fără ca fișierul să devină corupt. În acest caz, resursa comună este fișierul, iar regiunea critică va fi codul ce operează asupra fișierului.

Vom începe prin a deschide și închide un semafor cu nume, care va proteja regiunea critică.

```
1  void write_log(const char *message)
2  {
3      sem_t *logSem = NULL;
4
5      logSem = sem_open("/l08p1_log_semaphore", O_CREAT, 0644, 1);
6      if(logSem == NULL) {
7          perror("Could not aquire the semaphore");
8          return;
9      }
10
11     ...
```

```
12
13      sem_close(logSem);
14  }
```

La linia 3 se declară o variabilă de tip `sem_t*`, care va pointa spre structura de tip semafor.

La linia 5 apelăm `sem_open` pentru a obține acces la semafor. Mai multe programe care vor să intre în aceeași regiune critică trebuie să acceseze același semafor. Din acest motiv, numele semafoarelor sunt vizibile la nivelul întregului sistem. Numele semaforului trebuie să înceapă cu '/' și să nu conțină alt caracter '/'.

În acest punct, nu putem ști dacă suntem primii care accesează semaforul sau acesta există deja. Vom folosi valoarea `O_CREAT` pentru parametrul `oflag`, care are semantica de a crea semaforul doar dacă nu există deja.

În cazul în care semaforul este creat, se vor lua în considerare și ultimii doi parametri, respectiv permisiunile inițiale și valoarea inițială. Vom seta valoarea inițială pe 1, pentru a permite intrarea în regiunea critică a unui singur proces, la un moment dat.

În cazul în care nu se reușește obținerea accesului la semafor, `sem_open` returnează `NULL`, caz în care `perror` ne poate indica motivul eșecului.

La finalul funcției se va apela `sem_close` (linia 13), pentru a închide semaforul.

8.2.3 Operații pe semafoare

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);

int sem_post(sem_t *sem);
```

8.2.3.1 sem_wait

Acest apel de sistem marchează intrarea în regiunea critică și corespunde primitivei *wait* descrisă în secțiunea 8.2.1. La apelul `sem_wait` procesul sau thread-ul curent poate să își continue execuția, obținând permisiunea de la semafor, sau să fie blocat și pus în coada de așteptare până când permisiunea cerută va fi disponibilă.

8.2.3.2 sem_post

Acest apel de sistem marchează ieșirea din regiunea critică și returnează permisiunea obținută prin `sem_wait`. El corespunde primitivei *post* descrisă în secțiunea 8.2.1.

8.2.3.3 sem_trywait

Există situații în care vrem să obținem o permisiune de la un semafor, dar, în cazul în care aceasta nu este disponibilă, să nu fim blocați, ci să putem continua execuția.

Apelul de sistem `sem_trywait` are același efect cu `sem_wait`, dacă permisiunea e disponibilă, altfel nu va bloca procesul sau thread-ul curent, ci va returna o valoare de eroare diferită de 0.

8.2.3.4 Exemplu

Vom continua exemplul din secțiunea 8.2.2.4 pentru sincronizarea scrierii în fișier.

```
1  void write_log(const char *message)
2  {
3      sem_t *logSem = NULL;
4      int fd = -1;
5
6      logSem = sem_open("/l08p1_log_semaphore", O_CREAT, 0644, 1);
7      if(logSem == NULL) {
8          perror("Could not aquire the semaphore");
```

```
9         return;
10    }
11
12    //entering the critical region
13    sem_wait(logSem);
14
15    fd = open("log.txt", O_CREAT | O_WRONLY | O_APPEND, 0644);
16    if(fd < 0) {
17        perror("Could not open log for writing");
18    } else {
19        write(fd, message, strlen(message));
20        close(fd);
21    }
22
23    //exiting the critical region
24    sem_post(logSem);
25
26    sem_close(logSem);
27 }
```

Regiunea critică este delimitată de apelurile `sem_wait` de la linia 13 și `sem_post` de la linia 24. Două thread-uri sau procese diferite care folosesc acest semafor nu pot fi simultan cu execuția între `sem_wait` și `sem_post`.

În interiorul regiunii critice se poate scrie cod normal care accesează resursa. Chiar dacă întâlnim situații de eroare, nu este recomandat să părăsim funcția din interiorul regiunii critice (apelând `return`), deoarece nu am mai ajunge să apelăm `sem_post`, iar semaforul ar rămâne blocat.

8.2.4 Semafoare anonime

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
```

Semafoarele anonime funcționează pe același principiu cu semafoarele cu nume, cu excepția faptului că sunt sunt inițializate în procesul curent și se folosesc pentru a sincroniza thread-urile acestuia (deși se pot transmite și altor procese, dar numai folosind memoria partajată).

8.2.4.1 sem_init

Apelul de sistem primește ca prim parametru un pointer la o variabilă de tip semafor, pe care o va inițializa. Inițializarea unui semafor se face **o singură dată**.

Parametrul `pshared` trebuie să aibă valoarea 0 dacă vrem să folosim semaforul doar în procesul curent sau 1 dacă vrem să poată fi transmis și altor procese.

Ultimul parametru reprezintă valoarea inițială a semaforului, fiind similar cu parametrul cu același nume din `sem_open`.

8.2.4.2 sem_destroy

Acest apel de sistem primește ca argument un pointer la un semafor inițializat cu `sem_init` și eliberează resursele alocate acestuia.

8.2.4.3 Exemplu

Vom prezenta un exemplu similar cu cel de la semafoarele cu nume, dar vom folosi un semafor anonim pentru a sincroniza accesul la fișier pentru mai multe thread-uri din procesul curent. Fiecare thread va scrie câteva mesaje în fișier, așteptând un timp aleator între scrieri.

În primul rând, pentru a putea transmite informații thread-urilor, vom defini structura `TH_STRUCT`, care conține informațiile ce se doresc transmise. În mod particular, vom avea câte o valoare care să apară în mesaj și un pointer la semafor.

```
1  typedef struct {
2      int value;
3      sem_t *logSem;
4  } TH_STRUCT;
```

În funcția `main()` se va inițializa semaforul, se vor crea thread-urile și se va aștepta terminarea acestora.

```
1  int main(void)
2  {
3      sem_t logSem;
4      int i;
5      TH_STRUCT params[NR_THREADS];
6      pthread_t tids[NR_THREADS];
7
8      //initialize random number generator
9      srand(time(NULL));
10     //initialize the semaphore
11     if(sem_init(&logSem, 0, 1) != 0) {
12         perror("Could not init the semaphore");
13         return -1;
14     }
15     //create the threads
16     for(i=0; i<NR_THREADS; i++) {
17         params[i].value = 10 * (i + 1);
18         params[i].logSem = &logSem;
19         pthread_create(&tids[i], NULL, thread_function, &params[i]);
20     }
21     //wait for the threads to finish
22     for(i=0; i<NR_THREADS; i++) {
23         pthread_join(tids[i], NULL);
24     }
25     //destroy the semaphore
26     sem_destroy(&logSem);
27
28     return 0;
29 }
```

La linia 9 se inițializează generatorul de numere aleatoare, pe baza timpului curent.

La linia 11 se inițializează semaforul `logSem`. Observați că acesta a fost declarat la linia 3 ca `sem_t`, nu ca `sem_t*`, ca în cazul semafoarelor cu nume.

Liniile 16–20 conțin bucla de creare a thread-urilor. În primul rând se inițializează structura cu parametrii fiecărui thread (liniile 17–18), apoi se apelează `pthread_create` ce completează id-ul fiecărui thread creat în vectorul `tids`.

Următoarea buclă (liniile 22–24) așteaptă terminarea fiecărui thread. Deoarece nu ne interesează valorile returnate de funcțiile de thread, putem seta parametrul al doilea pe `NULL`.

```
1  void *thread_function(void *p)
2  {
3      TH_STRUCT *s = (TH_STRUCT*)p;
4      int i, count;
5      char message[20];
6
7      for(i=0; i<3; i++) {
8          usleep(1000 * (rand() % 20));
```

```
9           count = s->value + i + 1;
10          snprintf(message, 20, "Log entry %d\n", count);
11          write_log(message, s->logSem);
12      }
13      return NULL;
14  }
```

Funcția thread-urilor definită mai sus începe prin a face cast la pointer-ul primit către un pointer la `TH_STRUCT`, pentru a-și putea accesa parametrii.

În bucla de la liniile 7–12 se apelează scrierea de mesaje în fișier, la intervale aleatorii.

Pentru a aștepta un timp aleatoriu folosim funcția `usleep` care așteaptă numărul de microsecunde primit ca parametru. Valoarea dată va reprezenta un timp aleator între 0 și 19 milisecunde.

Liniile 9–10 construiesc mesajul ce trebuie afișat, folosind funcția `snprintf`, apoi acesta se scrie în log, la linia 11.

```
1   void write_log(const char *message, sem_t *logSem)
2   {
3       int fd = -1;
4       //entering the critical region
5       sem_wait(logSem);
6       fd = open("log2.txt", O_CREAT | O_WRONLY | O_APPEND, 0644);
7       if(fd < 0) {
8           perror("Could not open log for writing");
9       } else {
10          write(fd, message, strlen(message));
11          close(fd);
12      }
13      //exiting the critical region
14      sem_post(logSem);
15  }
```

Funcția `write_log` este similară cu cea din secțiunile anterioare, diferența fiind că nu se mai deschide semaforul, acesta fiind primit ca parametru.

8.3 Mersul lucrării

8.3.1 Întrebări recapitulative

1. Ce se întâmplă când apelăm `sem_wait` pe un semafor cu 0 permisiuni?
2. În ce condiții alegem să folosim un semafor cu nume față de unul anonim?
3. Care este diferența dintre `sem_close` și `sem_unlink`?
4. Pentru a sincroniza mai multe thread-uri se folosește un semafor comun pe care se apelează `sem_wait` la începutul funcției și `sem_post` la final. De ce este în general greșită această abordare?
5. Codul de mai jos se apelează pe mai multe thread-uri pentru a construi un vector global `v` cu numere prime. Explicați ce se întâmplă dacă un thread este preemptat între liniile 3 și 4 și se execută codul unui alt thread? Cum ați folosi un semafor pentru a evita cursa critică?

```
1   for(i=start; i<=end; i++) {
2       if(is_prime(i)) {
3           size++;
4           v[size] = i;
5       }
6   }
```

8.3.2 Probleme rezolvate

1. (*l08p1_log.c*) Să se scrie un program care scrie mai multe mesaje adăugând la finalul unui fișier, accesul la fișier fiind sincronizat printr-un semafor cu nume. Programul este descris în secțiunile 8.2.2.4 și 8.2.3.4.

2. (*l08p2_digits.c*) Să se scrie un program care scrie mesaje într-un fișier pe mai multe thread-uri, sincronizând accesul la fișier printr-un semafor anonim. Observați ordinea mesajelor scrise la execuția programului. Programul este descris în secțiunea 8.2.4.3.

8.3.3 Probleme propuse

1. (25p) Scrieți un program care conține o variabilă globală `long count`, ce are valoarea inițială 0. Creați N thread-uri, fiecare executând funcția de mai jos:

```
1  void *thread_function(void *unused)
2  {
3      int i;
4      long aux;
5
6      for(i=0; i<M; i++){
7          aux = count;
8          aux++;
9          usleep(random() % 10);
10         count = aux;
11     }
12     return NULL;
13 }
```

   Se pot alege valorile $N = 4$ și $M = 1000$.

