Universitatea "Politehnica" Timisoara
Departamentul Calculatoare
Digital Signal Processing Labs Timisoara

Sisteme de operare 2

Valentin STÂNGACIU

Ver. 1.59/2025

Secțiunea I

Introducere

Introducere

Disciplina Sisteme de Operare 2

- SO2 – disciplină obligatorie – secția Tehnologia Informației
- Precondiții
  - Tehnici de Programare
  - Programarea Calculatoarelor
  - Sisteme de Operare 1
  - Rețele de calculatoare
- Obiectivele disciplinei:
  - Înțelegerea arhitecturii și principiilor de funcționare a sistemelor de operare Linux
  - Proiectarea și implementarea de aplicații complexe în sisteme de operare Linux
  - Cunoștințe avansate de instalare/operare/depanare a sistemelor de operare Linux
- Organizarea disciplinei:
  - 28 ore de curs: 2 ore curs / săptămână
  - 28 ore de laborator: 2 ore laborator / săptămână
  - 14 ore de proiect: 1 ora proiect / săptămână
- Suport desfasurare si evaluare disciplină disciplină:
  - **Campus Virtual –** materiale curs, materiale laborator, discutii, anunturi, note, prezente, medii

Introducere

Disciplina Sisteme de Operare 2

- Evaluarea disciplinei:
  - Evaluare finală prin examen oral cu rezolvare de probleme practice pe calculator
  - Evaluare pe parcurs la laborator
  - Evaluare proiect
- Nota activitate pe parcurs:

$$AP = \frac{L + Pr}{2} + P; L \geq 5; Pr \geq 5$$

L – medie laborator, Pr – medie proiect, P – punct prezență curs

- Notă finală disciplină:

$$M_{SO2} = ROUND\,(0.5 \cdot E + 0.5 \cdot AP) = ROUND\left(\frac{E + AP}{2}\right),\ k_1 = 0.5,\ \ k_2 = 0.5$$

E – notă examen, AP – notă activitate pe parcurs, L – nota laborator, P – punct prezență curs

$$P = \begin{cases} +1,\ PrezentaCurs \in [90, 100]\% \\ 0,\ PrezentaCurs \in [60, 90)\ \% \\ -1,\ PrezentaCurs \in [30, 60)\ \% \\ -2,\ PrezentaCurs \in [0, 30)\ \% \end{cases}$$

- Punct prezență curs – se acordă doar daca $AP \geq 5$
- Notele la laborator, proiect, activitate pe parcurs si examen nu se rotunjesc, se calculează cu 2 zecimale
- Media finală se rotunjește conform regulamentului UPT

Introducere

Disciplina Sisteme de Operare 2

- Evaluare laborator:
  - 3 teste evaluare – se cere rezolvarea unei probleme prin implementare in C –
  - Medie laborator: medie aritmetică a celor 3 teste
  - Promovare laborator: medie laborator minim 5
- Evaluare proiect:
  - Se vor acorda 2 note: 1 nota pe activitate pe parcurs la proiect (30%) și o nota pe realizarea proiectului (70%)
  - Nota activitate pe parcurs la proiect – pe baza îndeplinirii anumitor etape în planificarea și realizarea proiectului
  - Medie proiect: 30 % nota activitate proiect + 70% nota finala proiect
  - Promovare laborator: medie finala proiect minim 5
- Activitate pe parcurs
  - Media aritmetică a mediilor finale de la proiect și laborator
  - Pentru a se calcula este necesar ca atât media de la proiect cât și media de la laborator să fie minim 5
- Examen
  - In format oral în laborator
  - Se va primi spre rezolvare o problema pe o durată de timp de max 60-90 min
  - Se va evalua succint problema – condiții minime: cod compilabil cu minimă funcționalitate
  - Se va discuta pe marginea problemei cu întrebări specifice din tematica probleme sau din orice tematică din curs
  - Se poate cere studentului exemplificare prin cod a unor teme din curs cu o discuție pe marginea acestora
  - Se poate cere studentului scrierea de cod adițional pentru a oferi examinatorului o imagine clară a gradului de cunoștințe

Introducere

Sisteme de Operare 2 – Echipa didactică

- **Titular curs**: sl. dr. ing. Valentin STÂNGACIU
  - Contact: B513, B417, valentin.stangaciu@cs.upt.ro
  - Domenii de interes: sisteme embedded, sisteme de operare, sisteme timp-real, protocoale de comunicații, rețele de senzori
- **Echipa de laborator și proiect**:
  - as. drd. ing. Petra CSEREOKA
    - Contact: B417, petra.csereoka@cs.upt.ro
    - Domenii de interes: Modeling, formal verification and testing, Embedded systems, Intelligent robotic environments, Operating systems, Artificial intelligence, Machine learning, Neural network

Introducere

Sisteme de Operare 2 – Precondiții

- **Programarea Calculatoare, Tehnici de programare**
  - Dezvoltare, compilare, depanare programe în limbajul C
  - programare de bază
  - alocare dinamică
  - prelucrare string-uri
  - tipuri de date utilizator (struct, union, typedef, enum)
  - fișiere text și binare,
  - funcții cu număr variabil de argumente (variadic functions)
  - pointeri, pointeri la funcții
  - Argumente în linie de comandă
  - programe complexe cu mai multe fișiere C și h
  - Preprocesorul C – compilare condiționată #ifndef…. #endif, macro, …
  - Compilare – linkeditare separată pe pași
  - Realizarea de biblioteci statice, dinamice (shared objects)

Introducere

Sisteme de Operare 2 – Precondiții

- **Sisteme de operare**
  - Operare de bază în Linux
  - Shell script
  - Sistemul de fișiere
  - Drepturi de acces fișiere și directoare
  - Parcurgere directoare: opendir(…), readdir(…)
  - Obținere inode – stat(…)
  - Citire/scriere fișiere folosind apeluri sistem open(…), read(…), write(…), close(…)
  - Procese: fork(…), wait(…), waitpid(…), exit(…)
  - Pipe: creare pipe, citire/scriere pipe, redirectare
  - Familia de funcții exec: execl, execlp, execle, execv, execvp, execvpe
  - Thread-uri: biblioteca pthread
- **Rețele de calculatoare**
  - Noțiuni elementare: ip, netmask, dns, socket, TCP, UDP
  - Utilizare wireshark

Introducere

Sisteme de Operare 2

- **Mediul de lucru**
  - Sisteme de operare: **GNU Debian** sau Ubuntu (Canonical)
    - Masini fizice
    - Masini virtuale folosind VirtualBox
    - Exclus MacOS !
  - Mediu de programare: orice mediu de programare C cu compilatorul gcc
    - Linie de comanda, editor de texte, … etc

      SAU
    - IDE – VSCode, Eclipse, …
    - Se oferă SUPORT pentru varianta in linie de comanda cu editor de texte separat si executie in linie de comanda:
  - Mediu de executie: in linie de comanda
  - Wireshark

Introducere

Sfaturi didactice

- **Recapitulați limbajul C**
- **Recapitulați sisteme de operare**
- **Veniți la curs**
- **Citiți materialul de laborator înainte de ora**
- **Scrieți mult cod, rezolvați probleme**
- **Nu folosiți ChatGPT**
- **Citiți paginile de manual – *manpage* (informațiile din suportul de curs sunt insuficiente – se completează cu informațiile din pagina de manual)**
- **Luați notițe – nu toate informațiile prezentate la curs sunt și în suportul de curs**

Secțiunea II

Recapitulare și completare

Recapitulare

Elemente de utilizare Linux

- Sistemul de fișiere
  - Structură arborescentă cu intrări bine definite
  - Rădăcina sistemului (/) – denumit și root (rădăcină)
  - Toate căile de fișiere și directoare se pot scrie absolut relativ la /
  - Suport pentru fișiere, directoare (foldere – Windows), legături simbolice (≈ shortcut – Windows)
  - Diferită fundamental și structural față de sistemul de fișiere din Windows dar prezintă și asemănări
  - Identic pentru toate variațiile (distribuțiile) de Linux și Unix
  - Directoarele, driverele, echipamentele periferice – tipuri diferite și specializate de fișiere

```
/
|--                                   bin
|--                                   boot
|--                                   dev
|--                                   etc
|--                                   home
|                       |--                 student
|                       |--                 student1
|--                                   lib
|--                                   lib32
|--                                   lib64
|--                                   media
|--                                   mnt
|--                                   opt
|--                                   proc
|--                                   root
|--                                   run
|--                                   sbin
|--                                   srv
|--                                   sys
|--                                   tmp
|--                                   usr
|                          |--                bin
|                       |--                   games
|                       |--                   include
|                          |--                lib
|                       |--                   lib32
|                       |--                   local
|                          |--                sbin
|                       |--                   share
|                          `--                src
|--                                   var
|--   initrd.img   ->   boot/initrd.img-4.9.0-8-amd64
|--  initrd.img.old  ->  boot/initrd.img-4.9.0-7-amd64
|--     vmlinuz      ->     boot/vmlinuz-4.9.0-8-amd64
`-- vmlinuz.old -> boot/vmlinuz-4.9.0-7-amd64
```

Recapitulare

Elemente de utilizare Linux

- Semnificația principalelor directoare
- **/bin** – “binaries” - binarele (fișierele executabile) ale principalelor programe utilitare fundamentale: ls, cp, ln, less, echo, touch,….
- **/boot** – fișiere necesare pornirii sistemului (kernel, imagine inițială, etc)
- **/dev** – “devices” – fișiere speciale ce reprezintă echipamentele hardware ale sistemului (/dev/mem – întreaga memorie a sistemului, /dev/cdrom – unitatea optică a sistemului)
- **/etc** – dedicat stocării fișierelor de configurare ale programelor instalate în sistem
- **/lib, /lib32, /lib64** – “libraries” – conține bibliotecile software din system (similar cu C:\Windows\System32)
- **/usr** – “user filesystem” – conține programele utilizator instalate în sistem sub formă arborescentă similară cu / (root) - /usr/bin, /usr/lib, …..
- **/tmp** – “temporar” - folosit pentru stocarea de fișiere temporare (similar cu C:\Windows\Temp)
- **/home** – conține directoarele personale ale fiecărui utilizator din sistem
  - Similar cu C:\Users – începând cu Windows 7
  - Fiecare utilizator are aici un director personal în care are drepturi depline de creare, ștergere, execuție fișiere și directoare

```
/
|--                                   bin
|--                                   boot
|--                                   dev
|--                                   etc
|--                                   home
|                       |--                 student
|                       |--                 student1
|--                                   lib
|--                                   lib32
|--                                   lib64
|--                                   media
|--                                   mnt
|--                                   opt
|--                                   proc
|--                                   root
|--                                   run
|--                                   sbin
|--                                   srv
|--                                   sys
|--                                   tmp
|--                                   usr
|                         |--                 bin
|                       |--                   games
|                       |--                   include
|                         |--                 lib
|                       |--                   lib32
|                       |--                   local
|                         |--                 sbin
|                       |--                   share
|                         `--                 src
|--                                   var
|--  initrd.img   ->  boot/initrd.img-4.9.0-8-amd64
|--  initrd.img.old  ->  boot/initrd.img-4.9.0-7-amd64
|--     vmlinuz     ->     boot/vmlinuz-4.9.0-8-amd64
`--  vmlinuz.old  ->  boot/vmlinuz-4.9.0-7-amd64
```

Recapitulare

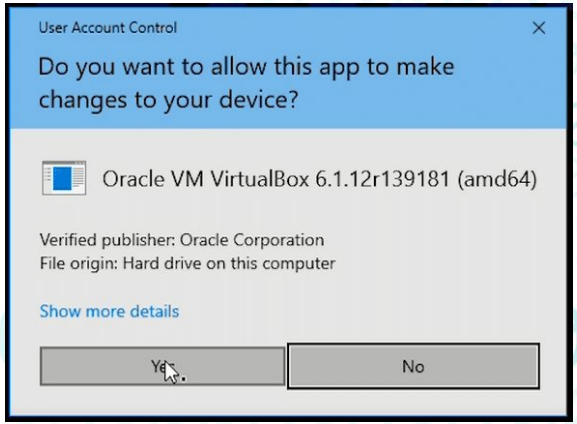
Elemente de utilizare Linux

- Utilizatorii din sistemele Linux
  - Utilizatori obișnuiți
    - Au director personal de home în /home/<nume_utilizator>
    - Au drepturi depline de a crea, sterge, modifica fișiere în directorul home
    - Au drepturi (sau nu) de a executa programre din directorul personal
    - Au drepturi (sau nu) de a executa aplicații/programe puse la dispoziție de sistem
    - NU au drepturi de a instala/dezinstala aplicații din sistem
    - NU au drepturi de a modifica parametrii și configurările sistemului
    - Protejați obligatoriu de o parolă
  - Utilizatorul ***root***
    - Singurul utilizator cu drept de adminstrare în sistemele Linux, UNIX
    - Are drept deplin asupra sistemului
    - Are drept de configurare/instalare/dezinstalare programe/aplicații din sistem
    - Are drept de a modifica drepturile altor utilizatori
    - Are director personal în /root (nu în /home ca și ceilalți utilizatori).
    - Similar cu utilizatorul “Administrator” din Windows