   (a) Ce valoare ar trebui să aibă variabila `count` la final?

   (b) Afișați valoarea variabilei `count` la finalul programului. De ce diferă de valoarea așteptată?

   (c) Sincronizați thread-urile astfel încât variabila `count` să aibă la final valoarea așteptată.

2. (25p) Scrieți un program care generează 16 thread-uri care execută în mod concurent funcția de mai jos:

```
1  void *limited_area(void *unused)
2  {
3      nrThreads++;
4      usleep(100);
5      printf("The number of threads in the limited area is: %d\n", nrThreads);
6      nrThreads--;
7      return NULL;
8  }
```

   `nrThreads` este o variabilă globală de tip `int`. Sincronizați thread-urile, astfel încât numărul maxim de thread-uri afișat în zona limitată să fie n, unde n se primește ca argument în linia de comandă. Având în vedere soluția de la problema precedentă, sincronizați corect accesul la variabila `nrThreads`!

3. (25p) Scrieți un program care generează N thread-uri sau N procese, fiecare din acestea afișând într-o buclă infinită propriul id, apoi așteaptând un timp aleatoriu. Sincronizați cele N thread-uri sau procese, astfel id-urile să se afișeze alternativ (de exemplu dacă avem 3 procese se va afișa `1 2 3 1 2 3 1 2 ...`).
   Indicație: Pentru a vă asigura că textul afișat cu `printf` ajunge pe ecran la momentul respectiv, după apelul de `printf` apelați `fflush(stdout)`.

4. (25p) Simulați utilizând thread-uri modul în care atomii de *Na* și *Cl* interacționează pentru a forma molecula de sare. Fiecare thread modelează un atom care se creează în mod aleator (fie de tip *Na*, fie de tip *Cl*). Atomul respectiv trebuie să aștepte apariția unui atom de tipul opus, după care ambii formează molecula și dispar (afișează cu care thread s-a format molecula, apoi se încheie thread-ul curent). Atomii de același tip trebuie să fie "consumați" în ordinea în care au ajuns la punctul de formare al moleculei. Pentru a valida implementarea, verificați faptul că dacă un atom de *Na* cu id-ul x afișează că a format molecula cu un atom de *Cl* cu id-ul y, atunci și atomul de *Cl* cu id-ul y trebuie să afișeze că a format molecula cu atomul de *Na* cu id-ul x. Cele două mesaje de la cele două thread-uri pentru atomi ar trebui să fie consecutive.
   Bonus: (7p) Aceeași cerință, pentru molecula H_2O.
   Bonus 2: (8p) Aceeași cerință, pentru molecula H_2SO_4.

Laborator 9

Lacăte și variabile condiționale

9.1 Scopul lucrării

Laboratorul curent prezintă lacătele (eng. *mutex*) și variabilele condiționale, ca mecanisme de sincronizare în Linux.

9.2 Fundamente teoretice

9.2.1 Conceptul de lacăt

Lacătul este unul dintre cele mai simple mecanisme de sincronizare, având doar două stări: blocat (eng. *locked*) și deblocat (eng. *unlocked*). În general se folosește pentru a sincroniza thread-uri din același proces.

Atunci când un thread dorește să intre într-o regiune critică va încerca să blocheze lacătul (operația *lock*). Dacă reușește, va trece mai departe, iar lacătul va fi blocat. Dacă lacătul este deja blocat, thread-ul curent va fi blocat până ce lacătul se deblochează.

La ieșirea din regiunea critică, thread-ul care a blocat lacătul trebuie să îl și deblocheze. Nici un alt thread nu poate debloca lacătul.

9.2.2 Inițializarea și ștergerea unui lacăt

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

9.2.2.1 pthread_mutex_init

Această funcție inițializează mutex-ul primit ca pointer, folosind atributele din parametrul `attr`. Dacă se dorește utilizarea atributelor implicite, parametrul `attr` poate fi `NULL`. În cazul în care inițializarea s-a efectuat cu succes, funcția returnează valoarea 0, iar lacătul se află în starea *deblocat*.

9.2.2.2 PTHREAD_MUTEX_INITIALIZER

În cazul în care dorim să inițializăm un mutex cu atributele implicite, putem utiliza și macro-ul `PTHREAD_MUTEX_INITIALIZER`, asignând-ul variabilei de tip `pthread_mutex_t`.

9.2.2.3 pthread_mutex_destroy

Această funcție se folosește pentru ștergerea și eliberarea resurselor mutex-ului primit ca parametru.

9.2.2.4 Exemplu

Dorim să scriem un program care pornește 4 thread-uri care execută aceeași funcție. În fiecare thread, avem o regiune comună, care se poate executa în paralel, respectiv o regiune critică, care nu are voie să se execute în paralel. În ambele regiuni se va apela `sleep(1)`.

```
1  int main()
2  {
3      int i;
4      pthread_t tids[NR_THREADS];
5      pthread_mutex_t lock;
6
7      if(pthread_mutex_init(&lock, NULL) != 0) {
8          perror("error initializing the mutex");
9          return 1;
10     }
11     for(i=0; i<NR_THREADS; i++) {
12         pthread_create(&tids[i], NULL, thread_function, &lock);
13     }
14     for(i=0; i<NR_THREADS; i++) {
15         pthread_join(tids[i], NULL);
16     }
17     pthread_mutex_destroy(&lock);
18
19     return 0;
20 }
```

În funcția `main()` se declară vectorul `tids` care va reține id-urile thread-urilor create (linia 4), respectiv mutex-ul `lock` (linia 5). La linia 7 se face inițializarea mutex-ului, apelând `pthread_mutex_init()`. Mutex-ul se putea inițializa și la declarație: `pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;`.

Thread-urile vor fi create în bucla de la liniile 11–13, apoi se va aștepta terminarea acestora în bucla de la liniile 14–16. Funcția executată de thread primește ca argument doar o referință spre lacăt.

La final, lacătul este distrus (linia 17).

Funcția thread-ului (fără protecția regiunii critice) este prezentată mai jos.

```
1  void *thread_function(void *arg)
2  {
3      pthread_mutex_t *lock = (pthread_mutex_t*)arg;
4
5      //common region
6      sleep(1);
7
8      //critical region
9      sleep(1);
10
11     return NULL;
12 }
```

Vom executa programul folosind comanda `time`:

```
$ time ./l09p1_mutex

real    0m2,005s
```

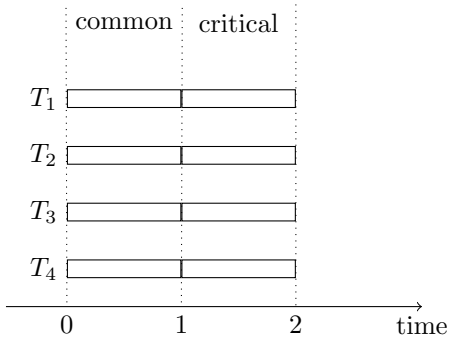


Figura 9.1: Execuția în paralel a thread-urilor fără protejarea regiunii critice

```
user    0m0,001s
sys     0m0,005s
```

Rezultatul comenzii ne indică faptul că cele 4 thread-uri s-au executat în paralel (atât regiunea comună, cât și regiunea critică). Fiecare thread a "dormit" timp de 2 secunde în total, iar cum totul s-a executat în paralel, timpul total de execuție (*real*) a fost de aproximativ 2 secunde, conform Figurii 9.1.

9.2.3 Operații pe lacăte

```c
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

9.2.3.1 pthread_mutex_lock

Apelul acestei funcții realizează blocarea lacătului. Dacă lacătul este deja blocat, thread-ul curent va fi pus în așteptare până la deblocarea acestuia, de către thread-ul care îl ține blocat.

9.2.3.2 pthread_mutex_unlock

Se realizează deblocarea lacătului. Doar thread-ul care a blocat lacătul are voie să îl deblocheze.

9.2.3.3 pthread_mutex_trylock

Se încearcă blocarea lacătului, la fel ca la `pthread_mutex_lock`. În cazul în care lacătul era liber, comportamentul este identic. În cazul în care lacătul era deja blocat, funcția nu blochează thread-ul curent, ci returnează o valoare diferită de zero.

9.2.3.4 Exemplu: protecția regiunii critice

Vom reveni la exemplul anterior, și vom modifica funcția `thread_function`, astfel încât regiunea critică să fie protejată prin lacăt.

```c
1  void *thread_function(void *arg)
2  {
3      pthread_mutex_t *lock = (pthread_mutex_t*)arg;
4
5      //common region
6      sleep(1);
7
8      //critical region
```

```
9       pthread_mutex_lock(lock);
10      sleep(1);
11      pthread_mutex_unlock(lock);
12
13      return NULL;
14  }
```

Pointer-ul spre lacăt se obține făcând cast argumentului primit la tipul `pthread_mutex_t` (linia 3). Regiunea critică începe prin blocarea lacătului, (linia 9) și se termină prin deblocarea acestuia (linia 11).

Vom executa din nou programul folosind comanda `time`:

```
$ time ./l09p1_mutex

real    0m5,004s
user    0m0,001s
sys     0m0,003s
```

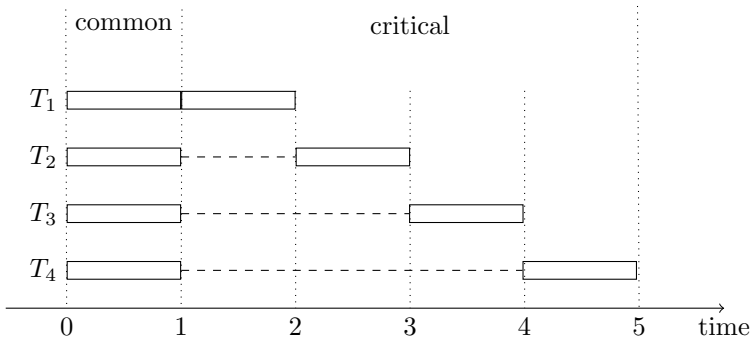


Figura 9.2: Execuția în paralel a thread-urilor cu protejarea regiunii critice

De acestă dată, timpul total a fost de aproximativ 5 secunde. Regiunea comună a fost executată în paralel, timp de o secundă. Regiunea critică a fost executată de fiecare thread, pe rând, timp de o secundă, deci 4 secunde în total. Figura 9.2 ilustrează execuția thread-urilor. Ordinea de execuție a regiunilor critice în cele 4 thread-uri poate să difere de la o rulare la alta.

9.2.4 Conceptul de variabilă condițională

La unele probleme de sincronizare, un thread trebuie să aștepte îndeplinirea unei condiții, pentru a trece mai departe. Variabilele implicate în condiție sunt protejate de un lacăt. Prin urmare, thread-ul trebuie să blocheze lacătul ca să poată accesa variabilele. Un alt thread, care ar putea modifica variabilele implicate în condiție, are și el nevoie de același lacăt. Dacă thread-ul care așteptă îndeplinirea condiției ar ține lacătul blocat, s-ar ajunge la o inter-blocare (eng. *deadlock*).

Variabilele condiționale oferă o soluție elegantă pentru problema de mai sus, prin primitiva de *așteptare* `wait`. Această primitivă poate debloca lacătul pe perioada cât thread-ul curent așteaptă, dând șansa altor thread-uri să modifice variabilele implicate în condiție. Atunci când un alt thread realizează îndeplinirea condiției, poate *semnaliza*, cu primitiva `signal`, variabila condițională, mecanism prin care thread-ul care așteaptă va fi trezit.

9.2.5 Inițializarea și ștergerea unei variabile condiționale

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

9.2.5.1 pthread_cond_init

Această funcție inițializează variabila condițională primită ca pointer, folosind atributele din parametrul `attr`, care poate fi `NULL` dacă dorim atributele implicite.

9.2.5.2 PTHREAD_COND_INITIALIZER

Acest macro se poate folosi pentru inițializarea unei variabile condiționale cu atributele implicite.

9.2.5.3 pthread_cond_destroy

Această funcție eliberează resursele variabilei condiționale primite ca parametru.

9.2.6 Operații pe variabile condiționale

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex,
                      const struct timespec *restrict abstime);

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

9.2.6.1 pthread_cond_wait

Funcția `pthread_cond_wait()` pune thread-ul curent în așteptare, până când se apelează funcția `pthread_cond_signal()` sau `pthread_cond_broadcast()`, cu aceeași variabilă condițională ca parametru al celor două funcții.

Parametrul `mutex` trebuie să fie un pointer spre un lacăt valid, care a fost blocat în prealabil de către thread-ul curent. La intrarea în starea de așteptare a thread-ului curent (în primitiva `wait`), lacătul va fi deblocat, iar la trezirea thread-ului și reluarea execuției lui, dar înainte de revenirea din `wait`, lacătul va fi blocat din nou.

9.2.6.2 pthread_cond_timedwait

Funcția `pthread_cond_timedwait()` are un comportament similar cu `pthread_cond_wait()`, dar mai primește în plus parametrul `abstime`, prin care se specifică un moment de timp în viitor. Dacă până la momentul de timp specificat nu se primește semnalul, thread-ul curent va fi oricum trezit.

9.2.6.3 pthread_cond_signal

Această funcție trezește **primul** thread pus în așteptare de variabila condițională primită ca parametru.

9.2.6.4 pthread_cond_broadcast

Această funcție trezește **toate** thread-urile puse în așteptare de variabila condițională primită ca parametru.

9.2.6.5 Exemplu

O companie are un cont comun, pentru mai mulți angajați, a cărui balanță este reprezentată de variabila globală `balance`. Asupra contului se pot face operațiuni de depunere (*deposit*) și retragere (*withdraw*), fiecare acțiune fiind modelată de un thread separat. Balanța contului nu poate deveni negativă, așa că un thread care vrea să retragă bani trebuie să aștepte ca suma dorită să existe în cont.

Vom avea în total 7 thread-uri:

- thread-urile 0, 2, 3 și 4 vor extrage fiecare câte 7 lei;
- thread-urile 1, 5 și 6 vor depune fiecare câte 11 lei.

Vom folosi un lacăt ca să sincronizăm accesul la variabila `balance`. Thread-urile care nu pot efectua retragerea vor aștepta folosind o variabilă condițională.

```
1  #define NR_THREADS 7
2
3  typedef struct {
4      int id;
5      pthread_mutex_t *lock;
6      pthread_cond_t *cond;
7  } TH_STRUCT;
8
9  int main()
10 {
11     int i;
12     TH_STRUCT params[NR_THREADS];
13     pthread_t tids[NR_THREADS];
14     pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
15     pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
16
17     for(i=0; i<NR_THREADS; i++) {
18         params[i].id = i;
19         params[i].lock = &lock;
20         params[i].cond = &cond;
21         if(i == 0 || i == 2 || i == 3 || i == 4) {
22             pthread_create(&tids[i], NULL, thread_withdraw, &params[i]);
23         } else {
24             pthread_create(&tids[i], NULL, thread_deposit, &params[i]);
25         }
26     }
27
28     for(i=0; i<NR_THREADS; i++) {
29         pthread_join(tids[i], NULL);
30     }
31
32     pthread_mutex_destroy(&lock);
33     pthread_cond_destroy(&cond);
34
35     return 0;
36 }
```

Fiecare thread va primi ca parametru un pointer de tip `TH_STRUCT*` (structură declarată la liniile 3–7), care va conține id-ul curent (un număr între 0 și 6), un pointer la lacăt, respectiv un pointer la variabila condițională. În funcția `main()` se va declara un vector de astfel de structuri (linia 12), câte una pentru fiecare thread, precum și un vector care va reține *thread id*-urile create (linia 13).

Lacătul `lock` și variabila condițională `cond` sunt declarate și inițializate la liniile 14 și, respectiv 15, distrugerea lor făcându-se la finalul programului (liniile 32 și, respectiv 33).

Bucla de la liniile 17–26 completează parametrii fiecărui thread, apoi creează thread-ul, alegând funcția potrivită, în funcție de id (`thread_withdraw` sau `thread_deposit`). Bucla de la liniile 28–30 așteaptă terminarea tuturor thread-urilor.

```
1  #define AMOUNT_WITHDRAW 7
2
3  void *thread_withdraw(void *arg)
4  {
5      TH_STRUCT *s = (TH_STRUCT*)arg;
6      usleep(10000 * s->id);
7
8      pthread_mutex_lock(s->lock);
9      while(balance < AMOUNT_WITHDRAW) {
10         printf("[Th%d] Not enough money (%d). Will wait...\n", s->id, balance);
11         pthread_cond_wait(s->cond, s->lock);
12     }
13     balance -= AMOUNT_WITHDRAW;
14     printf("[Th%d] Withdrawn %d. Current balance is %d.\n",
15            s->id, AMOUNT_WITHDRAW, balance);
16     pthread_mutex_unlock(s->lock);
17
18     return NULL;
19 }
```

Funcția `thread_withdraw()` protejează accesul la variabila `balance` folosind lacătul `lock`, care se blochează la linia 8 și se deblochează la linia 16.

Cât timp balanța este mai mică decât suma dorită (linia 9), thread-ul curent trebuie să aștepte ca alte thread-uri să mai depună bani (linia 11). Nu putem aștepta folosind un alt lacăt sau un semafor, deoarece suntem în interiorul regiunii critice, am blocat lacătul `lock`, iar în aceste condiții, nici un alt thread nu ar putea ajunge să modifice balanța. Variabila condițională, în schimb, realizează deblocarea lacătului pe perioada în care thread-ul curent e blocat, permițând altor thread-uri să modifice variabila `balance`.

```
1  #define AMOUNT_DEPOSIT 11
2
3  void *thread_deposit(void *arg)
4  {
5      TH_STRUCT *s = (TH_STRUCT*)arg;
6      usleep(10000 * s->id);
7
8      pthread_mutex_lock(s->lock);
9      balance += AMOUNT_DEPOSIT;
10     printf("[Th%d] Deposited %d. Current balance is %d.\n",
11            s->id, AMOUNT_DEPOSIT, balance);
12     if(balance >= 2 * AMOUNT_WITHDRAW) {
13         pthread_cond_broadcast(s->cond);
14     } else if (balance >= AMOUNT_WITHDRAW) {
15         pthread_cond_signal(s->cond);
16     }
17     pthread_mutex_unlock(s->lock);
18
19     return NULL;
20 }
```

Funcția `thread_deposit()` are un comportament similar cu `thread_withdraw()`, utilizând același lacăt (liniile 8 și, respectiv 17) pentru a sincroniza accesul la balanță. Depunerea de bani

se poate face oricând, deci acest thread nu trebuie să aștepte. Dacă noua balanță ar permite retrageri, eventualele thread-uri care așteaptă să facă acest lucru ar trebui anunțate. În caz că suma este suficientă pentru mai multe retrageri (linia 12), sunt trezite toate thread-urile, prin *broadcast* (linia 13). Dacă suma ajunge doar pentru o singură retragere (linia 14), atunci un singur thread este semnalizat (linia 15).

Faptul că fiecare thread așteaptă la început un timp direct proporțional cu id-ul acestuia (la apelul funcției `usleep()`) crește șansele ca intrarea în regiunea critică să se facă în ordinea id-urilor. Totuși, acest lucru nu este garantat. În a doua problemă propusă, se cere implementarea unui mecanism care să garanteze ordinea intrării thread-urilor în regiunea critică.

Rezultatul rulării programului se vede mai jos:

```
[Th0] Not enough money (0). Will wait...
[Th1] Deposited 11. Current balance is 11.
[Th0] Withdrawn 7. Current balance is 4.
[Th2] Not enough money (4). Will wait...
[Th3] Not enough money (4). Will wait...
[Th4] Not enough money (4). Will wait...
[Th5] Deposited 11. Current balance is 15.
[Th2] Withdrawn 7. Current balance is 8.
[Th4] Withdrawn 7. Current balance is 1.
[Th3] Not enough money (1). Will wait...
[Th6] Deposited 11. Current balance is 12.
[Th3] Withdrawn 7. Current balance is 5.
```

- Thread-ul 0 dorește să retragă 7 lei, dar balanța este 0, așa că este pus în așteptare.
- Thread-ul 1 depune 11 lei, apoi semnalizează thread-ului 1, care se deblochează și face retragerea.
- Thread-urile 2, 3 și 4, încearcă pe rând să retragă 7 lei, dar balanța este 4, așa că cele 3 thread-uri vor fi blocate.
- Thread-ul 5 depune 11 lei, ducând balanța la 15. Deoarece din această sumă se pot face mai multe retrageri, se va face broadcast, thread-urile 2, 3 și 4 fiind trezite. Primele două dintre acestea (în acest caz 2 și 4) reușesc să retragă bani, ultimul (3) fiind din nou pus în așteptare.
- Thread-ul 6 depune și el 11 lei, apoi trezește thread-ul 3 care face retragerea.

9.3 Mersul lucrării

9.3.1 Întrebări recapitulative

1. Care este diferența dintre un lacăt și un semafor?
2. Ce opțiuni avem pentru a inițializa un lacăt? Dar pentru o variabilă condițională?
3. De ce funcția `pthread_cond_wait` primește și un lacăt (mutex) ca parametru? Ce va face cu acel lacăt?
4. Care este diferența dintre *signal* și *broadcast* pentru o variabilă condițională?
5. De ce pattern-ul comun de utilizare a unei variabile condiționale presupune verificarea condiției cu `while`, nu cu `if`?

9.3.2 Probleme rezolvate

1. (*l09p1_mutex.c*) Să se scrie un program care pornește mai multe thread-uri ce execută aceeași funcție. Fiecare thread va avea o regiune comună, care se poate executa în paralel, respectiv o regiune critică, care nu se poate executa. Programul este descris în secțiunile 9.2.2.4 și 9.2.3.4.

2. (*l09p1_cond_var.c*) Să se scrie un program care modelează operațiile de depunere și retragere asupra unui cont, prin thread-uri. La operația de retragere, în cazul în care fondurile sunt insuficiente, trebuie să se aștepte completarea acestora printr-o operație de depunere. Programul este descris în secțiunea 9.2.6.5.

9.3.3 Probleme propuse

1. (25p) Se dă un vector `v` de numere întregi, împreună cu dimensiunea acestuia `n`, ca variabile globale. Se dorește construcția unui alt vector `nrDiv` (tot variabilă globală), care să conțină numărul de divizori pentru fiecare element din vector. Pentru asta, vom porni un număr de thread-uri (primit ca argument în linia de comandă), fiecare thread procesând într-o buclă elemente din vectorul `v`, până când se ajunge la finalul acestuia. Thread-urile trebuie sincronizate, astfel încât un element să nu fie procesat de mai multe ori.

2. (25p) Modificați exemplul din secțiunea 9.2.6.5, astfel încât ordinea în care thread-urile intră în regiunea critică să fie garantat cea a id-urilor. Pentru aceasta, se va folosi o variabilă globală numită `turn`, care va indica al cui thread e rândul. Fiecare thread va aștepta, utilizând o variabilă condițională, ca variabila `turn` să devină egală cu id-ul lui.

3. (25p) Implementați un program în care mai multe thread-uri citesc câte un octet dintr-un fișier. Fiecare thread va deschide fișierul apelând funcția `open_once()`. La primul apel al funcției se va realiza deschiderea fișierului, urmând ca la următoarele apeluri să se returneze descriptorul de fișier obținut anterior. Deschiderea fișierului trebuie sincronizată, astfel încât, chiar dacă două thread-uri apelează `open_once()` deodată, să se realizeze o singură deschidere a fișierului.
   Bonus: (5p) Funcția `open_once()` va primi ca parametru numele fișierului și se va asigura că fiecare fișier primit se deschide o singură dată.

4. (25p) Implementați un protocol pentru traversarea unui pod cu o singură bandă pentru mașini. În orice moment, mașinile pot traversa podul într-o singură direcție. Cel mult N mașini se pot afla simultan pe pod. Fiecare mașină va fi implementată printr-un thread care execută codul de mai jos:

```
1 void *car_thread(void *direction)
2 {
3     int dir = (int)direction;
4     enter_bridge(dir);
5     cross_bridge(dir);
6     exit_bridge(dir);
7     return NULL;
8 }
```

Laborator 10

Probleme de sincronizare

10.1 Scopul lucrării

Laboratorul curent este unul recapitulativ, în care se folosesc conceptele acumulate în laboratoarele anterioare pentru a rezolva unele probleme de sincronizare mai complexe.

10.2 Mersul lucrării

10.2.1 Problema bărbierului

Se va simula salonul unui bărbier, în care mai mulți clienți sosesc pe rând și sunt serviți, respectându-se următoarele cerințe:

- salonul are `NR_CHAIRS` scaune în sala de așteptare, plus scaunul bărbierului, pe care se așază clientul care e servit;
- clienții sosesc în salon la momente de timp aleatorii, iar dacă
  - sunt scaune disponibile în sala de așteptare, clientul așteaptă să îi vină rândul;
  - toate scaunele din sala de așteptare sunt ocupate, clientul decide că e prea aglomerat și pleacă;
- atunci când scaunul bărbierului e liber, clientul care așteaptă de cel mai mult timp se așază pe el și e servit de către bărbier;
- atunci când nu are clienți, bărbierul doarme și este trezit de clienți atunci când aceștia trebuie serviți;
- atât bărbierul cât și cei `NR_CUSTOMERS` clienți vor fi simulați printr-un thread separat.

Se vor implementa două soluții: prima dintre ele, în fișierul *l10p1_barber_sem.c* se bazează pe semafoare, în timp ce a doua soluție, din fișierul *l10p2_barber_mutex_cond.c* se bazează pe lacăte și variabile condiționale.

10.2.1.1 Soluția 1: semafoare

```
1 #define NR_CUSTOMERS 20
2 #define NR_CHAIRS 3
3
4 sem_t lock;
5 sem_t customer;
6 sem_t barber;
7 int waitingCustomers = 0;
8 int barberChair = 0;
```

Vom începe prin a defini constantele problemei, respectiv numărul de clienți care vin la salon de-a lungul zilei `NR_CUSTOMERS`, respectiv numărul de scaune disponibile în sala de așteptare `NR_CHAIRS`.

Numărul de clienți care se află la un moment dat în sala de așteptare se va reprezenta prin variabila globală `waitingCustomers`. Accesul la această variabilă va fi sincronizat prin semaforul `lock`.

Clienții așteaptă să fie serviți prin semaforul `customer`, iar bărbierul așteaptă clienții prin semaforul `barber`. Variabila `barberChair` indică id-ul clientului care se află la un moment dat pe scaunul bărbierului.

```
1  int main()
2  {
3      int i;
4      pthread_t tidC[NR_CUSTOMERS], tidB;
5
6      srand(time(NULL));
7
8      sem_init(&lock, 0, 1);
9      sem_init(&customer, 0, 0);
10     sem_init(&barber, 0, 0);
11
12     pthread_create(&tidB, NULL, th_barber, NULL);
13     for(i=0; i< NR_CUSTOMERS; i++) {
14         pthread_create(&tidC[i], NULL, th_customer, (void*)(ssize_t)(i+1));
15     }
16     for(i=0; i< NR_CUSTOMERS; i++) {
17         pthread_join(tidC[i], NULL);
18     }
19     pthread_cancel(tidB);
20     pthread_join(tidB, NULL);
21
22     sem_destroy(&lock);
23     sem_destroy(&customer);
24     sem_destroy(&barber);
25
26     return 0;
27 }
```

În funcția `main()` vom defini variabile pentru thread id-urile clienților și pentru thread id-ul bărbierului (linia 4). La linia 6 se inițializează generatorul de numere aleatoare, pe baza timpului curent. Cele trei semafoare utilizate se vor inițializa la începutul programului (liniile 8–10) și se vor distruge la final (liniile 22–24).

La linia 12 se creează thread-ul bărbierului iar la liniile 13–15 se creează thread-urile clienților.

La liniile 16–18 se așteaptă terminarea thread-urilor clienților. Thread-ul bărbierului va executa o buclă infinită (codul lui va fi detaliat mai jos), așa că trebuie oprit explicit după terminarea thread-urilor clienților, apelând `pthread_cancel()` (linia 19). Funcția `pthread_cancel()` nu oprește thread-ul imediat, doar îi transmite semnalul de oprire. La linia 20 se așteaptă oprirea acestuia.

Fiecare thread client trebuie să primească ca argument propriul id (un număr cuprins între 1 și `NR_CUSTOMERS`). Cum variabila `i` iterează între `0` și `NR_CUSTOMERS - 1`, vom transmite valoarea `i + 1`. Funcția `pthread_create()` se așteaptă ca argumentul transmis să fie de tipul `void*`, așa că trebuie să facem cast la acel tip. Dacă am fi făcut cast-ul direct, de la `int` la `void*` am fi primit următorul warning, pe un sistem pe 64 bit: `cast to pointer from integer of different size`. Acest warning ne spune că tipul `int` este pe 32 bit, în tip ce un pointer e pe 64 bit. Din acest motiv, se vor face două cast-uri: întâi de la `int` la `ssize_t` (care este un tip de date numeric, de aceeași dimensiune cu cea a unui pointer), apoi de la `ssize_t` la `void*`.

În continuare, vom folosi un mecanism similar cu cel din problema *rendezvous*.

```
1  void *th_barber(void *arg)
2  {
3      for(;;) {
4          sem_post(&customer);
5          sem_wait(&barber);
6          printf("[B ] Serving customer %d\n", barberChair);
7          usleep(1000);
8      }
9      return NULL;
10 }
```

Thread-ul bărbierului va executa o buclă infinită, în care va aștepta venirea clienților și îi va servi. Acest thread nu se va termina niciodată singur, dar va fi oprit de thread-ul principal, prin apelul funcției `pthread_cancel()`.

La linia 4, se dă o permisiune semaforului `customer`, indicând faptul că scaunul bărbierului e liber și se așteaptă clienți. La linia 5, se așteaptă după semaforul `barber`. Acest semafor va primi permisiuni de la clienți, atunci când aceștia se așază pe scaunul bărbierului. Atunci când ajungem la liniile 6–7 înseamnă că semaforul `barber` a primit o permisiune, deci avem un client de servit.

```
1  void *th_customer(void *arg)
2  {
3      int myId = (int)(ssize_t)arg;
4      int tooBusy = 0;
5
6      usleep(1000 * (rand() % 20));
7      printf("[C%02d] Entering the barber shop\n", myId);
8
9      sem_wait(&lock);
10     if(waitingCustomers < NR_CHAIRS) {
11         ++waitingCustomers;
12         printf("[C%02d] %d customer(s) waiting\n", myId, waitingCustomers);
13     } else {
14         tooBusy = 1;
15         printf("[C%02d] Too busy, will come back another day.\n", myId);
16     }
17     sem_post(&lock);
18
19     if(!tooBusy) {
20         sem_wait(&customer);
21
22         sem_wait(&lock);
23         --waitingCustomers;
24         sem_post(&lock);
25
26         barberChair = myId;
27         sem_post(&barber);
28         printf("[C%02d] being served\n", myId);
29         usleep(1000);
30     }
31
32     return NULL;
33 }
```

Thread-urile clienților vor începe prin obținerea id-ului propriu, făcând cast argumentului `arg` (linia 3).

Fiecare client va aștepta un timp aleatoriu (linia 6), apoi își va anunța sosirea în salon (linia 7).

Mai departe, trebuie verificată variabila `waitingCustomers`, pentru a decide dacă mai este loc în sala de așteptare. Accesul la această variabilă va fi protejat de semaforul `lock` (liniile 9–17).

Dacă numărul de clienți din sala de așteptare este mai mic decât `NR_CHAIRS`, se ia loc în sala de așteptare, incrementând variabila `waitingCustomers` (linia 11) și se afișează câți clienți sunt în așteptare (linia 12). Altfel, se setează variabila locală `tooBusy` și se afișează că salonul e prea aglomerat (linia 15). Observați că variabila `tooBusy` este o variabilă locală, fiecare thread având propria "versiune" a acelei variabile. Prin urmare, nu există acces concurent la acea variabilă între thread-uri, motiv pentru care valoarea ei poate fi testată și după eliberarea semaforului `lock`.

După ce am ieșit din regiunea critică, se verifică variabila `tooBusy`. Dacă acesta nu a fost setată (a rămas 0) așteptăm să se elibereze scaunul bărbierului, așteptând după semaforul `customer` (linia 20). După ce trecem de acest semafor decrementăm variabila `waitingCustomers`, protejând accesul la variabilă cu semaforul `lock` (liniile 22–24). Decrementarea acestei variabile indică faptul că s-a părăsit sala de așteptare și urmează să fim serviți de bărbier. În acest punct avem acces exclusiv la variabila `barberChair`, pe care o setăm la id-ul thread-ului curent, apoi dăm o permisiune semaforului `barber` (linia 27), care va trezi bărbierul, iar clientul curent va fi servit.

10.2.1.2 Soluția 2: lacăte și variabile condiționale

Vom prezenta o soluție similară cu cea precedentă, dar vom utiliza lacăte și variabile condiționale în loc de semafoare.

```
1  #define NR_CUSTOMERS 20
2  #define NR_CHAIRS 3
3
4  pthread_mutex_t lock;
5  pthread_cond_t barber;
6  pthread_cond_t customer;
7  int waitingCustomers = 0;
8  int barberChair = 0;
```

Lacătul `lock` va proteja regiunile critice, în timp ce variabilele condiționale `barber` și `customer` vor fi folosite pentru a pune thread-urile în așteptare. Variabila `barberChair` indică id-ul clientului care se află la un moment dat în scaunul bărbierului, sau 0 dacă scaunul e liber.