Recapitulare

Elemente de utilizare Linux

- Utilizatorii din sistemele Linux
  - Utilizatorul se poate schimba oricând în timpul rulării sistemului prin logout ☾ login
  - Se poate schimba și în utilizatorul root în orice moment (prin comanda ***su***)
  - Unii utilizatori pot avea dreptul de a cere drept de *root (de administrare)* pentru execuția unor programe și comenzi folosind comanda ***sudo***. (fără a se schimba în userul *root*)
  - Comanda sudo permite unui utilizator să execute o comanda sau program cu drept de root
    - Similar cu user account control (UAC) din Windows



Recapitulare

Elemente de utilizare Linux

- Prompt
  - Sintaxă: user@hostname:path$
    - user – reprezintă numele utilizatorului autentificat
    - hostname – reprezintă numele calculatorului
    - path – reprezintă calea curentă
  - Calea
    - Absolută – se specifică întreaga cale începând cu referința /
      ex: /home/student1/directorul_meu
    - relativă – se specifică calea relativ la calea curentă
      ex: calea curentă /home , calea relativă student1/directorul_meu
    - Semnul ~ (tilda) – reprezintă o scurtătură către directorul personal (home) al utilizatorului autentificat

    Ex: pentru student1 – ~ Ⓟ /home/student1
    pentru root - ~ Ⓟ /root
    pentru student - ~ Ⓟ /home/student
  - Consola/terminalul implicit se deschide în directorul home al utilizatorului autentificat

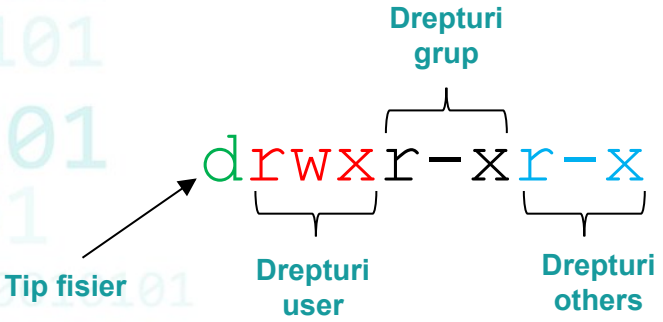
```
/
|--                                   bin
|--                                   boot
|--                                   dev
|--                                   etc
|--                                   home
|                       |--                 student
|                       |--                 student1
|--                                   lib
|--                                   lib32
|--                                   lib64
|--                                   media
|--                                   mnt
|--                                   opt
|--                                   proc
|--                                   root
|--                                   run
|--                                   sbin
|--                                   srv
|--                                   sys
|--                                   tmp
|--                                   usr
|                          |--                bin
|                          |--                games
|                          |--                include
|                          |--                lib
|                          |--                lib32
|                          |--                local
|                          |--                sbin
|                          |--                share
|                          `--                src
|--                                   var
|--   initrd.img   ->   boot/initrd.img-4.9.0-8-amd64
|--  initrd.img.old  ->  boot/initrd.img-4.9.0-7-amd64
|--     vmlinuz      ->     boot/vmlinuz-4.9.0-8-amd64
`-- vmlinuz.old -> boot/vmlinuz-4.9.0-7-amd64
```

Recapitulare

Elemente de utilizare Linux

- Comenzi de bază
  - Schimbarea directorului: *cd* (change directory)
  - Listare conținutului unui director: *ls* (list directory contents)
  - Crearea unui director: *mkdir* (make directory)
  - Pagina de manual: *man*
  - Afisarea unui fișier text în terminal: *cat*
  - Vizualizare procese: *ps*
  - Redirectare intrare,iesire-standard, pipe: < > |
  - Expresii regulate: comanda grep
  - Alte Comenzi: head, tail , sort, uniq, cut, tr, tee, wc, xargs, who, echo, touch, date, ln, du, df, ln, chmod, chown, find

Recapitulare



- primul byte reprezintă tipul fișierului. Acesta poate fi:
  - caracterul '-' (minus) în cazul în care este un fișier obișnuit (regular file)
  - caracterul ‘d’ – în cazul în care fișierul respectiv este director
  - caracterul ‘l’ – în cazul în care fișierul reprezintă o legătură simbolică
  - caracterul ‘c’ – în cazul în care fișierul este un fișier de tip caracter

- Următorii octeți pot constitui 3 grupe care se referă la drepturile asupra acestui fișier pentru:
  - utilizatorul owner (prima grupă - octeții 2,3,4)
  - utilizatorul care face parte din același grup ca și fișierul (a doua grupă - octeții 5,6,7)
  - ceilalți utilizatori (cei ce nu sunt nici owner și nici nu fac parte din același grup din care face parte fișierul) - a treia grupă - octeții 8,9,10)

Drepturi de acces

```
drwxr-xr-x  5 valy staffcs      4096 Jan  8  2018  ipxe
drwxr-xr-x  2 valy staffcs      4096 Oct 18  2016  key
drwxr-xr-x  2 valy staffcs      4096 Dec  2  2019  mac_process
drwxr-xr-x  3 valy valy         4096 Jan  7 15:31  mate
lrwxrwxrwx  1 valy valy           26 May 28 10:40  my_link -> /etc/zsh_command_not_found
-rwxr-xr-x  1 valy valy        16608 Jul 13  2023  p
drwxr-xr-x  4 valy staffcs      4096 May 11  2018  pi
drwxr-xr-x 17 valy staffcs      4096 Aug 11  2023  public_html
-rwxr-xr-x  1 valy valy          583 Nov 18  2019  removeoffender.sh
-rw-------  1 valy valy         3686 Oct 27  2023  r.key
-rw-r--r--  1 valy valy         2757 Oct 27  2023  rootca.pfx
```

- coloana 7 - reprezintă numele fișierului/directorului
- coloana 6 - reprezintă data ultimei modificări a fișierului respectiv
- coloana 5 - reprezintă dimensiunea (în bytes) a fișierului respectiv
- coloana 2 - reprezintă numărul de legături hard-link
- coloana 3 - reprezintă utilizatorul proprietar al acestui fișier (owner)
- coloana 4 - reprezintă grupul din care face parte acest fișier
- coloana 1 - reprezintă drepturi de acces

Recapitulare

Fișiere header

- Pe langă fișierele .c în limbajul C mai sunt definite și fișiere header .h
- sunt fișiere ce conține doar declarații de tipuri, variabile, funcții și macro-uri
- nu conțin definiții și cod – nu conțin corpul funcțiilor
- fișierele header se includ cu directive de #include
- fișierele header nu se compilează și nu se transmit la compilator – ele doar se includ
- fișierele C se transmit compilatorului – fișierele C NU SE INCLUD (este conceptual greșit !!!)
- de obicei un fișier header însoțește un fișier C dar pot să existe și mai multe fișiere header fără vreun fișier C asociat
- un program C mai complex nu se scrie într-un singur fișier C ci se separă în mai multe fișiere C și header în funcție de rol și funcționalitate
- unul sau mai multe fișiere .c cu unul sau mai multe fișier .h pot forma o bibliotecă – o colecție de funcții și tipuri de date asociate

Recapitulare

Fișiere header

- Să se implementeze o bibliotecă ce gestionează operații minimale pe numere complexe. Să se implementeze de asemenea un program principal, separat care să testeze și să folosească biblioteca de numere complexe

```c
#ifndef __COMPLEX_H
#define __COMPLEX_H

typedef struct
{
  int re;
  int im;
}COMPLEX;

COMPLEX complex_add(COMPLEX a, COMPLEX b);
COMPLEX complex_sub(COMPLEX a, COMPLEX c);

#endif
```

complex.h

```c
#include <stdio.h>
#include "complex.h"

int main(void)
{
  COMPLEX a = {1,-2};
  COMPLEX b = {10,20};
  COMPLEX t;
  t = complex_add(a,b);
  t = complex_sub(a,b);
  return 0;
}
```

complex_test.c

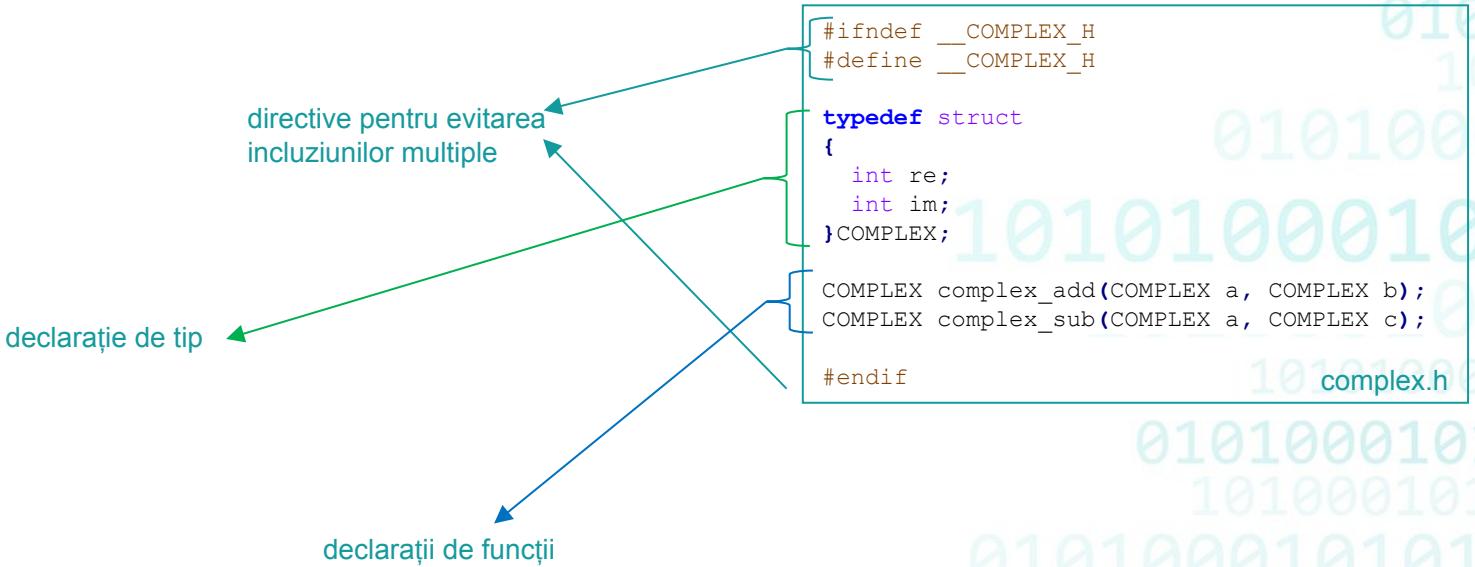
```c
#include "complex.h"

COMPLEX complex_add(COMPLEX x, COMPLEX y)
{
  COMPLEX r;
  r.re = x.re + y.re;
  r.im = x.im + y.im;
  return r;
}

COMPLEX complex_sub(COMPLEX x, COMPLEX y)
{
  COMPLEX r;
  r.re = x.re - y.re;
  r.im = x.im - y.im;
  return r;
}
```

complex.c

```
gcc -Wall -o p complex.c complex_test.c
```

Recapitulare

Fișiere header



Recapitulare

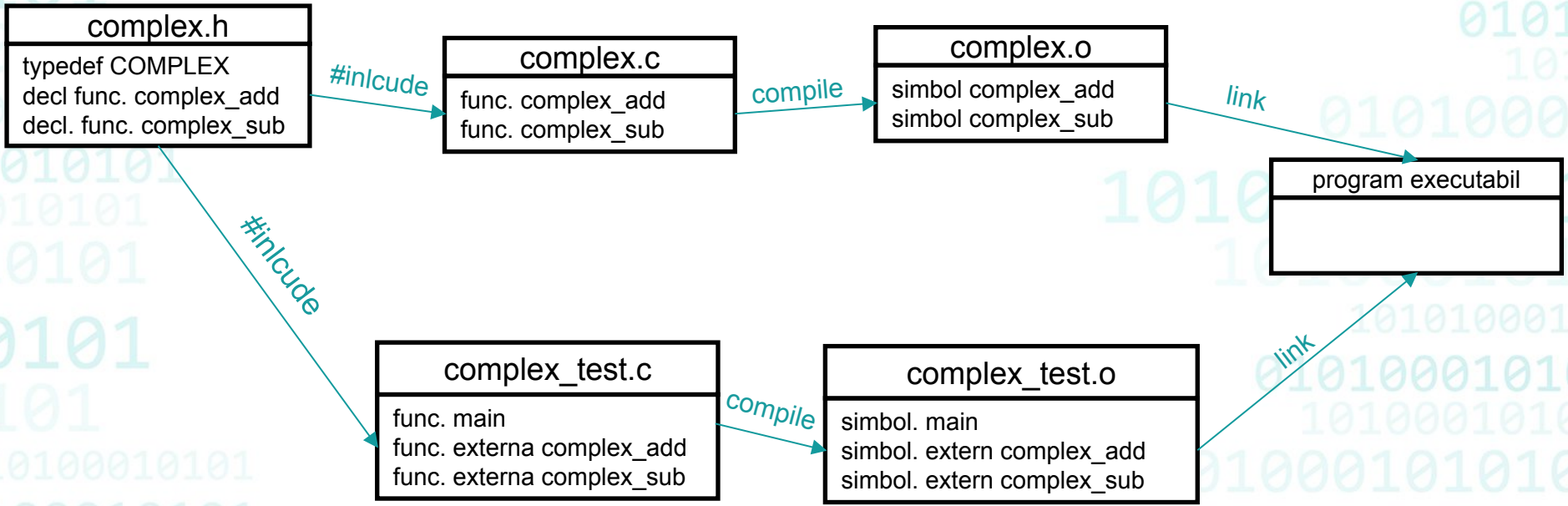
Fișiere header

- Directivele pentru evitarea incluziunilor multiple (once-only headers)
  - un fișier header poate ajunge să fie închis direct sau indirect de mai multe ori în același fișier C fiind practic parcurs de mai multe ori de către preprocesor si compilator ☾ pot rezulta erori de definiții și declarații multiple
  - directivele rezolva problema: doar prima dată va intra în corpul #indef…#endif – la a doua parcurgere va găsi ca fiind deja definit macro-ul
  - macro-urile din definiții multiple sunt realizate prin niște convenții de cod – este necesar doar să fie unice între fișiere
  - în standard-ul POSIX se recomandă să se declare folosind numele fișierului header precedat de 2 caractere underscore (__) iar în loc de punct se va pune tot un caracter underscore. ex: __COMPLEX_H
  - toate declarațiile se vor scrie în corpul #ifndef…#endif
- În fișierele header nu se definesc funcții (nu se implementează) și nu se definesc variabile

Recapitulare

Fișiere header

- Compilare exemplu precedent:



Recapitulare

Fișiere header

- Definiții de funcții și variabile în fișiere header - poate duce la erori de tipul *multiple definition*
- Definiția unei variabile este implicită cu declarație
  - declarația unei variabile ☾ specificarea tipului și a numelui variabilei
  - definiția unei variabile ☾ alocarea de memorie pentru variabilă
- Cum se poate face declarație de variabilă fără definiție?
- Soluție: modificatorul **extern** (keyword) – specifică compilatorului că acea variabilă nu se va defini în fișierul obiect curent ci ea este definită intr-un alt fișier obiect
- în complex.h *n* este doar declarat iar când se include în complex.c nu va genera eroare de tipul *multiple definition*
- daca nu se compilează și complex.c se va genera eroare de linkeditare: *unreferenced symbol*

```c
#include "complex.h"
int n;
```

complex.c

```c
#ifndef __COMPLEX_H
#define __COMPLEX_H

extern int n;

#endif
```

complex.h

```c
#include "complex.h"

int main(void)
{
  n = 1;
  return 0;
}
```

complex_test.c

Recapitulare

Fișiere header

- în cazul anterior: variabila n este definită în fișierul obiect complex.o și accesată din fișierul obiect complex_test.o
- există și posibilitatea ca variabila n să poată fi blocată la nivelul fișierului obiect din care face parte prin folosirea modificatorului static
- în acest caz, folosirea modificatorului extern va genera eroare la includerea în complex_test.c, variabila n nefiind vizibilă în afara fișierului obiect complex.o
- Wfolosirea modificatorului static în fața unei declarației unei variabile globale limitează utilizarea acestei variabile la nivelul fișierului obiect din care va face parte – se va permite link-editare în exterior

```c
#include "complex.h"
static int n;
```

complex.c

```c
#ifndef __COMPLEX_H
#define __COMPLEX_H

extern int n; // va genera eroare la link

#endif
```

complex.h

```c
#include "complex.h"

int main(void)
{
  n = 1;
  return 0;
}
```

complex_test.c

Recapitulare

Biblioteci

- Bibliotecă – un fișier binar, compilat ce conține o colecție de funcții, (similar cu un fișier executabil fără funcța main())
- Fișierul de tip bibliotecă este de obicei însoțit de unul sau mai multe fișier header pentru a oferi programatorului o interfață la codul conținut
- In Linux, există convenția ca orice bibliotecă să aibă numele precedat de prefix *lib.* (pentru o bibliotecă cu numele test, se va da numele fișierului bibliotecă ca fiind *libtest*)
- Sistemul de operare Linux caută bibliotecile în directoarele /lib, /lib32, /lib64, /usr/lib, /usr/local/lib sau în alte directoare specificate de utilizator la execuție sau la compilare
- Tipuri de biblioteci : statice, dinamice
- **Statice** – biblioteca linkediteaza în fișierul executabil și se integrează în acesta. Fișierul executabil va conține atât codul lui cât și întregul cod al bibliotecii. O modificare ulterioară a bibliotecii necesită recompilarea tuturor programelor ce o folosesc
- **Dinamice** – biblioteca se link-editeaza parțial și nu va fi inclusă în fișierul executabil. La momentul execuției sistemul de operare va realiza secvența finală de link-editare și va face astfel legătura dintre program și biblioteca dinamică folosită de acesta. dinamice sunt doar "inspectate" în procesul de link-editare și nu sunt folosite efectiv. După compilare, în procesul de rulare a programului (la "run-time") bibliotecile necesare sunt folosite. Acestea fac parte din mediul de run-time al sistemului și la momentul execuției sistemul caută bibliotecile necesare și le poate link-edita dinamic atunci când este nevoie. Dacă bibliotecile nu există la momentul execuției, programul nu este lansat și sistemul generează o eroare în care explică ce bibliotecă nu a putut fi găsită.

Recapitulare

Biblioteci

- Exemplu compilare și utilizare bibliotecă statică

```
gcc –Wall –shared –o libcomplex.so complex.c
```

s-a compilat fișierul complex.c și s-a obținut biblioteca dinamica (shared object) libcomplex.so

```
gcc –Wall –c –o main.o main.c
```

s-a compilat fișierul main.c fără linkeditoare și s-a obținut fișierul obiect

```
Gcc –Wall –o prog main.o –L . -lcomplex
```

s-a linkeditat fișierul obiect al programului principal cu biblioteca statică
-L. – precizează compilatorului că urmează să se linkediteze o bibliotecă din
directorul curent (.)
-lcomplex – precizează compilatorului să linkediteze biblioteca *complex* care se află în fișierul libcomplex – se observă exemplificarea convenției

- Execuția unui program ce conține referințe către biblioteci dinamice
- Sistemul de operare va găsi în fișierul executabil referințe către biblioteci dinamice și va încerca să le găsească pentru a l rula programul. Va căuta bibliotecile referite într-o listă de căi prestabilite: /lib, /lib32, /lib64, …
- Dacă nu găsește bibliotecile referite în lista de căi, sistemul de operare va da o eroare la execuția programului

```
valy@debiandev:~/shared/static$ ./prog
./prog: error while loading shared libraries: libcomplex.so: cannot open shared object file: No such file or directory
```

Recapitulare

Biblioteci

- Eroarea a apărut pentru că în procesul de link-editare la run-time sistemul nu a găsit în baza lui de date biblioteca libtest.so. Acest lucru se poate vedea și prin invocarea programului ldd. Acesta afișează la *stdout* toate bibliotecile dinamice referite de un program executabil, precum și biblioteca cu care se poate face link-editarea la *run-time*. În cazul nostru vom primi următorul răspuns:

```
valy@debiandev:~/shared/static$ ldd prog
linux-vdso.so.1 (0x00007ffe7a7e9000)
libcomplex.so => not found
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe407651000)
/lib64/ld-linux-x86-64.so.2 (0x00007fe407bf2000)
```

- În partea stângă a semnului "=>" se afișează numele bibliotecii referite de programul executabil, iar în partea dreaptă a semnului "=>" se afișează biblioteca cu care s-a făcut link-editarea dinamică (și adresa acestuia). Se poate observa că în cazul bibliotecii libcomplex.so nu s-a făcut link-editarea, cauza fiind lipsa fișierului. Acesta nu a fost găsit în baza de date a sistemului ce conține bibliotecile dinamice. Această bază de date se află în general în fișierul /etc/ld.so.cache, gestionată de programul ldconfig (ce trebuie executat cu drepturi de root). Pe lângă această bază de date, sistemul mai poate căuta bibliotecile dinamice și în alte locații.
- Pentru a se adăuga o nouă cale în baza de date */etc/ld.so.cache* unde sistemul de operare să caute biblioteci shared object, este necesar să se creeze un fișier un fișier nou în directorul */etc/ld.so.conf.d* . Fișierul nou va avea obligatoriu extensia
.conf și va conține câte o cale absolută pe fiecare linie. După ce s-a creat fișierul, se va executa comanda *ldconfig*. Această comandă va parcurge toate fișierele din /etc/ld.so.conf.d și va adăuga căile gasite în baza de date */etc/ld.so.cache*.
- Baza de date */etc/ld.so.cache* se poate interoga apeland comanda anterioară cu argumentul –p: *ldconfig –p*. Astfel se va printa la ieșirea standard lista tuturor directoarelor în care sistemul de operare va căuta biblioteci dinamice la momentul execuției unui program – *pentru a executa comanda ldconfig sunt necesare drepturi de root*

Recapitulare

Biblioteci

- O alta metodă de adăuga temporar o cale unde sistemul de operare să caute biblioteci dinamice este setarea cu acea nouă cale în variabila de mediu LD_LIBRARY_PATH astfel:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/valy/shared/static
```

- După setarea acestei variabile de mediu cu calea directorului în care se află biblioteca, comanda ldd va da următorul răspuns:

```
valy@debiandev:~/shared/static$ ldd prog
linux-vdso.so.1 (0x00007ffe75568000)
libcomplex.so (0x00007f933d797000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f933d3f8000)
/lib64/ld-linux-x86-64.so.2 (0x00007f933db9b000)
```

Recapitulare

Organizare sistem de fisiere Linux

- Sistemul de fisiere Linux: ext4 (evoluat din ext2, ext3)
- Organizare teoretică
  - Boot
  - Superblock – metadata – definește structurile sistemului de fișiere și adresele unde acestea sunt scrise pe disk
  - Tabela inodes – fiecare inode conține informații despre un fișier (nume, permisiuni, cale, tip, și adresele zonelor de date)
  - DATA – conține efectiv datele stocate pe disc
- Tipuri de fișiere:
  - **regular file (–)** – fișier obișnuit – stocare de date
  - **directory (d)** – director (folder, catalog …) – identificat prin nume, nu poate fi scris de utilizator
  - Link file (l) – fișier legătură
    - Symbolic link (legătură simbolică) – este un fișier ce conține un “pointer”, o referință textuală (calea) către un alt fișier
    - Hard link – un fișier cu un inode separat care referă datele de la un inode existent (fișier existent) – similar cum s-ar copia un fișier dar fără a i se duplica datele
  - **Block file (b)** – fișier special care în mod uzual identifică un dispozitiv hardware (harddrive, monitor, printer, mouse…) Fiind block file specifică faptul că un transfer de date către dispozitivul respectiv se va face la nivel de block de date
  - **Character file (c)** – fișier special care în mod uzual identifică un dispozitiv hardware. Fiind character file – implică faptul că transferul se va face la nivel de byte
  - **Socket file (s)** – un fișier de tip socket (endpoint ce permite comunicație între procese) – UNIX socket. Poate fi și netwrok socket unde endpoint-ul de comunicație este format dintr-un port și o adresa IP – scrierea și ctirea dintr-un network socket implică utilizarea apelurilor sistem dedicate `sendmsg()` și `recvmsg()`.
  - **Named pipe file (p)** – fișier de tip pipe – funcționează ca un pipe anonim format cu apelul sistem pipe() cu diferența că acesta are un nume și poate fi folosit de orice prioces nefiind limitat la accesul către descriptorii acestuia. Funcționeaza după politică FIFO

Recapitulare

Organizare sistem de fisiere Linux

- Aflarea tipului unui fișier: comanda *file* - manual: *man file*
- Crearea unei legături simbolice: comanda *ln* – manual: *man ln*
- Obținerea i-node-ului unui fișier: comanda *stat* – manual: *man stat*
- Crearea unui named pipe: comanda *mkfifo* – manual: *man mkfifo*
- Informații despre un sistem de fișiere de poate obține folosind comanda: *dumpe2fs*
  ex: `dumpe2fs /dev/sda1`

IN LIMBAJUL C – folosind apeluri sistem

- Obținerea i-node-ului unui fișier: *int stat(const char *pathname, struct stat *statbuf);* manual: *man 2 stat*
- Obținerea i-nodu-ului unui fișier de tip legătură simbolică:

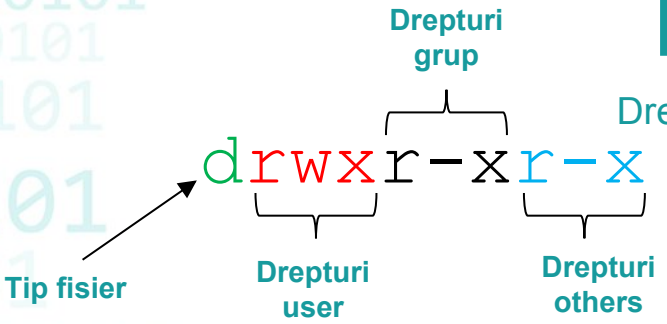
*int lstat(const char *pathname, struct stat *statbuf);* manual: man 2 stat

- Crearea unui named pipe: *int mkfifo(const char *pathname, mode_t mode)*; manual *man 2 mkfifo*
- Redenumirea unui fișier: *int rename(const char *oldpath, const char *newpath);* manual: *man rename*
- Gestionarea legăturilor:

```
int link(const char *oldpath, const char *newpath);    // creeaza legaturi fixe spre fisiere
int symlink(const char *oldpath, const char *newpath); //  creeaza legaturi simbolice spre fisiere sau directoare
int unlink(const char *pathname);                      // sterge o intrare in director (legatura, fisier sau director)
```

Recapitulare

Drepturi de acces – comanda chmod



- Comanda *chmod* poate fi folosită pentru a schimba drepturi de acces ale unui fișier
- Sintaxa 1: **`chmod [OPTION]... MODE[,MODE]...FILE...`**
- FILE – calea către fișierul căruia i se schimba drepturile
- MODE – o combinație (listă de combinații) de litere ce vor adăuga sau elimina anumite drepturi astfel (modul simbolic)
- Semnele +/- – vor adăuga (+) sau elimina un drept
- Litera u/g/o/a – schimbarea drepturilor se va referi la u – user, g – group, o – other, a – all
- Literele r/w/x – vor schimba drepturile astfel: r – read, w – write, x – execute
- Exemplul 1: chmod +ugwx /home/user/file.txt
  - Va adăuga (+) drepturi de scriere si executie (w,x) la user și grup (u,g)
- Exemplul 2: chmod –ar /home/user/file.txt
  - Va elimina (-) dreptul de citire (r) pentru toți utilizatorii

- Sintaxa 2: **`chmod [OPTION]... OCTAL MODE ... FILE...`**
- FILE – calea către fișierul căruia i se schimba drepturile de acces (modul octal)
- OCTAL MODE – 3 digiți în octal (0-7) ce va reprezenta combinația pe a drepturilor ce vor fi setate pentru user (digit 2), group (digit 1) și others (digit 0)
- Exemplul 1: chmod 040 /home/user/file.txt – drepturile: - - - r - - - - - nici un drept in afara de cel de citire pentru group
- Exemplul 2: chmod 741 /home/user/file.txt – drepturile: r w x r - - - x drepturi depline (rwx) pentru user, doar drept de citire pentru grup și doar drept de execuție pentru others

Recapitulare

Pagina de manual

- Paginile de manual se apeleaza folosind comanda man
- Se folosește comanda man
  - Sintaxa: `man [optiuni] [seciune] comanda`
- Secțiuni pagini de manual:
  1. Secțiunea 1 - descrie comenzile standard (programe executabile și comenzi shell script)
  2. Secțiunea 2 - apeluri sistem UNIX apelabile în limbajul C
  3. Secțiunea 3 - funcțiile de bibliotecă C
  4. Secțiunea 4 - informații despre fișierele speciale (în principal cele din /dev)
  5. Secțiunea 5 - informații despre convențiile și formatele anumitor fișiere specifice sistemului
  6. Secțiunea 6 - manuale de la jocurile din Linux
  7. Secțiunea 7 - informații despre diverse teme ce nu pot fi incluse în alte secțiuni (spre exemplu man 7 signal)
  8. Secțiunea 8 - comenzi de administrare a sistemului (de obicei doar pentru userul root)
  9. Secțiunea 9 - rutine kernel
- Daca nu se specifică secțiunea, se caută comanda în ordin crescător al secțiunilor și se returnează prima apariție

Recapitulare

Pagina de manual

- Structura unei pagini de manual (nu toate elementele sunt obligatorii – pot lipsi din anumite pagini de manual)
- NAME – numele paginii / comenzii / apelului….
- LIBRARY – în ce biblioteca se găsește
- SYNOPSIS – sintaxa
- DESCRIPTION – descrierea detaliata comenzii / apelului
- RETURN VALUE – valoarea returnată de comandă / funcție
- ERRORS – posibilele erori semnalate – de obicei se trec erorile setate în variabila errno
- VERSIONS – informații despre versiunile comenzii
- STANDARDS – standardele în care comanda / funcția apare
- NOTES – informații adiționale
- BUGS – probleme semnalate la versiunea curenta
- SEE ALSO – comenzi / apeluri pe același subiect – se dă comanda și apoi în paranteză secțiunea:
  ex: chown(2) – comanda chown din sectiunea 2 ☾ se va apela man 2 chown
- EXAMPLES – exemplu de utilizare a funcției / apelului

Recapitulare

Apeluri sistem

- Apelul sistem chmod() `int chmod(const char *pathname, mode_t mode);`
  - Are rolul de a schimba drepturile de acces pentru fișierul specificat prin pathname cu drepturile specificare prin mode
  - Modul de utilizare al acestui apel sistem este similar cu utilizarea comenzii chmod
  - Primește ca argument 1 o cale absolută sau relativă a fișierului căruia i se vor schimba drepturile de acces
  - Primește ca și argument 2 o valoare în octal cu noile drepturi (similar ca si la comanda chmod). Sunt definite o serie a macro-uri cu măștile corespunzătoare pentru fiecare tip de permisiune. Se pot folosi fie valori absolute în octal fie o combinație a acestor macro-uri (prin operația bitwise-or). Macro-urile sunt descrise in pagina de manual
  - Pagina de manual: `man 2 chmod`
  - Studiați pagina de manual

Recapitulare

Apeluri sistem

- Apelul sistem open()

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

  - Are rolul de a deschide un fișier
  - Pagina de manual: `man 2 open`
  - Returnează un număr ce reprezintă descriptorul fișierului dechis sau -1 în caz de eroare cu setarea valorii *errno*
  - Descriptorul va fi apoi folosit pentru a identifica în mod unic fișierul deschis. Se va folosi ca și parametru la alte apeluri sistem cum ar fi: read(), write(), close()
  - Primește ca argument o cale absolută sau relativă a fișierului pe care să îl deschidă
  - Parametrul flags specifică modul în care să fie deschis fișierul. Se folosește printr-o combinație (prin bitwise-or) a unor MACRO-uri cum ar fi (cele mai utilizate): O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_TRUNC
    - O_APPEND – dacă fișierul există se va deschide cu setarea cursorului la sfârșitul acestuia (în caz de scriere se va adăuga la sfârșit)
    - O_TRUNC – dacă fișierul există conținutul lui se va șterge
    - O_CREAT – dacă fișierul nu există atunci acesta va fi creat este necesar să se specifice drepturile de acces prin parametrul mode ce reprezintă drepturile de acces în format octal pe biți (similar cu chmod) sau MACRO-uri cu măști cu drepturi de acces (S_IRWXU, S_IRUSR, S_IWUSR, ….) – descrise în pagina de manual
  - În cazul în care fișierul nu există și nu se adaugă flag-ul O_CREAT, funcția open() nu creează noul fișier și returnează eroare
  - În caz de eroare va seta variabila *errno* (erorile sunt descrise în pagina de manual la secțiunea ERRORS)

Recapitulare

Apeluri sistem

- Apelul sistem open()

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

  - Este absolut necesar a unul din flag-urile O_RDONLY, O_WRONLY, O_RDWR sa fie prezent (altfel funcția open() nu știe cum să deschidă fișierul)
  - În cazul în care fișierul nu există și nu se adaugă flag-ul O_CREAT, funcția open() nu creează noul fișier și returnează eroare
  - În caz de eroare va seta variabila *errno* (erorile sunt descrise în pagina de manual la secțiunea ERRORS)
  - Exemplu de combinare a flag-urile (folosind operația bitwise – OR): O_RDONY | O_APPEND | O_CREAT

Recapitulare

Apeluri sistem

- Apelul sistem close()

```
int close(int fd);
```

  - Are rolul de a închide un fișier deschis în prealabil fie cu apelul sistem open() fie cu alte apeluri sistem
  - Returnează 0 la succes sau -1 în caz de eroare cu setarea valorii *errno*
  - Primește ca argument un întreg ce reprezintă descriptorul unui fișier deschis (fișierul poate fi orice tip de fișier: normal, pipe, socket, … etc)
  - În caz de eroare va seta variabila *errno* (erorile sunt descrise în pagina de manual la secțiunea ERRORS)
  - Pagina de manual: `man 2 close`

Recapitulare

Apeluri sistem

- Apelul sistem read()

```
int read(int fd, void *buff, size_t count);
```

  - Are rolul de a citi dintr-un un fișier (orice tip de fișier – regular, pipe, socket, … etc) deschis în prealabil
  - Funcția va citi din fișierul referit de descriptorul `fd` maxim `count` bytes și îi va scrie în zona de memorie referită de `buff`
  - Returnează numărul de bytes citiți sau -1 în caz de eroare
  - În caz de eroare va seta variabila *errno* (erorile sunt descrise în pagina de manual la secțiunea ERRORS)
  - Pagina de manual: `man 2 read`

- Apelul sistem write()

```
int write(int fd, const void *buff, size_t count);
```

  - Are rolul de a scrie intr-un un fișier (orice tip de fișier – regular, pipe, socket, … etc) deschis în prealabil
  - Funcția va scrie din fișierul referit de descriptorul `fd` maxim `count` bytes din zona de memorie referită de `buff`
  - Returnează numărul de bytes ce au fost scriși sau -1 în caz de eroare
  - În caz de eroare va seta variabila *errno* (erorile sunt descrise în pagina de manual la secțiunea ERRORS)
  - Pagina de manual: `man 2 write`

Recapitulare

Apeluri sistem

- Apelul sistem stat()

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

  - Are rolul de a obține i-node-ul unui fișier dat
  - Pagina de manual: `man 2 open`
  - Returnează 0 pentru succes sau -1 în caz de eroare
  - Parametrul *pathname* reprezinta calea fișierului căruia i se va obține i-node-ul
  - *statbuf* – reprezintă un parametru de ieșire - funcția va scrie în această zonă informațiile inode-ului cerut
  - În caz de eroare va seta variabila *errno* (erorile sunt descrise în pagina de manual la secțiunea ERRORS)
  - În cazul în care se apelează pe o legătură simbolică, funcția stat() va returna i-node-ul fișierului target . Pentru obținerea i-node-ului fișierului legătură simbolică se poate folosi lstat()
  - Funcția fstat() se poate folosi pentru un fișier deja deschis

Recapitulare

Apeluri sistem

- Apelul sistem stat()
  - Informațiile din structura *struct stat*
  - Pagina de manual pentru structura *struct stat*
    `man 7 inode`

```
struct stat {
               dev_t     st_dev;         /* ID of device containing file */
               ino_t     st_ino;         /* inode number */
               mode_t    st_mode;        /* file type and mode */
               nlink_t   st_nlink;       /* number of hard links */
               uid_t     st_uid;         /* user ID of owner */
               gid_t     st_gid;         /* group ID of owner */
               dev_t     st_rdev;        /* device ID (if special file) */
               off_t     st_size;        /* total size, in bytes */
               blksize_t st_blksize;     /* blocksize for filesystem I/O */
               blkcnt_t  st_blocks;      /* number of 512B blocks allocated
*/