```
1  int main()
2  {
3      int i;
4      pthread_t tidC[NR_CUSTOMERS], tidB;
5
6      srand(time(NULL));
7
8      pthread_mutex_init(&lock, NULL);
9      pthread_cond_init(&customer, NULL);
10     pthread_cond_init(&barber, NULL);
11
12     pthread_create(&tidB, NULL, th_barber, NULL);
13     for(i=0; i< NR_CUSTOMERS; i++) {
14         pthread_create(&tidC[i], NULL, th_customer, (void*)(size_t)(i+1));
15     }
16     for(i=0; i< NR_CUSTOMERS; i++) {
17         pthread_join(tidC[i], NULL);
18     }
19     pthread_cancel(tidB);
20     pthread_join(tidB, NULL);
21
22     pthread_mutex_destroy(&lock);
23     pthread_cond_destroy(&customer);
24     pthread_cond_destroy(&barber);
25
26     return 0;
```

```
27 }
```

Funcția `main()` este similară cu cea din soluția anterioară, dar în loc de semafoare inițializăm (liniile 8–10) și distrugem (liniile 22–24) lacătul și variabilele condiționale declarate anterior.

```
1  void *th_barber(void *arg)
2  {
3      int crtCustomer = 0;
4      for(;;) {
5          pthread_mutex_lock(&lock);
6          while(barberChair == 0){
7              pthread_cond_wait(&barber, &lock);
8          }
9          crtCustomer = barberChair;
10         pthread_mutex_unlock(&lock);
11
12         printf("[B ] Serving customer %d\n", crtCustomer);
13         usleep(1000);
14
15         pthread_mutex_lock(&lock);
16         barberChair = 0;
17         pthread_cond_signal(&customer);
18         pthread_mutex_unlock(&lock);
19     }
20     return NULL;
21 }
```

Bărbierul va executa o buclă infinită, la fel ca la soluția anterioară.

Vom proteja variabila `barberChair` prin lacătul `lock`. Cât timp aceasta are valoarea 0 (nu avem clienți pe scaun), bărbierul trebuie să doarmă (liniile 6–8).

După ce avem un client care s-a așezat pe scaun, îi reținem id-ul (linia 9), apoi ieșim din regiunea critică (linia 10) și îl servim (liniile 12–13).

După servirea clientului, setăm variabila `barberChair` pe 0 (s-a eliberat scaunul), apoi semnalizăm eventualului client în așteptare.

```
1  void *th_customer(void *arg)
2  {
3      int myId = (int)(size_t)arg;
4      int tooBusy = 0;
5
6      usleep(1000 * (rand() % 20));
7      printf("[C%02d] Entering the barber shop\n", myId);
8
9      pthread_mutex_lock(&lock);
10     if(waitingCustomers < NR_CHAIRS) {
11         ++waitingCustomers;
12         printf("[C%02d] %d customer(s) waiting\n", myId, waitingCustomers);
13         while(barberChair != 0){
14             pthread_cond_wait(&customer, &lock);
15         }
16         --waitingCustomers;
17         barberChair = myId;
18         pthread_cond_signal(&barber);
19     } else {
20         tooBusy = 1;
21         printf("[C%02d] Too busy, will come back another day.\n", myId);
22     }
23     pthread_mutex_unlock(&lock);
```

```
24
25      if(!tooBusy) {
26          printf("[C%02d] being served\n", myId);
27          usleep(1000);
28      }
29
30      return NULL;
31  }
```

La fel ca la soluția anterioară, clienții își obțin propriul id (linia 3), așteaptă un timp random (linia 6), apoi verifică numărul de locuri ocupate din sala de așteptare (linia 10).

Dacă au fost suficiente locuri în sala de așteptare, incrementăm variabila `waitingCustomers` și afișăm valoarea acesteia. În acest punct clientul a ajuns în sala de așteptare și trebuie să aștepte eliberarea scaunului bărbierului (valoarea variabilei `barberChair` trebuie să devină 0). După ce a trecut de bucla de la liniile 13–15, clientul se poate așeza pe scaunul bărbierului. Se va elibera scaunul din sala de așteptare (linia 16), se va ocupa scaunul bărbierului (linia 17), apoi acesta va fi trezit (linia 18).

10.2.2 Probleme propuse

1. (25p) În soluțiile prezentate în secțiunile 10.2.1.1 și 10.2.1.2 bărbierul execută o buclă infinită și este oprit de threadul principal prin funcția `pthread_cancel()`. Modificați exemplele date, astfel încât thread-ul bărbierului să se închidă singur, după ce toți cei `NR_CUSTOMER` clienți și-au executat thread-ul.

2. (25p) Servirea propriu-zisă a clienților începe sincronizat (thread-ul bărbierului și thread-ul clientului se sincronizează la intrarea în această etapă), dar clientul ar putea de exemplu să "părăsească scaunul" înainte ca bărbierul să termine. Modificați unul din cele două exemple date, astfel încât clientul servit să nu își încheie thread-ul până nu este anunțat de bărbier că a terminat.

3. (50p) **Problema mesei de restaurant:** La o masă de restaurant au loc 5 persoane. Dacă o persoană ajunge la restaurant și sunt locuri libere la masă, se poate așeza imediat. Dacă, în schimb, întreaga masă e ocupată, înseamnă că persoanele de la masă formează un grup și trebuie să aștepte ca toate persoanele să plece de la masă, ca să se poată așeza. Fiecare persoană stă la masă un timp random.

4. (50p) **Traversarea râului:** Pe marginea Someșului se află bărci care au o capacitate de 4 persoane, și care pot fi folosite doar atunci când sunt pline. Avem două tipuri de persoane, care se vor simula prin thread-uri: suporteri "U" și suporteri "CFR". Din motive de siguranță, nu este permis ca într-o barcă să se găsească un suporter al unei echipe împreună cu 3 suporteri ai celeilalte echipe (deci fie avem 2-2, fie 4-0). Nu se va începe îmbarcarea într-o barcă, până ce barca anterioară nu s-a umplut.

5. (60p) **Problema fumătorilor:** Pentru ca o persoană să fumeze are nevoie de 3 ingrediente: tutun, hârtie și brichetă. Există 3 fumători, fiecare dintre ei având o cantitate nelimitată din unul din ingrediente, și un "dealer" (furnizor), care la momente random de timp oferă câte două ingrediente random. De fiecare dată când dealer-ul oferă cele două ingrediente, fumătorul care are ingredientul complementar trebuie să le ia. Cele 3 ingrediente vor fi simulate prin 3 semafoare, iar dealer-ul și cei 3 fumători prin câte un thread. Dealer-ul trebuie doar să acorde permisiuni la cele două semafoare alese random, nu are voie să execute alte acțiuni.

Laborator 11

Comunicarea prin pipe-uri

11.1 Scopul lucrării

Laboratorul curent prezintă principiile comunicări între procese folosind pipe-uri. Se vor studia apelurile de sistem Linux pentru a lucra atât cu pipe-uri cu nume cât și cu pipe-uri anonime.

11.2 Fundamente teoretice

11.2.1 Principiile comunicări prin pipe-uri

Un *pipe* este un tip special de fișier, utilizat pentru comunicarea între procese. Operațiile uzuale pe fișiere (`read` și `write`) vor funcționa și în cazul pipe-urilor.

Pipe-urile funcționează după principiul FIFO (*First In, First Out*). Conceptual, putem să ne imaginăm un pipe similar cu o coadă, în care citirea se face la unul din capete, iar scrierea la celălalt. Principiul FIFO este strict, operația `lseek` pentru deplasarea cursorului în fișier nefiind permisă.

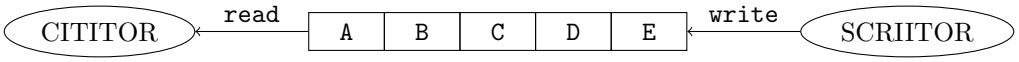


Figura 11.1: Ilustrarea principiului FIFO pentru un pipe

Figura 11.1 reprezintă un pipe în care s-au scris octeții `A`, `B`, `C`, `D`, `E`. La operația de scriere, se va scrie în dreapta lui `E`. Citirea se va face din partea stângă, începând cu `A`.

În condiții normale de utilizare, la fiecare din capetele unui pipe este conectat un proces (unul citește, celălalt scrie). Sistemul de operare permite aceluiași proces să facă ambele operații, dar acest caz se întâlnește mai rar în practică.

Dacă ambele capete sunt conectate, în momentul în care pipe-ul este gol, apelul de sistem `read` nu va returna 0, ca și în cazul unui fișier obișnuit, ci va bloca procesul apelant până ce datele cerute sunt scrise. La fel, dacă pipe-ul este plin (există o capacitate maximă) și sunt ambele capete conectate, apelul de sistem `write` va bloca procesul apelant până ce în pipe va fi suficient spațiu liber pentru scrierea datelor.

Dacă se încearcă citirea dintr-un pipe gol, dar nu avem nici un scriitor conectat, cititorul nu va mai fi blocat, ci operația `read` va returna 0.

Dacă se încearcă scrierea într-un pipe la care nu este conectat nici un cititor, procesul care apelează `write` va primi semnalul `SIGPIPE`, care în mod normal închide procesul respectiv (se poate preveni închiderea recepționând explicit semnalul `SIGPIPE`).

La deschiderea unui pipe cu nume folosind apelul de sistem `open`, fie în citire, fie în scriere, procesul curent este blocat până ce un alt proces se va atașa la celălalt capăt al pipe-ului, apelând `open` cu operația opusă. Cu alte cuvinte, operația `open` funcționează ca o barieră, ce nu permite proceselor cititor și scriitor să continue decât împreună.

11.2.2 Pipe-uri cu nume

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

Apelul de sistem `mkfifo` este utilizat pentru a crea un pipe cu nume, primind ca prim argument calea spre acesta, respectiv permisiunile de acces, ca al doilea argument. În caz de succes se returnează 0.

11.2.2.1 Exemplu de utilizare a pipe-urilor cu nume

Vom scrie două programe, `writer` și `reader`. Primul va crea un pipe cu nume, îl va deschide și va scrie un întreg, apoi îl va închide și îl va șterge. Procesul `reader` va deschide pipe-ul și va citi întregul scris de `writer`.

writer.c:

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/stat.h>
4  #include <fcntl.h>
5  #include <unistd.h>
6
7  #define FIFO_NAME "l11_my_fifo"
8
9  int main(void)
10 {
11     int fd = -1;
12     int x = 42;
13
14     //create fifo
15     if(mkfifo(FIFO_NAME, 0600) != 0){
16         perror("Err creating FIFO");
17         return 1;
18     }
19
20     //open, write and close fifo
21     fd = open(FIFO_NAME, O_WRONLY);
22     if(fd == -1) {
23         perror("Could not open FIFO");
24         return 1;
25     }
26     printf("Writing value %d\n", x);
27     write(fd, &x, sizeof(x));
28     close(fd);
29
30     //delete fifo
31     unlink(FIFO_NAME);
32
33     return 0;
34 }
```

reader.c:

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/stat.h>
4  #include <fcntl.h>
5  #include <unistd.h>
6
7  #define FIFO_NAME "l11_my_fifo"
8
9  int main(void)
10 {
11     int fd = -1;
12     int x = 0;
13
14     //open, read and close fifo
15     fd = open(FIFO_NAME, O_RDONLY);
16     if(fd == -1) {
17         perror("Could not open FIFO");
18         return 1;
19     }
20     read(fd, &x, sizeof(x));
21     printf("The read value is: %d\n", x);
22     close(fd);
23
24     return 0;
25 }
```

Programul *writer* începe prin a crea pipe-ul, apelând `mkfifo` la linia 15. În caz de reușită se continuă deschizând pipe-ul pentru scriere (linia 21), ca și cum ar fi un fișier obișnuit. După deschidere, se scrie în pipe folosind apelul de sistem `write` (linia 27), tot similar cu operația pe un fișier. Pipe-ul este închis la linia 28, apoi șters la linia 31 folosind apelul de sistem `unlink`.

Nu trebuie să ne facem griji dacă pipe-ul mai este utilizat încă de către *reader*, pentru că dacă la ștergere un fișier este deschis de anumite procese, deși numele lui dispare din arborele de fișiere, inode-ul și datele lui sunt păstrate până când fișierul este închis de către toate procesele. Singura problemă ar fi dacă pipe-ul ar fi șters de *writer* înainte de a fi de deschis de *reader*,lucru care, însă, nu se poate întâmpla din motive explicate puțin mai jos.

Programul *reader* nu mai creează sau șterge pipe-ul, deoarece aceste operații sunt efectuate de *writer*. Pipe-ul este deschis în citire la linia 15, se citește întregul la linia 20, apoi se închide la linia 22.

După ce compilăm cele două programe, ar trebui executat întâi *writer*. Acesta creează pipe-ul, apoi se blochează la apelul `open` de la linia 21. Sistemul de operare nu ne permite să trecem mai departe, până nu se deschide și celălalt capăt al pipe-ului, în citire. Continuăm executând și programul *reader* într-un alt terminal. Atunci când se apelează `open` (linia 15), pipe-ul va avea ambele capete deschise și ambele programe pot continua. Din acest moment programele continuă într-o ordine arbitrară, fie *writer* ajunge primul la apelul `write`, fie *reader* ajunge primul la `read`. În al doilea caz, programul *reader* va fi blocat până ce datele sunt scrise în pipe de către *writer*.

11.2.3 Pipe-uri anonime

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

Pipe-urile anonime nu au un nume după care să fie identificate, acestea fiind accesibile doar prin descriptorii de fișier obținuți prin apelul de sistem `pipe`. Acesta primește ca argument adresa unui vector cu două elemente de tip `int`, în care va completa doi descriptori de fișier: `pipefd[0]` va fi folosit pentru citire, în timp ce `pipefd[1]` va fi folosit pentru scriere.

Pipe-urile anonime nu se pot utiliza decât pentru comunicarea dintre un părinte și un copil. Părintele va crea pipe-ul și va obține cei doi descriptori de fișier (pentru citire și scriere). Ulterior, va efectua apelul de sistem `fork` pentru a da naștere procesului copil. Copilul va moșteni structurile de date ale părintelui, inclusiv descriptorii de fișiere și astfel va putea comunica prin pipe-ul creat.

11.2.3.1 Exemplu de utilizare a pipe-urilor anonime

Vom relua exemplul din secțiunea 11.2.2.1, în care două procese comunică printr-un pipe, de data asta anonim, primul proces transmițând o valoare întreagă celui de-al doilea.

Pentru a comunica printr-un pipe anonim, cele două procese vor trebui să fie în relația părinte-copil.

```
1  int main(void)
2  {
3      int fd[2];
4      int x = 0;
5
6      if(pipe(fd) != 0) {
7          perror("Could not create pipe");
8          return 1;
9      }
10
11     if(fork() != 0) {
12         //parent
13         close(fd[0]); //no use for read end
14         x = 42;
15         write(fd[1], &x, sizeof(x));
16         printf("Parent: wrote %d to pipe\n", x);
17         close(fd[1]);
18         wait(NULL);
```

```
19      } else {
20          //child
21          close(fd[1]); //no use for write end
22          read(fd[0], &x, sizeof(x));
23          printf("Child: read %d from pipe\n", x);
24          close(fd[0]);
25      }
26
27      return 0;
28  }
```

Programul începe prin construcția pipe-ului. Va trebui să declarăm un vector de doi întregi, care vor stoca descriptorii de fișier pentru cele două capete ale pipe-ului (linia 3). Procesul părinte inițializează pipe-ul prin apelul de sistem `pipe` la linia 6. În acest moment, descriptorii de fișier sunt scriși în vectorul `fd`.

La linia 11, efectuăm apelul de sistem `fork`, ce dă naștere procesului copil. Dacă `fork` returnează o valoare nenulă suntem în procesul părinte, altfel în copil.

Ambele procese continuă prin a închide capătul nefolosit al pipe-ului. Procesul părinte va scrie în pipe, deci nu va avea nevoie de capătul de citire și va închide `fd[0]` (linia 13). Analog, procesul copil doar citește, deci poate închide capătul de scriere `fd[1]` (linia 21).

Pentru a scrie valoarea întreagă în pipe, procesul părinte va folosi apelul de sistem `write` (linia 15), apoi va închide capătul de scriere al pipe-ului (linia 17) și va aștepta terminarea procesului copil (linia 18).

Similar, procesul copil va citi întregul din pipe (lina 22), apoi va închide capătul de citire (linia 24).

11.2.4 Comunicarea bidirecțională

În exemplele anterioare, am observat o comunicare unidirecțională între două procese. În exemplul din secțiunea 11.2.2.1, procesul `writer` transmitea un întreg spre procesul `reader`, în timp ce în exemplul din secțiunea 11.2.3.1 părintele transmitea întregul spre copil. În multe situații din lumea reală este necesar să se comunice în ambele sensuri, un proces putând atât să scrie cât și să citească mesaje.

Deși este posibil să se comunice bidirecțional folosind un singur pipe, sincronizând explicit operațiile de scriere și citire (folosind semafoare), este mult mai simplu să comunicăm bidirecțional folosind două pipe-uri. Dacă folosim un singur pipe pentru comunicare bidirecțională fără să sincronizăm comunicarea explicit, se poate întâmpla ca un proces să citească datele pe care tot el le-a scris, lucru pe care în general nu îl dorim.

În exemplul de mai jos, două procese părinte și copil comunică din nou folosind pipe-uri anonime. De data asta, comunicarea va fi bidirecțională. Părintele inițializează o variabilă de tip caracter pe care o transmite copilului. Urmează 10 iterații în care fiecare proces primește litera, o afișează, apoi transmite litera următoare interlocutorului.

```
1   int main(void)
2   {
3       int fdP2C[2], fdC2P[2];
4       char c = 0;
5       int i;
6
7       if(pipe(fdP2C) != 0 || pipe(fdC2P) != 0) {
8           perror("Could not create pipes");
9           return 1;
10      }
11
12      if(fork() != 0) {
13          //parent
14          close(fdP2C[0]);
```

```
15          close(fdC2P[1]);
16          c = 'a';
17          for(i=0; i<10; i++) {
18              write(fdP2C[1], &c, sizeof(c));
19              read(fdC2P[0], &c, sizeof(c));
20              printf("Parent: %c\n", c);
21              c++;
22          }
23          close(fdP2C[1]);
24          close(fdC2P[0]);
25          wait(NULL);
26      } else {
27          //child
28          close(fdP2C[1]);
29          close(fdC2P[0]);
30          for(i=0; i<10; i++) {
31              read(fdP2C[0], &c, sizeof(c));
32              printf("Child: %c\n", c);
33              c++;
34              write(fdC2P[1], &c, sizeof(c));
35          }
36          close(fdP2C[0]);
37          close(fdC2P[1]);
38      }
39
40      return 0;
41  }
```

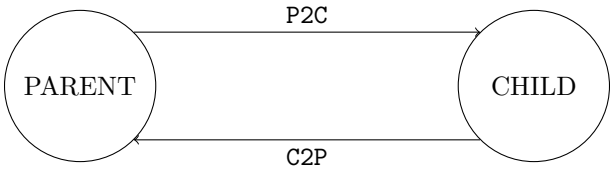


Figura 11.2: Comunicarea bidirecțională între două procese

Figura 11.2 ilustrează modul de comunicare între procesele părinte și copil din exemplul curent. Se folosesc două pipe-uri, pe care le vom numi `P2C` (*parent to child*) și `C2P` (*child to parent*).

La linia 7 se inițializează cele două pipe-uri anonime, iar la linia 12 se apelează `fork` pentru crearea procesului fiu. Ambele procese încep prin a închide capetele nefolosite ale pipe-urilor (liniile 14–15 și 28–29).

Procesul părinte inițializează variabila `c` cu litera `'a'` (linia 16), apoi începe o buclă de 10 iterații în care:

- transmite litera către copil prin pipe-ul `P2C` (linia 18);
- citește litera primită de la copil prin pipe-ul `C2P` (linia 19);
- afișează litera (linia 20);
- incrementează litera (linia 21).

Procesul copil conține o buclă similară, dar care începe cu citirea literei de pe pipe-ul `P2C` (linia 31) și se termină cu trimiterea literei incrementate către părinte, prin pipe-ul `C2P` (linia 34).

Executând programul se poate observa că cele două procese ajung să afișeze alternativ primele 20 de litere din alfabet (de la `'a'` la `'t'`).

11.3 Mersul lucrării

11.3.1 Întrebări recapitulative

1. În ce ordine sunt citiți octeții scriși într-un pipe?
2. Ce se întâmplă când încercăm să citim dintr-un pipe gol, care are un scriitor conectat? Dar dacă nu există un scriitor conectat?
3. În ce situații preferăm să folosim un pipe cu nume față de unul anonim?
4. Ce primește ca argument apelul de sistem `pipe` și la ce este utilizat acest argument?
5. Cum pot două procese să comunice bidirecțional?

11.3.2 Probleme rezolvate

1. (*l11p1_writer.c* și *l11p1_reader.c*) Să se scrie două programe care comunică printr-un pipe cu nume, primul transmițând o valoare întreagă către al doilea. Programele sunt descrise în secțiunea 11.2.2.1.
2. (*l11p2_unnamed.c*) Să se se scrie un program care dă naștere unui proces copil cu care comunică printr-un pipe anonim, transmițând o valoare întreagă. Programul este descris în secțiunea 11.2.3.1.
3. (*l11p3_bidir.c*) Să se scrie un program care creează un proces copil cu care comunică bidirecțional prin pipe-uri anonime, transmițând o literă de la unul la altul și incrementând-o. Programul este descris în secțiunea 11.2.4.

11.3.3 Probleme propuse

1. (25p) Modificați exemplul din secțiunea 11.2.3.1 astfel încât să se transmită un string de dimensiune variabilă în loc de întreg. Ca soluții posibile ar fi să citiți de pe pipe caracter cu caracter până la terminatorul `'\0'` sau să transmiteți înainte de șirul propriu-zis lungimea lui, ca un întreg (reprezentat pe număr fix de octeți).
2. (25p) Folosind informația că un proces este blocat atunci când scrie într-un pipe plin, scrieți un program care determină dimensiunea maximă pentru un pipe cu nume și pentru un pipe anonim.
3. (25p) Scrieți două programe *client.c* și *server.c*. Într-o buclă, clientul va citi doi întregi și o operație de efectuat (`'+'` sau `'-'`) și le va transmite printr-un pipe server-ului. Server-ul va efectua operația și va transmite rezultatul către client, care îl va afișa pe ecran. Clientul se poate termina când operatorul introdus este `'x'`.
4. (25p) Implementați comunicarea bidirecțională din exemplul din secțiunea 11.2.4 utilizând un singur pipe în loc de două. Evitați situația ca un proces să citească datele scrise de el însuși în pipe, sincronizând explicit comunicarea.

Laborator 12

Maparea fișierelor în memorie și memoria partajată

12.1 Scopul lucrării

Laboratorul curent va prezenta principalele concepte legate de maparea fișierelor în memorie și memoria partajată, precum și apelurile de sistem Linux, în varianta POSIX pentru lucrul cu aceste concepte.

12.2 Fundamente teoretice

12.2.1 Maparea fișierelor în memorie

Operațiile comune pe fișiere (`read` și `write`) ajută la accesarea secvențială a unui fișier (citirea sau scrierea datelor "în ordine"). Atunci când dorim acces aleator la un fișier (vrem să accesăm date de la adrese arbitrare) putem folosi apelul de sistem `lseek`, dar apelul frecvent al acestuia și ținerea evidență poziției cursorului pot complica un program.

Maparea unui fișier în memorie simplifică operațiile asupra acestuia, oferind utilizatorului acces aleator la nivel de octet, ca și cum întreg fișierul ar fi fost citit în memorie. În realitate, spațiul este doar rezervat în memoria virtuală, urmând să se facă citiri doar atunci când datele sunt accesate.

12.2.2 Memoria partajată

Memoria partajată este un mecanism de comunicare între procese, care permite existența unei zone de memorie comună ambelor procese.

În condiții normale, thread-urile împart spațiul de adrese din procesul în care rulează, în timp ce procesele au spații de adrese diferite, pentru a fi protejate unele de altele. Totuși, există situații când dorim ca două procese să comunice împărțind o zonă de memorie comună.

Figura 12.1 ilustrează modul în care sistemul de operare implementează conceptul de memorie partajată. Fiecare proces are la dispoziție un spațiu de adrese, din care anumite pagini sunt utilizate, având un corespondent în memoria fizică. Legătura dintre memoria virtuală și memoria fizică se face transparent, un proces neputând adresa memorie din afara spațiului său de adrese. Chiar dacă două procese P_1 și P_2 folosesc aceleași adrese de memorie virtuală, sistemul de operare le va asocia în mod normal cu adrese diferite din memoria fizică. Doar dacă procesele cer explicit memorie partajată, paginile acestora vor fi asociate cu aceeași zonă de memorie fizică. În acest caz, o scriere în memorie partajată efectuată de P_1 va fi văzută automat de P_2.

12.2.3 Lucrul cu fișiere mapate în memorie