               /* Since Linux 2.6, the kernel supports nanosecond
                  precision for the following timestamp fields.
                  For the details before Linux 2.6, see NOTES. */

               struct timespec st_atim;  /* time of last access */
               struct timespec st_mtim;  /* time of last modification */
               struct timespec st_ctim;  /* time of last status change */

   #define st_atime st_atim.tv_sec      /* Backward compatibility */
   #define st_mtime st_mtim.tv_sec
   #define st_ctime st_ctim.tv_sec
};
```

Recapitulare

Apeluri sistem

- Apelul opendir()

```
DIR *opendir(const char *name);
```

  - Are rolul de a deschide un director
  - Pagina de manual: `man 3 opendir`
  - Returnează un pointer către un tip de date de tip structura (DIR) – membrii structurii sunt ascunși – ce reprezintă un flux (stream) de directoare poziționat la prima intrare din directorul dat ca și argument prin calea sa
  - Returnează NULL în caz de eroare și va seta variabila *errno* (erorile sunt descrise în pagina de manual la secțiunea ERRORS)

- Apelul closedir()

```
int closedir(DIR *dirp);
```

  - Are rolul de a închide un director deschis în prealabil cu *opendir()*
  - Pagina de manual: `man 3 closedir`
  - Returnează 0 pentru succes și -1 în caz de eroare și va seta variabila *errno* (erorile sunt descrise în pagina de manual la secțiunea ERRORS)

Recapitulare

Apeluri sistem

- Apelul readdir() `struct dirent *readdir(DIR *dirp);`
  - Are rolul de a itera o structură de directoare. La fiecare apel va returna un pointer la o structură *struct dirent* ce reprezintă următoarea intrare (practic următorul fișier) din stream-ul referit de dirp dat ca și parametru și deschis in prealabil cu *opendir()*
  - Funcția returnează NULL când a ajung la sfârșitul stream-ului referit de dirp sau în caz de eroare cu setarea valorii *errno*
  - Pagina de manual: `man 3 readdir`
  - Apelul funcționează doar pentru intrările directe din director, de pe primul nivel următor și nu recursiv
  - Structura *struct dirent*:
    - Câmpul d_type – identifică tipul intrării astfel
      - DT_BLK – block device file
      - DT_CHR – character device file
      - DT_DIR – director
      - DT_FIFO – named pipe
      - DT_LNK – legătură simbolică
      - DT_REG – fișier obișnuit (regular)
      - DT_SOCK – fișier UNIX socket
      - DT_UNKNOWN – tip fișier neidentificat

```
struct dirent {
    ino_t          d_ino;       /* Inode number */
    off_t          d_off;       /* Not an offset; see below */
    unsigned short d_reclen;    /* Length of this record */
    unsigned char  d_type;      /* Type of file;
                                   not supported by all filesystem types */
    char           d_name[256]; /* Null-terminated filename */
};
```

Recapitulare

Alte apeluri sistem

Apelul sistem unlink(): `int unlink(const char *pathname);`

- Pagina de manual: `man 2 unlink`
- Dacă calea dată ca și argument reprezintă o legătura simbolică o va șterge pe aceasta
- Dacă calea dată ca și argument reprezintă ultima legătură către un fișier atunci va șterge acel fișier
- Va efectua operația de ștergere doar dacă fișierul respectiv nu este deschis de vreun proces
- Returnează NULL în caz de eroare și va seta variabila *errno* (erorile sunt descrise în pagina de manual la secțiunea ERRORS)

Apelul sistem rmdir() – 2 – șterge un director dacă acesta este gol

Apelul sistem rename() – 2 – redenumește un fișier

Funcția de bibliotecă remove() – 3 – șterge un fișier sau director – pentru fișier apelează unlink() iar pentru director rmdir

Apelul sistem symlink() – 2 – creează o legătura simbolică

Recapitulare

Apeluri sistem

- Problema: Să se scrie un program care primește ca și argument în linie de comandă o cale relativă sau absolută de director și o cale către un fișier și are următoarele funcționalități
  - Se presupune că directorul dat ca și argument este o structura de directoare ce conține directoare, fișiere text, legături simbolice și named pipes
  - Parcurge recursiv directorul dat ca și parametru
  - Pentru fiecare fișier întâlnit programul va număra literele mici din fișier
  - Programul va scrie rezultatul într-un fișier text dat ca și argument în linie de comandă astfel: pentru fiecare intrare din director va scrie o linie de forma:

    <CALE_DIRECTOR> <DIMENSIUNE> <LITERE_MICI> <TIP_FISIER>

    Unde tip fisier poate fi: REG (fisier obisnuit), LNK (legatura simbolica), DIR (director), PIPE (named pipe)

  - Dacă primul argument nu este director programul va da o eroare si va returna un cod de eroare diferit
  - Dacă primul argument reprezintă un fișier ce există programul îl va suprascrie

Secțiunea II

Gestionarea proceselor

Gestionarea proceselor

Procese

- Program – secvență de instrucțiuni stocate ca un flux de codificări binare ce implementează anumite operații de prelucrare a unor date de intrare
  - Este de fapt doar stocat pe disc într-un fișier fără să facă nimic activ
- Procesul – o abstractizare a unui program aflat în execuție
  - Prin intermediul unui proces programul ajunge să fie executat de către procesor (CPU)
  - Fiecare proces primește o cuantă de timp de CPU (este executat de procesor pentru o anumite perioadă de timp)
  - Pe baza unui algoritm, sistemul de operare asignează câte o cuantă finită de CPU fiecărui proces – un program nu este executat de CPU în mod continuu ci este întrerupt (programul “nu știe” acest lucru)
  - Oferă posibilitatea de a avea pseudo-paralelism și de a rula operații concurente chiar și atunci când exista un singur procesor
- Procesul – o instanță a unui program și cuprinde următoarele componente (și le izolează)
  - Codul programului
  - Valoarea curenta a PC (program counter)
  - Adresele zonelor de memorie ale programului (variabile, stiva)
  - Valorile regiștrilor procesorului
  - Descriptorii de fișiere

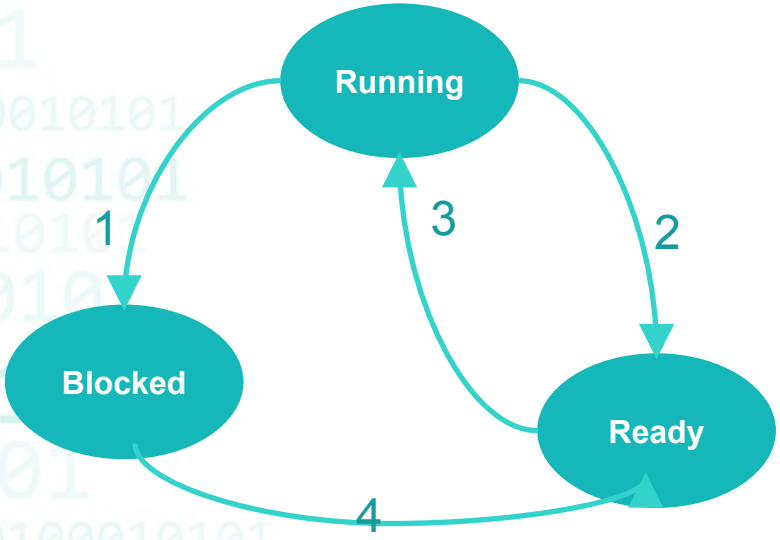
Gestionarea proceselor

Procese

- Când procesul este în execuție programul are acces la toate componentele procesului
  - Procesul rulează codul programului pentru o anumită perioadă de timp (o cuantă CPU) – asignată pe baza unui algoritm de către o componentă a sistemului de operare: planificatorul de task-uri (scheduler)
- Când procesul este suspendat din execuție:
  - Programul “nu știe” – procedura de suspendare din execuție și suspendarea în sine este transparenta pentru program
  - Se salvează întregul context al procesului (toate componentele)
- Când procesul este reluat în execuție
  - I se restaurează întregul context
  - Programul își reia execuția fără să știe că a fost întrerupta
- Procesele primesc o cuanta CPU în mod secvențial prin decizia scheduler-ului
  - è Procese secvențiale (sequential processes) sau simplu procese
- Schimbarea CPU-ului de la un proces la altul ☾ multiprogramming
- Nu se poate prezice într-un mod determinist ordinea execuției proceselor, cuanta de timp alocată, momentul intrării în execuție și momentul suspendării din execuție

Gestionarea proceselor

Stările unui proces

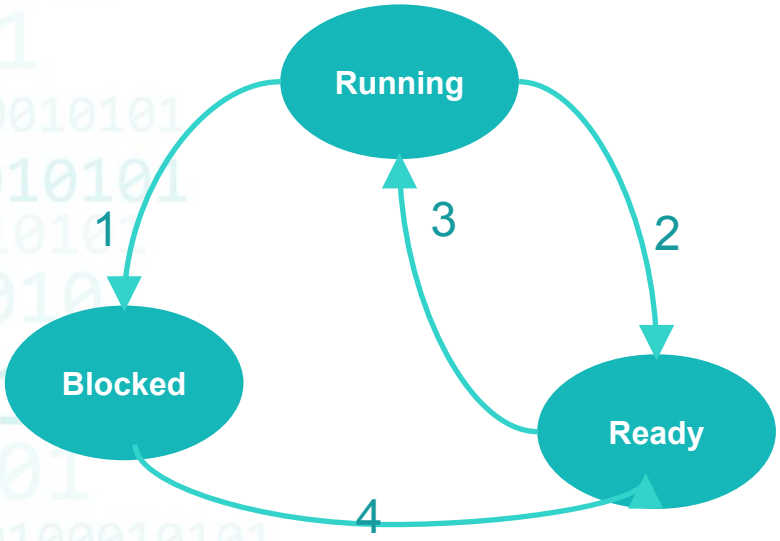


- Running (în execuție) – procesul se află în execuție și folosește CPU în acel moment de timp
- Ready (pregătit pentru execuție) – procesul este în acel moment suspendat din execuție pentru a permite execuția altui proces
- Blocked (blocat) – procesul nu este în execuție și nici nu va putea fi pus în execuție până când condiția de blocare nu va fi eliminată

  Condiții de blocare:
  - Se așteaptă un periferic
  - Se așteaptă după anumite operații de intrare/ieșire
  - Se așteaptă eliberarea unei condiții de conflict
  - Se așteaptă primirea datelor de intrare

Gestionarea proceselor

Stările unui proces – tranziții



1. Procesul trece în starea de blocked din starea de execuție. În timpul execuției procesul a intrat într-o stare de așteptare de date de intrare / evenimente / operații de intrare/ieșire și astfel el este suspendat până la rezolvarea situației de așteptare pentru a nu ține CPU ocupat cu operații inutile de așteptare
2. Execuția procesului este suspendată în principal pentru a permite execuția unui alt proces. Se salvează contextul de execuție al procesului și va trece în starea ready în așteptarea de a reveni în execuție
3. Procesul intră în stadiul de execuție. Scheduler-ul a alocat o cuantă CPU pentru procesul aflat în așteptare și sistemul de operare îi reia execuția
4. Procesul iese din starea de blocked și ajunge în starea ready prin care este marcat ca fiind pregătit pentru execuție atunci când procesul este disponibil. Starea precedentă de blocare pentru așteptarea de date / evenimente / operații de intrare-ieșire s-a terminat în urma satisfacerii condiției de așteptare.

Gestionarea proceselor

Procese

- Componentele unui process
  - cod – o secvență binară de instrucțiuni
  - Segment de memorie de date
  - Stiva
  - Program counter
  - Descriptori fișiere
  - **Thread – fir de execuție – codul programului aflat în execuție**
- Thread – unitatea elementară de execuție – thread-ul “primește” cuantă de timp CPU
- Scheduler (planificator de task-uri – thread-uri) – o componentă software a unui sistem de operare care pe baza unor algoritmi de planificare atribuie cuante de timp CPU pentru firele de execuție
- Algoritmi de planificare – politici de scheduling – algoritmi care stabilesc ordinea și durata de execuție a unei cuante atribuite firelor de execuție

Gestionarea proceselor

Procese

- Constrângeri de timp – luate în calcul de către planificator pentru a satisface nevoile de execuție în timp ale task-urilor
  - Release time – momentul de timp când un task devine disponibil pentru execuție
  - Deadline – momentul de timp până când un task trebuie să-și termine execuție
  - Response time – intervalul de timp definit dintre momentul apariției instanței unui task și momentul când execuția acestuia este încheiată și s-a furnizat un rezultat
  - Worst Case Execution Time (WCET) – cel mai lung timp posibil de execuție a unui task în cazul cel mai defavorabil reprezintă timpul maxim absolut de execuție a unui task
- Sisteme timp real – acele sisteme în care timpul este un parametru esențial
  în aceste sisteme nu doar rezultatul operației este important ci și timpul în care acesta a fost obținut
  - *” those in which the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced*” [1]
  - „*any type of system where the response time is an essential parameter*” [2]

[1] Daintith, John "Real-time system." A Dictionary of Computing. 2004.Encyclopedia.com.
[2] John A. Stankovic, Krithi Ramamritham, Marco Spuri, Giorgio C. Buttazzo, “Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms”, Kluwer Academic Publishers Norwell, MA, USA ©1998, ISBN:079238

Gestionarea proceselor

Procese

- Tipuri de politici de planificare – în funcție de constrângerilr de timp:
  - hard real-time (HRT) – aplicațiile au condiții stricte de timp și se consideră că dacă aplicația și-a depășit timpul alocat (deadline-ul) atunci consecințele asupra sistemului sunt catastrofale.
  - Firm real-time (FRT) – la aplicațiile din aceasta categorie depășirea unui deadline poate fi tolerata dar depășirile repetate ale deadline-ului pot duce la nefuncționarea totală a sistemului
  - Soft real-time (SRT) - la aplicațiile din aceasta categorie se consideră că depășirea unui deadline poate afecta doar performanta sistemului fără sa-i afecteze funcționalitatea de bază
  - non real-time: la această categorie de aplicații nu există nici un fel de constrângeri de timp
- Exemple de sisteme de timp real
  - HRT – controlul procesor tehnologice într-un reactor nuclear, sistemul de frânare, sistem de detecție, alertare și stingere incendii
  - FRT – sistem de control acces, sampling audio
  - SRT – streaming audio-video
  - Non real-time: hello world !

Gestionarea proceselor

Procese

- Tipuri de politici de planificare în sisteme Linux (POSIX):
  - SCHED_OTHER (SCHED_NORMAL) - politica implicita de planificare a thread-urilor in Linux. Thread-urile vor avea în acest caz toate prioritatea 0. Thread-urile vor fi alese pentru a fi executate dintr-o listă, având fiecare drept egal de execuție. Există totuși un mecanism pentru a favoriza anumite thread-uri, reprezentat printr-o prioritate dinamica numita nice-level. Aceasta prioritate dinamică se aplică doar pentru prioritatea statica 0
  - SCHED_BATCH - similar ca si SCHED_NORMAL, dar în plus planificatorul va considera în algoritmul de planificare aspectul că un astfel de thread poate folosi procesorul în mod intensiv și aplică anumite optimizări
  - SCHED_IDLE - politica de planificare folosită pentru thread-uri cu o prioritate extrem de mică
  - SCHED_RR, SCHED_FIFO - aceste 2 politici de planificare sunt considerate soft real-time si se folosesc cu prioritate statică mai mare decât 0
  - SCHED_DEADLINE – această politică de planificare a fost introdusa în nucleul de Linux începând cu versiunea 3.14 în urma unui proiect desfășurat la Universitatea Scuola Superiore Sant'Anna din Pisa și inclus in nucleul de Linux in 2014. Este o politica de scheduling hard real-time bazata pe algoritmul de planificare EDF (Earliest Deadline First). Prin aceasta politica de planificare sistemele Linux devin astfel utilizabile și in domenii cu constrângeri stricte de timp
- Mai multe informații se pot obține prin: *man 7 sched*

Gestionarea proceselor

Crearea proceselor (Linux)

Apelul sistem `fork()`

- Procesul apelant (procesul părinte – parent) creează un nod proces (fiu – child)
- Procesul fiu va fi o copie fidelă și exactă a procesului părinte cu contextul imediat înaintea apelului fork()
- Procesul fiu va fi independent de procesul părinte și va avea propria lui zonă de memorie, propriu cod, propriul *program counter (PC)*, proprii descriptori toate cu conținutul copiat de la procesul părinte
- Pagina de manual: `man 2 fork`

Apelul fork() – valori de return:

-1 – în caz de eroare cu setarea valorii *errno*

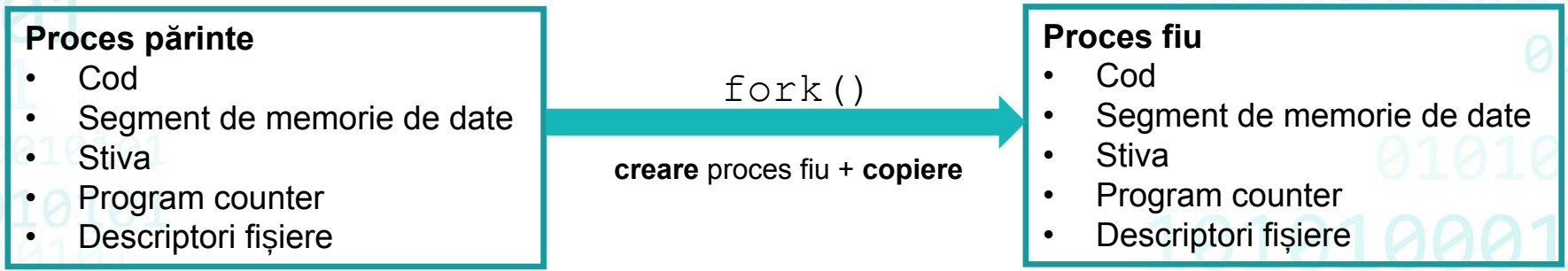
0 – în caz de succes în codul procesului fiu

PID – în codul părintelui, unde pid reprezintă identificatorul de proces unic al procesului fiu creat

```
if( ( pid=fork() ) < 0)
  {
    perror("Eroare");
    exit(1);
  }
if(pid==0)
  {
    /* codul fiului - doar codul fiului poate ajunge aici (doar in fiu fork() returneaza 0)*/
    ...
    exit(0); // apel necesar pentru a se opri codul fiului astfel incat acesta sa nu execute si codul parintelui
  }
/* codul parintelui */
```

Gestionarea proceselor

Crearea proceselor (Linux)



Apelul sistem `fork()`

- Procesul apelant (procesul părinte – parent) creează un nod proces (fiu – child)
- Procesul fiu va fi o copie fidelă și exactă a procesului părinte cu contextul imediat înaintea apelului fork()
- Procesul fiu va fi independent de procesul părinte și va avea propria lui zonă de memorie, propriu cod, propriul *program counter (PC)*, proprii descriptori toate cu conținutul copiat de la procesul părinte

Apelul fork() – valori de return:

-1 – în caz de eroare cu setarea valorii *errno*

0 – în caz de succes în codul procesului fiu

PID – în codul părintelui, unde pid reprezintă identificatorul de proces unic al procesului fiu creat

Gestionarea proceselor

Crearea proceselor (Linux)

1. Procesul părinte își rulează codul și ajunge la apelul fork()
2. Se suspendă execuția procesului părinte
3. Se creează procesul fiu (se alocă segmente de memorie diferite față de cele ale procesului părinte – pentru cod, date, stivă)
4. Se copiază tot conținutul procesului părinte în procesul fiu (memorie, cod, program counter, descriptori…) – procesul fiu devine o copie exactă a procesului părinte
5. Se setează program counter-ul procesului fiu cu aceeași valoare ca și cel al părintelui
6. Se lansează în execuție cele 2 procese (părinte și fiu)
7. Cele două procese vor rula în mod independent fiind izolate unul față de celălalt

Gestionarea proceselor

Crearea proceselor (Linux)

- PID
  - identificator unic pentru fiecare proces – un număr întreg
  - Valorile proceselor terminate se refolosesc
  - Valoarea maxima a PID-ului este: /proc/sys/kernel/pid_max
- Erori posibile fork()
  - Memorie insuficientă
  - Nu mai sunt PID-uri disponibile
  - S-a ajuns la limita maximă de thread-uri ce se pot crea în sistem: /proc/sys/kernel/threads-max

Gestionarea proceselor

Terminarea proceselor (Linux)

1. Terminare normală voluntară – procesul se termină în urma unui apel al apelului sistem `exit()`
   - Terminare fără eroare – exit(0)
   - Terminare cu eroare – exit(n), unde n != 0
   - Un program single-process se va termina după apelul de return din funcția main() – după return de va continua execuția unui cod adițional introdus de compilator ce va apela în final apelul sistem exit()
2. Terminare involuntară în urma unei erori critice (fatal error)
   - Procesul va fi oprit de către sistemul de operare prin trimiterea unui semnal în cazul unor erori critice precum: illegal instruction, segmentation fault, division by zero, … etc
3. Terminare involuntară de către alt proces
   - Procesul va fi oprit prin recepționarea unor semnale de la alte procese – semnale ce pot duce inevitabil la terminarea procesului

- În momentul în care procesul își termină execuția el nu este complet șters de către kernel ci este trecut într-o stare de proces terminat sau *zombie* – procesul este menținut în tabela de procese din sistem și se așteaptă ca procesul părinte să-i citească starea cu care acesta s-a terminat – prin apelurile sistem *wait* sau *waitpid*
- În starea *zombie*, sistemul de operare nu face nici o modificare asupra procesului și toate componentele sale rămân alocate (inclusiv PID-ul rămâne rezervat).

Gestionarea proceselor

Terminarea proceselor (Linux)

Apelul sistem wait(): `pid_t wait(int *status);`

- Apelat de către un proces părinte pentru a aștepta terminarea unui proces fiu (orice) pentru a prelua starea acestuia
- Primește ca argument un pointer către o zona de memorie de tip întreg unde va scrie starea procesului ce s-a terminat
- Returnează PID-ul procesului fiu ce s-a terminat sau -1 în caz de eroare cu setarea *errno*
- În cazul în care nu mai sunt procese fiu a căror terminare să o aștepte procesul părinte apelant, wait() va returna -1 și va seta *errno* cu valoarea ECHILD – aceasta nu reprezintă neapărat o eroare !!
- Valoarea scrisă în parametrul status poate fi interpretată folosind niște MACRO-uri puse la dispoziție de biblioteca de apeluri sistem, cum ar fi: WIFEXITED, WEXITSTATUS, WIFSIGNALED, …
- Pagina de manual: `man 2 wait`

Gestionarea proceselor

Terminarea proceselor (Linux)

Apelul sistem waitpid(): `pid_t waitpid(pid_t pid, int *status, int flags);`

- Similar cu apelul wait() dar mult mai general
- Apelat de către un proces părinte pentru a aștepta terminarea unui proces fiu, identificat prin parametrul pid, pentru a prelua starea acestuia
- În cazul în care pid are valoarea -1 se va aștepta după orice proces fiu al procesului părinte apelant (exact ca și la wait())
- Primește ca argument un pointer către o zona de memorie de tip întreg unde va scrie starea procesului ce s-a terminat
- Returnează PID-ul procesului fiu ce s-a terminat sau -1 în caz de eroare cu setarea errno
- În cazul în care nu mai sunt procese fiu a căror terminare să o aștepte procesul părinte apelant, wait() va returna -1 și va seta errno cu valoarea ECHILD – aceasta nu reprezintă neapărat o eroare !!
- Valoarea scrisă în parametrul status poate fi interpretată folosind niște MACRO-uri puse la dispoziție de biblioteca de apeluri sistem, cum ar fi: WIFEXITED, WEXITSTATUS, WIFSIGNALED, …
- Parametrul flags poate să fie 0 sau o combinație binară a flagurilor: WNOHANG, WUNTRACED, WCONTINUED
- Pagina de manual: `man 2 waitpid`
- Apelul `waitpid(-1, &wstatus, 0)` este echivalent cu `wait(&wstatus)` Ⓟ *wait* devine caz particular de *waitpid*

Gestionarea proceselor

Cod exemplu

COD EXEMPLU:

Să se scrie un program care lansează în execuție 5 procese fiu. Fiecare proces va printa la ieșirea standard 20 de linii de text ce vor conține PID-ul procesului și numărul linie după care fiecare proces fiu se va termina returnând o valoare diferită. Procesul părinte va printa și el la ieșirea standard 20 de linii cu același conținut după care va aștepta terminarea tuturor proceselor fiu create. La terminarea fiecărui proces fiu procesul părinte va printa faptul că procesul fiu respectiv s-a terminat, PID-ul acestuia și valoarea returnată de acesta. **(cod2-1.c)**

Gestionarea proceselor

Suprascriere cod proces fiu

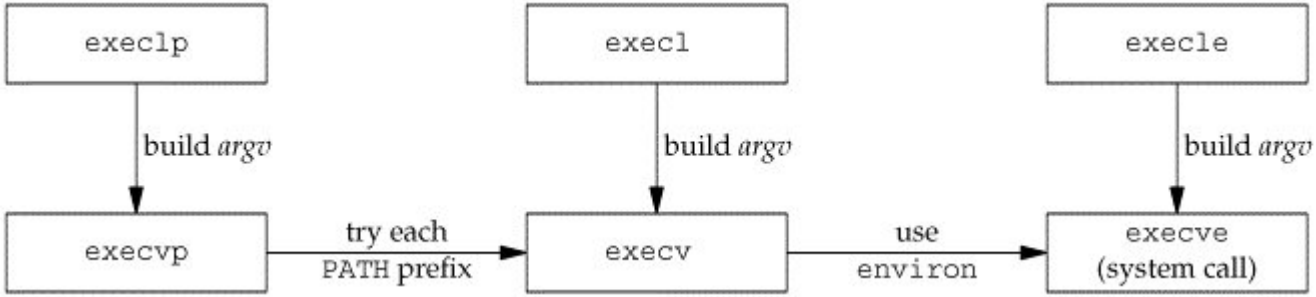
Funcții din familia exec:

- Funcții de biblioteca ce permit suprascrierea codului unui proces cu codul conținut într-un fișier binar executabil de tip ELF
- Programul va fi lansat astfel încât se vor suprascrie codul, datele si stiva procesului care apelează exec( )
- Imediat dupa acest apel codul programului initial (al procesului apelant) nu va mai exista in memorie
- Procesul va rămâne, însă, identificat prin același număr (PID) și va moșteni toate eventualele redirectări făcute în prealabil asupra descriptorilor de fișiere (de exemplu intrarea și ieșirea standard).
- Procesul apelant va păstra relația părinte-fiu cu procesul care a apelat fork( )
- Toate funcțiile din familia exec() în principiu fac același lucru dar pun la dispoziția programatorului mai multe forme de apel
- În final toate aceste funcții ajung la apelul sistem execve(). Putem deci considera ca funcțiile din familia exec() sunt doar niște funcții de tip wrapper peste apelul sistem execve():

Gestionarea proceselor

Suprascriere cod proces fiu

Funcții din familia exec:




```
int execl(const char *path, const char *arg, ... /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ... /* (char  *) NULL */);
int execle(const char *path, const char *arg, ... /*, (char *) NULL, char * const envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]); // APEL SISTEM
```

Gestionarea proceselor

Suprascriere cod proces fiu

Exemplu teoretic de utilizare:

```
int main(void)
{
        int pid;
        int status;
        if ((pid = fork()) < 0)
        {
                ... tratare erori  ...
        }
        if (pid == 0)
        {
                /* cod fiu */
                execlp ("ls","ls","-l",NULL);
                // daca codul a ajuns la aceasta linie -> execlp a esuat
                ... (tratare erori etc) ...
        }
        /* cod parinte */
        ... operatii normale de continuare/finalizare a procesului principal ...
}
```

Gestionarea proceselor

Alte apeluri sistem

Funcția getpid() – returnează PID-ul procesului apelant
Funcția getppid() – returnează PID-ul părintelui procesului apelant
Funcția system() – creează un nou proces ce execută comanda primită ca și argument
reprezintă de fapt o combinație de fork() + execl

Gestionarea proceselor

Comenzi pentru gestionarea proceselor în linie de comandă

Comanda ps

- Pagina de manual: `man ps`
- Afișează la ieșirea standard informații despre procesele active în sistem
- In funcție de argumente poate afișa: PID, user, procent procesor utilizat, procent memorie utilizata, data de start, commanda care a lansat procesul, starea procesului (idle, running, stopped, zombie, … etc)
- Fără nici un argument ps afișează procesele userului curent ce a lansat comanda din terminalul curent

Comanda top

- Similar cu comanda ps dar lista este reîncărcată la o perioadă prestabilită – similar cu task manager din Windows

Comanda htop

- O versiune mult îmbunătățită a comenzii top

Secțiunea III

Comunicarea intre procese

Semnale

Comunicarea între procese

Comunicarea între procese folosind semnale

- Semnalele: reprezintă o modalitate elementară de comunicare între procese prin care acestea pot trimite sau recepționa evenimente
- Un proces poată să trimită semnale către alte procese dar poate să și recepționeze
- La recepția unor semnale procesul poate reacționa astfel:
  - Să le capteze și să execute o acțiune specifică prin intermediul unei funcții de tratare a semnalului (signal handler)
  - Să le ignore
  - Să execute acțiunea implicită la primirea unui semnal (de obicei terminarea procesului)
- Nu toate semnalele pot fi ignorare și nu la toate semnalele se permite modificarea acțiunii standard de terminare a procesului ce e recepționat semnalul
- La apariția unui semnal se întrerupe codul ce rulează în mod uzual – se întrerupe chiar și un apel sistem aflat în execuție sau în așteptare

Comunicarea între procese

Comunicarea între procese folosind semnale

- Semnalele pot sau nu să aibă o acțiune implicită. În POSIX posibila acțiune implicită este definită astfel
  - Term – Acțiunea implicită este terminarea procesului
  - Ign – Acțiunea implicită este ignorarea semnalului
  - Core – Acțiunea implicită este terminarea procesului și producerea unui fișier de core-dump (un fișier ce conține imaginea procesului la momentul terminării – *man 5 core*)
  - Stop – Acțiunea implicită de suspendare a procesului (atenție – nu oprirea acestuia)
  - Cont – Acțiunea implicită de continuare a execuției unui proces suspendat
- Lista semnalelor disponibile într-un sistem POSIX precum și acțiunile implicite (default), ID și alte explicații se găsește într-o pagină de manual dedicată: *man 7 signal*
- Când un proces primește un semnal și are setată un funcție handler evenimentul va fi tratat astfel: (1) se suspendă execuția normală a procesului salvându-i-se contextul (stiva, regiștrii procesorului, PC, …), (2) se execută funcția handler, (3) se restaurează contextul și se continuă execuția codului procesului de unde a rămas înainte de apariția semnalului
- Execuția unui handler de semnal este similară cu execuția unei rutine de tratare a unei întreruperi în CPU sau cu un apel de funcție în care funcția nu este apelată direct de programator ci de sistemul de operare
- Execuția funcției handler este asincronă și codul procesului “nu știe” că aceasta s-a executat sau că a fost întrerupt

Comunicarea între procese

Comunicarea între procese folosind semnale

Semnal	ID	Descriere	Observații
SIGHUP	1	Hangup – terminalul folosit de proces a fost închis	Se poate ignora sau instanția un handler folosind sigaction()
SIGINT	2	Interrupt - întrerupere de la tastatură (în general prin CTRL+C)	
SIGQUIT	3	Quit – cerere de ieșire din program de la tastatură (CTRL+\)	
SIGILL	4	Illegal Instruction – se generează atunci când procesul a executat o instrucțiune al cărei opcode nu are corespondent în setul de instrucțiuni al CPU sau pentru care nu există privilegii suficiente	
SIGABRT	6	Abort – semnal de terminare anormala a procesului generat în general de funcția abort()	
SIGFPE	8	Floating Point Exception – semnal generat atunci când în execuția procesului a apărut o eroare la o operație în virgulă flotantă (ex. împărțire la 0)	
SIGKILL	9	Kill – Semnalul are ca efect terminarea imediată a procesului	Nu se poate ignora și nici instanția o funcție *handler*. Apelul *sigaction* se termină cu eroarea EINVAL dacă este apelat pentru SIGKILL
SIGUSR1	10	User defined 1,2 – Semnale fără semnificație prestabilită lăsate pentru a putea fi folosite de către utilizator	Se poate ignora sau instanția un handler folosind sigaction()
SIGUSR2	12		

Comunicarea între procese

Comunicarea între procese folosind semnale

Semnal	ID	Descriere	Observații
SIGSEGV	11	Segmentation fault – Semnalul apare atunci când procesul a facut un acces ilegal la memorie	Se poate ignora sau instanția un handler folosind sigaction()
SIGPIPE	13	Broken pipe – se generează atunci când s-a încercat scrierea într-un pipe care are descriptorul de la capătul de citire închis	
SIGALRM	14	Timer alarm – semnal generat în urma expirării timer-ului setat de apelul alarm().	
SIGTERM	15	Terminate – specifica o cerere de terminare normală a programului. Utilizatorul poate implementa cum dorește acest semnal	
SIGCONT	18	Continue – are ca efect continuarea unui proces suspendat prin SIGSTOP	
SIGSTOP	19	Stop - Are ca rezultat suspendarea execuției procesului până când aceasta va fi reluata prin primirea unui semnal SIGCONT	Nu se poate ignora și nici instanția o funcție *handler*. Apelul *sigaction* se termină cu eroarea EINVAL dacă este apelat pentru SIGSTOP
SIGCHLD	17	Child terminated – Semnalul este primit de procesul părinte atunci când un proces fiu și-a terminat execuția	Se poate ignora sau instanția un handler folosind sigaction()

Comunicarea între procese

Comunicarea între procese folosind semnale – Semnale timp real

- Sistemele POSIX suportă semnale timp real (real-time signals) după cum este definit începând POSIX.1b
- Sistemele POSIX suportă un număr n de semnale timp real în intervalul [SIGRTMIN; SIGRTMAX]. În general sunt suportate 32 de semnale iar în majoritatea sistemelor linux SIGRTMIN are valoarea 32 si SIGRTMAX are valoarea 64. Totuși se recomandă ca să NU se folosească niciodată în cod valori hard-coded ci în funcție de MACRO-urile SIGRTMIN și SIGRTMAX
- Exemplu: daca se dorește să se folosească semnalul real-time 3 se va folosi valoarea SIGRTMIN+3
- Semnalele real-time se folosesc la fel ca și semnalele standard
- Semnalele real-time nu sunt rezervate pentru acțiuni predefinite și acțiunea implicită este terminarea procesului
- Semnalele real-time se transmit în ordinea în care au fost trimise și acest lucru este garantat. Semnalele cu un număr mai mic au o prioritate mai mare. Nu se specifică clar de POSIX, dacă semnalele real-time au prioritate mai mare decât semnalele standard. În Linux, semnalele standard au prioritate peste semnalele real-time
- În contrast față de semnalele standard care nu pot transmite și date, semnalele real-time pot fi insoțite fie de un întreg fie de un pointer

Comunicarea între procese

Comunicarea între procese folosind semnale

Apelul sistem sigaction()

```
int sigaction(int signum,
              const struct sigaction *_Nullable restrict act,
              struct sigaction *_Nullable restrict oldact);
```

- Apelul sistem instanțiază un comportament pentru semnalul specificat se *signum*. Comportamentul nou este definit de parametrul *act* iar în parametrul *oldact* se poate salva comportamentul precedent.
- act și oldact pot fi NULL astfel:
  - act ☾ NULL – nu se va stabili un comportament nou pentru semnalul signum ci doar se va salva comportamentul curent în oldact
  - ocdact ☾ NULL – nu se va salva comportamentul curent al semnalului signum
  - act și oldact ☾ NULL – apelul nu face nimic
- Pagina de manual: *man 2 sigaction*

Comunicarea între procese

Comunicarea între procese folosind semnale

Structura *struct sigaction*