```
#include <sys/mman.h>
```

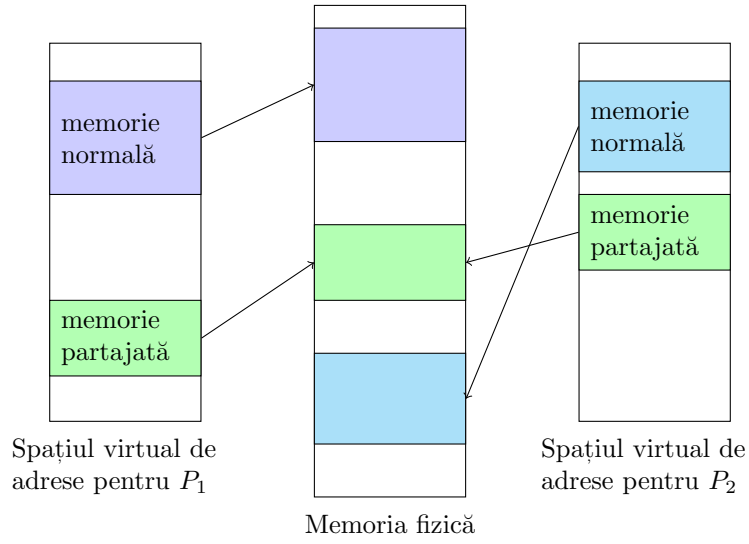


Figura 12.1: Modul în care sistemul de operare implementează conceptul de memorie partajată

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

12.2.3.1 mmap

Apelul de sistem `mmap` ne permite maparea în memorie a unui fișier sau chiar obținerea unei zone de memorie partajată cu alt proces.

Pentru a mapa un fișier în memorie, trebuie să îl deschidem întâi folosind apelul de sistem `open`, descriptorul de fișier obținut fiind transmis ca al 5-lea argument funcției `mmap`. Primul argument al funcției, `addr` poate fi `NULL`, caz în care sistemul de operare alege adresa unde se va realiza maparea sau se poate forța o adresă specificată.

`mmap` ne permite să nu mapăm neapărat tot fișierul, putând să specificăm un offset, ca ultim argument, respectiv un număr de octeți care trebuie mapați, ca argumentul 2.

Argumentul al treilea, `prot` specifică protecția paginilor mapate (adică modul în care memorie poate fi accesată) , care poate fi `PROT_NONE` sau o combinație dintre flag-urile `PROT_READ`, `PROT_WRITE` și `PROT_EXECUTE`. Protecția memoriei mapate trebuie să fie în acord cu modul în care fișierul a fost deschis, adică, de exemplu, dacă protecția este `PROT_READ`, modul de deschidere a fișierului trebuie să fie `O_RDONLY` sau *O_RDWR*.

Al 4-lea argument, `flags`, trebuie să fie una din valorile `MAP_SHARED` și `MAP_PRIVATE` în combinație cu anumite flag-uri specifice operației de mapare. `MAP_SHARED` indică faptul că maparea poate fi partajată cu alte procese, iar în cazul în care un fișier este mapat pentru scriere, modificările vor ajunge pe disc. `MAP_PRIVATE` nu permite partajarea mapării cu alte procese și nu propagă modificările la fișier.

Funcția `mmap` returnează un pointer la adresa unde s-a făcut maparea, în caz de succes, sau `(void*)-1`, în caz de eșec.

12.2.3.2 munmap

Acest apel de sistem închide o mapare obținută cu `mmap`. Primul argument reprezintă adresa în memorie unde s-a realizat maparea, în timp ce al doilea conține dimensiunea mapării.

12.2.3.3 Exemplu

Exemplul de mai jos va afișa conținutul unui fișier invers, folosind tehnica mapării în memorie. Fără această tehnică, soluția ar fi să facem operații de `lseek` și `read` pentru fiecare caracter. O altă soluție ar fi fost să citim tot fișierul într-un vector în memorie, apoi să parcurgem invers vectorul

citit. A doua soluție ar fi mai ușor de scris, dar pentru fișiere mari am avea un consum mare de memorie.

Tehnica mapării în memorie ne oferă acces la fișier ca și cum acesta ar fi fost citit în întregime, dar realizând citirea în mod transparent, doar atunci când datele sunt necesare.

```
1  int main(int argc, char **argv)
2  {
3      int fd;
4      off_t size, i;
5      char *data = NULL;
6
7      if(argc != 2) {
8          fprintf(stderr, "Usage: %s <file_name>\n", argv[0]);
9          return 1;
10     }
11
12     fd = open(argv[1], O_RDONLY);
13     if(fd == -1) {
14         perror("Could not open input file");
15         return 1;
16     }
17     size = lseek(fd, 0, SEEK_END);
18     lseek(fd, 0, SEEK_SET);
19
20     data = (char*)mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
21     if(data == (void*)-1) {
22         perror("Could not map file");
23         close(fd);
24         return 1;
25     }
26
27     for(i=size-1; i>=0; i--) {
28         printf("%c", data[i]);
29     }
30     printf("\n");
31
32     munmap(data, size);
33     close(fd);
34
35     return 0;
36 }
```

Programul începe prin verificarea numărului de argumente primite în linia de comandă (linia 7), apoi se realizează deschiderea fișierului în citire (linia 12). Modul de deschidere a fișierului trebuie să corespundă modului în care vom accesa pe viitor memoria partajată. În cazul acesta am deschis în citire, deoarece vrem doar să citim datele, fără să facem modificări.

Următorul pas este aflarea dimensiunii fișierului apelând `lseek` pentru a ajunge la finalul acestuia (linia 17). Dimensiunea fișierului se poate afla și folosind apelul de sistem `fstat`.

Urmează maparea fișierului în memorie la linia 20, unde folosim descriptorul de fișier `fd` și dimensiunea obținută anterior `size`. Lăsăm sistemul de operare să decidă adresa mapării, transmițând `NULL` ca prim argument și cerem ca paginile mapate să fie accesibile doar în citire (`PROT_READ`). Nu dorim să partajăm maparea, nici să salvăm modificări asupra fișierului, deci alegem modul `MAP_PRIVATE`. Funcția `mmap` returnează un pointer la începutul memoriei mapate, pe care îl salvăm în variabila `data`.

Având la dispoziție memoria mapată, parcurgea în ordine inversă a fișierului se rezumă la o simplă buclă `for` descrescătoare (liniile 27–29).

În final, trebuie să închidem maparea (linia 32) și fișierul (linia 33). Închiderea fișierului poate

fi făcută și imediat după mapare, deoarece funcția `mmap` creează în mod intern un duplicat al descriptorului de fișier.

12.2.4 Lucrul cu memoria partajată

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);

int ftruncate(int fd, off_t length);
```

Atunci când utilizăm memoria partajată, trebuie să cerem compilatorului (componentei *linker*) să adauge biblioteca *rt*, adăugând în linia de comandă opțiunea `-lrt`.

Compilarea programelor care folosesc memoria partajată

```
gcc -Wall program.c -o program -lrt
```

12.2.4.1 shm_open

Apelul de sistem `shm_open` creează sau deschide o zonă de memorie partajată. Funcționează în mod similar cu apelul de sistem `open` pentru fișiere, returnând un descriptor de fișier, care apoi se poate mapa în memorie cu `mmap`.

Primul parametru, `name`, este numele memoriei partajate. Dacă două procese apelează `shm_open` cu același nume, vor primi acces la aceeași zonă de memorie. Standardul POSIX recomandă ca numele unei memorii partajate să înceapă cu caracterul `'/'`, să nu mai conțină alte caractere `'/'` în afară de primul și să aibă maxim 255 de caractere.

Parametrul `oflag` poate fi `O_RDONLY` (citire) sau `O_RDWR` (citire și scriere), combinat prin SAU logic cu alte flag-uri, precum `O_CREAT`, care indică crearea memoriei partajate, în caz că nu există.

Parametrul `mode` conține permisiunile memoriei partajate (similare cu permisiunile unui fișier) și este luat în considerare doar când memoria se creează.

În caz de eșec se returnează o valoare negativă.

12.2.4.2 shm_unlink

Acest apel de sistem realizează ștergerea unei zone de memorie partajată, primind ca parametru numele acesteia (același nume care a fost folosit și la `shm_open`).

După ce se apelează `shm_unlink` memoria partajată nu se mai poate deschide (decât dacă se creează din nou), dar procesele care o mapează o pot accesa în continuare, ștergerea efectivă efectuându-se doar după ce ultimul proces a închis maparea.

12.2.4.3 ftruncate

Acest apel de sistem funcționează atât pentru fișiere obișnuite cât și pentru zone de memorie partajată și are rolul de a redimensiona fișierul cu descriptorul `fd`, la noua dimensiune `length`.

În cazul unei zone de memorie partajată, e important să apelăm `ftruncate` după ce aceasta a fost creată, pentru a defini dimensiunea acesteia.

12.2.4.4 Exemplu

Vom prezenta un exemplu simplu, în care două procese partajează o zonă de memorie de un singur octet. Primul proces creează zona de memorie partajată și scrie în ea un anumit caracter, apoi așteaptă ca un alt proces să îl modifice.

Atunci când declarăm în C un pointer către o zonă de memorie partajată (linia 3) se recomandă utilizarea cuvântului-cheie `volatile`, care indică compilatorului faptul că acea variabilă se poate modifica în moduri neprevăzute (de exemplu de către un alt proces cu care se partajează memoria). Operațiile pe variabila respectivă nu vor fi optimizate și vor fi efectuate precum în codul sursă.

p1.c:

```
1  int main(void) {
2      int shmFd;
3      volatile char *sharedChar = NULL;
4
5      shmFd = shm_open("/l12_myshm", O_CREAT | O_RDWR, 0600);
6      if(shmFd < 0) {
7          perror("Could not aquire shm");
8          return 1;
9      }
10     ftruncate(shmFd, sizeof(char));
11     sharedChar = (volatile char*)mmap(0, sizeof(char),
12         PROT_READ | PROT_WRITE, MAP_SHARED, shmFd, 0);
13     if(sharedChar == (void*)-1){
14         perror("Could not map the shared memory");
15         return 1;
16     }
17
18     *sharedChar = 'A';
19     while(*sharedChar == 'A') {
20         printf("sharedChar: %c\n", *sharedChar);
21         sleep(1);
22     }
23     printf("sharedChar new value: %c\n", *sharedChar);
24
25     munmap((void*)sharedChar, sizeof(char));
26     sharedChar = NULL;
27     close(shmFd);
28     shm_unlink("/l12_myshm");
29
30     return 0;
31 }
```

Primul program, *p1*, are rolul de crea zona de memorie partajată, apelând `shm_open` la linia 5. Numele zonei de memorie `"/l12_myshm"` este folosit și de *p2*, pentru a lucra cu aceeași zonă de memorie. După ce obținem descriptorul de fișier `shmFd`, trebuie să apelăm `ftruncate` pentru a stabili dimensiunea zonei de memorie (linia 10), în acest caz 1 byte (`sizeof(char)`).

În continuare, zona de memorie partajată va fi mapată în spațiul curent de adrese, folosind `mmap` (liniile 11–12). Spre deosebire de exemplul din secțiunea 12.2.3.3, memoria mapată trebuie să fie partajată (folosim `MAP_SHARED`), iar protecția paginilor să fie atât `PROT_READ` cât și `PROT_WRITE`. Rezultatul operației este un pointer pe care îl vom stoca în variabila `sharedChar`.

La linia 18 se setează conținutul memoriei partajate la litera `'A'`, apoi urmează o buclă în care se verifică dacă acest conținut s-a modificat (liniile 19–22). După modificare, se afișează noua valoare (linia 23), apoi se închide maparea (linia 25), descriptorul `shmFd` (linia 27) și se șterge zona de memorie partajată folosind apelul de sistem `shm_unlink` (linia 28).

După închiderea mapării, pointer-ul `sharedChar` încă pointează la vechea adresă, care acum este invalidă. Este o practică bună în C ca pointerii rămași invalizi să fie setați la `NULL`, astfel încât la reutilizarea acestora necorespunzătoare să nu corupem date.

p2.c:

```
1  int main(void) {
2      int shmFd;
```

```
3     volatile char *sharedChar = NULL;
4
5     shmFd = shm_open("/l12_myshm", O_RDWR, 0);
6     if(shmFd < 0) {
7         perror("Could not aquire shm");
8         return 1;
9     }
10    sharedChar = (volatile char*)mmap(0, sizeof(char),
11        PROT_READ | PROT_WRITE, MAP_SHARED, shmFd, 0);
12    if(sharedChar == (void*)-1){
13        perror("Could not map the shared memory");
14        return 1;
15    }
16
17    printf("found sharedChar: %c\n", *sharedChar);
18    *sharedChar = 'X';
19
20    munmap((void*)sharedChar, sizeof(char));
21    sharedChar = NULL;
22    close(shmFd);
23
24    return 0;
25 }
```

Programul *p2* accesează memoria partajată creată de *p1*, folosind aceeași valoare pentru parametrul `name` în apelul funcției `shm_open` (linia 5). Modul de deschidere va fi `O_RDWR`, iar permisiunile au valoarea 0, deoarece apelul curent nu creează nimic, doar obține descriptorul zonei de memorie ce a fost creată în prealabil de *p1*. Deasemenea, nu mai e nevoie să apelăm `ftruncate`, deoarece dimensiunea memorie partajate a fost deja stabilită.

În mod similar cu *p1*, mapăm zona de memorie (liniile 10–11), obținând pointer-ul `sharedChar`, a cărui valoare o afișăm (linia 17) apoi o setăm la caracterul `'X'` (linia 18). Programul se termină prin închiderea mapării (linia 20), setarea pointer-ului la `NULL` (linia 21) și închiderea descriptorului de fișier (linia 22).

Vom rula întâi programul *p1*, care creează zona de memorie, apoi așteaptă în buclă modificarea caracterului din aceasta. Putem observa că o dată pe secundă, valoarea este afișată. Într-un alt terminal, rulăm *p2* și observăm că acesta găsește caracterul `'A'` în memoria partajată, apoi îl modifică. Între timp, dacă revenim la terminalul în care am rulat *p1*, observăm că după rularea lui *p2* valoarea caracterului s-a modificat, iar programul a ieșit din buclă și s-a încheiat.

Observație: Cele două programe vor rula ca mai sus doar dacă al doilea program este pornit după primul. Altfel, cel de-al doilea program nu va găsi memoria partajată și își va încheia execuția afișând un mesaj de eroare la linia 7. O implementare robustă ar putea folosi semafoare cu nume pentru a sincroniza cele două programe, inclusiv accesul la memoria partajată.

12.3 Mersul lucrării

12.3.1 Întrebări recapitulative

1. Ce returnează funcția `mmap`?
2. Care este avantajul de a mapa un fișier în memorie în loc să îl citim folosind `read`?
3. Cu ce apel de sistem deschidem o zonă de memorie partajată?
4. Ce face apelul de sistem `ftruncate` și la ce e util în lucrul cu memoria partajată?
5. Dați un exemplu de situație când se preferă comunicarea a două procese prin memorie partajată în loc de pipe-uri.

6. Ce beneficii aduce comunicarea prin pipe-uri în loc de memoria partajată?

7. Cum putem sincroniza două procese diferite utilizând un lacăt (mutex)?

12.3.2 Probleme rezolvate

1. (*l12p1_invert.c*) Să se scrie un program care afișează conținutul unui fișier pe ecran, inversat, folosind tehnica mapării în memorie. Programul este descris în secțiunea 12.2.3.3.

2. (*l12p2_shm1.c* și *l12p2_shm2.c*) Să se scrie două programe care comunică prin memorie partajată. Primul program va scrie un caracter în zona de memorie partajată, apoi va aștepta ca al doilea program să îl modifice. Programele sunt descrise în secțiunea 12.2.4.4.

12.3.3 Probleme propuse

1. (25p) Modificați exemplul din secțiunea 12.2.3.3 astfel încât conținutul fișierului să fie inversat pe disc, în loc să se afișeze pe ecran.

2. (25p) Implementați două programe (sau un program care pornește și un proces copil) care incrementează în mod alternativ o variabilă întreagă aflată într-o zonă de memorie partajată.

3. (25p) Utilizând tehnica mapării în memorie, scrieți un program care șterge toate vocalele dintr-un fișier, umplând spațiul liber rămas la finalul fișierului cu caractere ‘ ’ (spații).

4. (35p) Implementați un program care caută numere prime într-un interval primit ca argumente în linia de comandă și le adaugă la finalul unui vector aflat într-o zonă de memorie partajată. Vectorul va avea pe prima poziție numărul de elemente scrise până în prezent, urmate de elementele propriu-zise. Scrierea în memoria partajată trebuie sincronizată cu un semafor. Implementați de asemenea un program *manager* care inițializează memoria partajată și vectorul din aceasta, apoi împarte un interval primit la N procese copii, care vor căuta numerele și le vor scrie în memorie. La final, *manager*-ul va afișa toate numerele din vector.

Laborator 13

Probleme recapitulative

13.1 Scopul lucrării

În laboratorul curent se vor folosi cunoștințele acumulate de-a lungul semestrului pentru rezolvarea unor probleme de programare C pe Linux.

13.2 Mersul lucrării

13.2.1 Probleme propuse

1. (35p) Scrieți un program care creează un proces fiu și comunică cu el printr-un pipe anonim (eng. *nameless pipe*). Părintele va deschide un fișier binar, îl va citi octet cu octet și va transmite doar cifrele (octeții a căror valoare reprezintă codul ASCII al unei cifre) către copil, prin pipe. La terminarea fișierului, se va transmite pe pipe un octet cu valoarea 0. Procesul copil va primi cifrele care vin pe pipe și va afișa pe ecran doar cifrele pare. La primirea valorii 0 (nu a cifrei '0'), se va închide.

2. (35p) Scrieți un program care primește în linia de comandă mai multe cuvinte și creează câte un thread pentru procesarea fiecărui cuvânt. Thread-urile vor număra câte vocale conține cuvântul primit și vor aduna valoarea obținută la o variabilă globală `vowelCount`. Thread-ul principal va aștepta după thread-urile create și va afișa variabila. Accesul thread-urilor la variabilă trebuie sincronizat.

3. (30p) Folosind apelul de sistem `fork()`, implementați ierarhia de procese din Figura 13.1. Fiecare proces trebuie să aștepte terminarea propriilor copii. Procesul P_3 trebuie să pornească doar după ce P_4 s-a terminat.

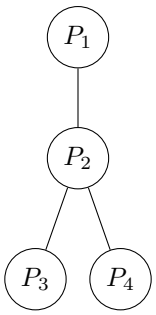


Figura 13.1: Ierarhia de procese

4. (30p) Scrieți un program care primește ca argumente în linia de comandă calea spre un fișier și două string-uri $s1$ și $s2$ de aceeași lungime. Programul va înlocui toate aparițiile lui $s1$ din fișierul dat cu $s2$. Se va lucra direct pe fișier, nu se va construi un fișier destinație.

5. (35p) Scrieți un program care primește ca argumente în linia de comandă căile spre două fișiere obișnuite, care se presupun a fi de aceeași dimensiune și inversează conținutul acestora. Astfel, dacă inițial *f1* conține `1234567` și *f2* conține `abcdefg`, după rularea programului *f1* va conține `abcdefg`, iar *f2* conține `1234567`. Pentru punctaj maxim nu se vor folosi fișiere auxiliare și se va presupune că fișierele pot fi oricât de mari, deci conținutul acestora nu se poate citi în întregime în memorie.

6. (40p) Un număr perfect este un număr natural, egal cu suma divizorilor săi, diferiți de el însuși (ex. $6 = 1 + 2 + 3$). Scrieți un program care afișează toate numerele perfecte din intervalul $[1 \dots N]$, folosind M thread-uri. N și M se primesc ca argumente în linia de comandă.

   - Pentru 60% din punctaj, fiecare thread poate să afișeze numerele perfecte, pe măsură ce le găsește.
   - Pentru punctaj maxim, numerele găsite trebuie colectate într-un vector și să fie afișate doar la final, de către thread-ul principal. Accesul la vector trebuie sincronizat.

7. (35p) Scrieți două programe `p1.c` și `p2.c` care comunică printr-un pipe cu nume. `p1` va parcurge un folder (nerecursiv) și va transmite prin pipe numele fișierelor din acesta (nu și a subfolderelor). `p2` va afișa pe ecran doar numele de fișiere (dintre cele primite) care încep cu o cifră.