```
struct sigaction {
            void     (*sa_handler)(int);
            void     (*sa_sigaction)(int, siginfo_t *, void *);
            sigset_t   sa_mask;
            int        sa_flags;
            void     (*sa_restorer)(void);
};
```

sa_handler ☾ pointer la o funcție void cu un parametru întreg ce va reprezenta handler-ul ce se va executa la apariția semnalului

sa_flags ☾ o serie de flag-uri ce se pot combina prin bitwise OR

sa_mask ☾ specifică un set de semnale ce vor fi blocate. Pentru operarea acestui parametru se folosesc funcții precum: *sigemptyset, sigfillset, sigaddset, sigdelset, sigismember*.

sa_restorer ☾ nu se foloseste, în unele implementări nici nu există

sa_sigaction ☾ o variantă mai complexa a sa_handler. Se folosește ori sa_sigaction ori sa_handler. Când se folosește sa_sigaction trebuie specificat flag-ul SA_SIGINFO în caz contrar se va folosi sa_handler.

Se recomandă inițializarea întregii structuri cu 0 înainte de utilizare: *memset(&act, 0, sizeof(act))*.

Comunicarea între procese

Comunicarea între procese folosind semnale

sa_handler mai poate primi și următoarele valori:

- SIG_DFL – reprezintă funcția de tratare implicită a unui semnal (afișare de mesaj și terminarea procesului)
- SIG_IGN – valoare ce semnifică ignorarea semnalului

sa_flags poate primi următoarele valori:

- SA_NOCLDSTOP – dacă signum este SIGCHLD, procesul nu va primi un semnal SIGCHLD atunci când procesul fiu este suspendat (de exemplu cu SIGSTOP), ci numai când acesta își termină execuția;
- SA_ONESHOT sau SA_RESETHAND – va avea ca efect resetarea rutinei de tratare a semnalului la SIG_DFL după prima rulare a rutinei
- SA_ONSTACK – execuția rutinei de tratare va avea loc folosind altă stivă;
- SA_RESTART – oferă compatibilitate cu comportamentul semnalelor în sistemele din familia 4.3BSD – se permite restartarea apelurilor sistem ce au fost întrerupte de apariția semnalului
- SA_NOMASK sau SA_NODEFER – semnalul în discuție nu va fi inclus în mod automat în sa_mask (comportamentul implicit este acela de a împiedica apariția unui semnal în timpul execuției rutinei de tratare a semnalului respectiv);

Comunicarea între procese

Comunicarea între procese folosind semnale

- SA_SIGINFO - se specifică atunci când se dorește utilizarea lui sa_sigaction în loc de sa_handler. În acest caz funcția de tratarea a semalului este diferită de cea simplă și conține pe lângă id-ul semnalului, un
pointer către o structură siginfo_t ca și al doilea argument și un void pointer ca și al treilea argument.
Ex: `void sigaction_handler(int signum, siginfo_t *info, void *ucontex)`
În acest caz parametrii au următoare semnificație
  - Signum – reprezintă ID-ul semnalului
  - info – reprezintă un pointer către o structură de tip siginfo_t ce conține mai multe informații despre semnalul primit
  - Ucontex – reprezintă un pointer către o structură de tip ucontext_t castată la void*

Referitor la acest mod de tratarea a unui semnal este important de menționat că astfel se pot transmite și valori împreună cu semnalul folosind câmpul si_value din structura siginfo_t dar numai în anumite condiții (prin folosirea unui timer)

Comunicarea între procese

Comunicarea între procese folosind semnale

Apelul sistem sigprocmask():

```
int sigprocmask(int how, const sigset_t *_Nullable restrict set,
                         sigset_t *_Nullable restrict oldset);
```

- Apelul are rolul de a schimba masca de semnale pentru thread-ul (atenție: nu proces) apelant. Practic prin acest apel se pot crea seturi de semnale ce vor fi ignorate sau nu de către thread-ul apelant
- Prin acest apel se poate și obține starea curentă de tratare a semnalelor din thread-ul apelant: se poate obține o listă cu semnalele ignorate și cu cele ce nu sunt ignorate
- Parametrul how poate fi:
  - SIG_BLOCK – setul de semnale specificat în *set* reunit cu setul curent vor fi ignorate (blocate)
  - SIG_UNBLOCK – setul de semnale specificat în *set* reunit cu setul curent vor fi tratate
  - SIG_SETMASK – setul de semnale specificat în *set* vor fi ignorate (blocate)
- Parametrul set – lista de semnale se poate construi și manipula folosind următoarele funcții
  - *sigemptyset()* – se inițializează setul la 0 – nu există nici un semnal setat în set
  - *sigfillset()*– se setează toate semnalele din set
  - *sigaddset()* – se adaugă un semnal în set
  - *sigdelset()* – se șterge un semnal din set
  - *sigismember()* – se verifică dacă un anumit semnal este setat în set

Comunicarea între procese

Comunicarea între procese folosind semnale

Apelul sistem sigpending(): scrie în pointerul primit ca și argument lista semnalelor ce așteaptă să fie tratate de proces dar acestea au fost blocate în prealabil

Apelul sistem kill(): `int kill(pid_t pid, int sig);`

- Permite trimiterea unui semnal specificat prin sig de la procesul apelant către procesul specificat prin pid
  - Daca pid > 0, semnalul va fi trimis procesului care are PID-ul egal cu pid;
  - Daca pid == 0, semnalul va fi trimis tuturor proceselor din același grup de procese cu procesul curent;
  - Daca pid == -1, semnalul va fi trimis tuturor proceselor care rulează în sistem (de notat faptul că, în majoritatea implementărilor, nu se poate trimite în acest fel către procesul **init** un semnal pentru care acesta nu are prevăzută o rutina de tratare, si de asemenea, faptul ca de obicei in acest fel procesului curent nu i se trimite semnalul respectiv);
  - Daca pid < -1, se va trimite semnalul către toate procesele din grupul de procese cu numărul -pid.

Apelul sistem raise(): caz particular de kill: kill(getpid(), sig);

Funcția abort(): are ca efect trimiterea imediată către procesul apelant a semnalului SIGABRT. Chiar dacă procesul are instanțiat un handler pentru acest semnal sau procesul ignoră acest semnal la apelul abort() procesul sa va termina. Dacă procesul are instanțiat un handler pentru SIGABRT acesta se va executa după care procesul se va termina în urma apelului abort(). Toate fișierele deschise in interiorul acestuia vor fi închise

Comunicarea între procese

Comunicarea între procese folosind semnale

Apelul sistem alarm(): unsigned int alarm(int seconds);

- are ca efect faptul că nucleul instanțiază un timer inițializat cu numărul de secunde specificat ca parametru
  După expirarea timpului, nucleul trimite procesului apelant un semnal SIGALRM. Dacă o alarma a fost deja programata, ea este anulata în momentul execuției ultimului apel, iar dacă valoarea lui *seconds* este zero, nu va fi programata o alarma nouă ci se returneaza numărul de secunde ramase din alarma precedenta, sau 0 daca nu era programata nici o alarma

Funcțiile sleep() și usleep()

```
unsigned int sleep(unsigned int seconds);
int usleep(useconds_t usec);
```

- Cele 2 functii au ambele rolul de a suspenda executia procesului (sau a thread-ului) apelant pentru o perioada de timp specificata ca parametru. Functia sleep() suspenda executia pentru un numar de secunde iar functia usleep() pentru un numar de microsecunde. Procesul (sau thread-ul) nu va fi suspendat exact cat a fost specificat ci este posibil sa fie suspendat putin mai mult in functie de gradul de incarcare a sistemului si de granularitatea timer-elor din sistem.
- Aceste apeluri vor fi întrerupte la apariția unui semnal

Comunicarea între procese

Comenzi pentru semnale în linie de comandă

Comanda kill

- Pagina de manual: `man 1 kill`
- Comanda permite trimiterea unui semnal din linie de comandă către un proces identificat prin PID
- Practic comanda apelează apelul sistem kill(…)

Comanda killall

- Pagina de manual: `man 1 killall`
- Similar cu comanda kill dar procesul poate fi identificat prin nume. Dacă există mai multe procese cu același nume se va trimite semnalul către toate

Comanda pidof

- Pagina de manual: `man 1 pidof`
- Afișează la ieșirea standard PID-ul unui proces identificat prin nume

Exemplu de cod cu utilizarea apelului alarm și a flag-ului SA_RESTART: cod3-1.c

Pipes

Comunicarea între procese

Comunicarea între procese folosind pipes

Pipe

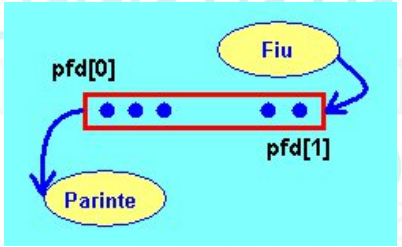
- Un canal de date unidirecțional ce poate fi folosit în comunicarea între procese
- Datele din pipe nu sunt structurate și reprezintă doar octeți raw ce sunt interpretați doar de către aplicație
- Reprezintă de fapt un buffer limitat gestionat de kernel
- Are un capăt de scriere și un capăt de citire, câte un descriptor pentru fiecare
- Se poate accesa folosind apeluri sistem read(), write(), close()
- Politica de acces: FIFO (First In First Out)

Apelul sistem pipe(): `int pipe(int filedes[2]);`

- Are rolul de a crea un pipe. În parametrul filedes apelul va scrie valoarea descriptorilor pentru capătul de scriere respectiv pentru capătul de citire:
  - filedes[0] – descriptorul pentru capătul de citire
  - filedes[1] – descriptorul pentru capătul de scriere
- Doar utilizarea simplă a apelului pipe() nu implementează comunicarea între procese ci este necesară stabilirea unei proceduri ce implică crearea unui pipe înainte de crearea procesului (proceselor) ce vor avea acces la capetele acestuia. Se va face uz de faptul că la momentul creării unui proces fiu acesta va moșteni descriptorii deschiși de procesul părinte

Comunicarea între procese

Comunicarea între procese folosind pipes



Scenariu de utilizare

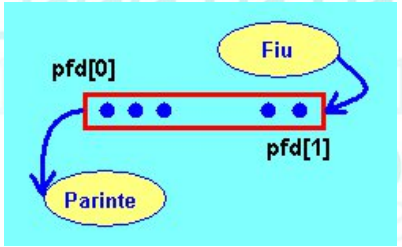
- Un program format din 2 procese
- Procesul fiu va scrie în pipe
- Procesul părinte va citi din pipe

Implementare

1. Procesul părinte creează un pipe prin apelul sistem pipe()
2. Procesul părinte apelează fork() pentru a crea procesul fiu – procesul fiu va moșteni astfel ambele capete ale pipe-ului
3. Procesul fiu va scrie în pipe Ⓟ va închide capătul de citire
4. Procesul părinte va citi din pipe Ⓟ va închide capătul de scriere
5. Procesul fiu va scrie folosind apelul sistem write()
6. Procesul părinte va citi folosind apelul sistem read()
7. Când procesul fiu nu mai are date să scrie în pipe va închide și capătul de scriere

Comunicarea între procese

Comunicarea între procese folosind pipes




```
int main(void)
{
                int pfd[2];
                int pid;

                ...
                if(pipe(pfd)<0)
                {
                  perror("Eroare la crearea pipe-ului\n");
                  exit(1);
                }
                ...
                if((pid=fork())<0)
                {
                  perror("Eroare la fork\n");
                  exit(1);
                }
                if(pid==0)
                {
        /* procesul fiu */
                  close(pfd[0]); /* inchide capatul de citire; */
                                        /* procesul va scrie in pipe  */
                  ...
                  write(pfd[1],buff,len); /* operatie de scriere in pipe */
                  ...
                  close(pfd[1]); /* la sfarsit inchide si capatul utilizat */
                  exit(0);
                }
                /* procesul parinte */
                close(pfd[1]); /* inchide capatul de scriere; */
                                       /* procesul va citi din pipe  */
                ...
                read(pfd[0],buff,len); /* operatie de citire din pipe */
                ...
                close(pfd[0]); /* la sfarsit inchide si capatul utilizat */
```

Comunicarea între procese

Comunicarea între procese folosind pipes

Observații

- Apelul sistem read() va citi datele disponibile din pipe
- Când nu sunt date disponibile în pipe apelul read() se va bloca în așteptare de date (procesul va trece în starea blocked prin tranziția 1). Procesul va rămâne blocate până când vor fi disponibile date în pipe sau până când nu mai există nici un capăt de scriere deschis (logic nu mare are cine și cum să scrie în pipe)
- Se spune că pipe-ul este gol atunci când nu există date în acesta dar există încă deschise capete de scriere (read în acest caz va returna 0)
- Apelul sistem write() va scrie date în pipe
- Apelul sistem write() se va bloca atunci când mai are date de scris în pipe dar pipe-ul este plin iar procesul va trece în starea blocked prin tranziția 1. Procesul va rămâne blocat până când se va elibera spațiu în pipe (prin faptul că alt proces va citi din pipe)
- Dacă se încearcă scrierea într-un pipe ce nu mai are deschis nici un capăt de citire apelul sistem write() se va termina, va returna -1, se va seta eroarea EPIPE în errno și procesul apelant va primi semnalul SIG_PIPE ☾ această situație este denumită broken pipe
- Operațiile de scriere/citire din pipe sunt atomice
- Dimensiunea unui pipe este limitată. Se poate obține dimensiunea curenta din /proc/sys/fs/pipe-max-size,
- Situații de deadlock – situații de blocare ce nu mai pot fi rezolvate
  - Capete de scriere au rămas deschise și nu se mai pot închide – apelurile de read() se vor bloca fără posibilitate de deblocare
  - Capete de citire au rămas deschise și nu se mai pot închide – apelurile de write() se vor bloca fără posibilitate de deblocare

Comunicarea între procese

Comunicarea între procese folosind pipes

Scriere/citire formatată

- Apelurile sistem write() și read() oferă acces la pipe dar se limitează la date nestructurate ☹ se pot transfera doar date la nivel de octeți raw

Funcția fdopen(): `FILE *fdopen(int fd, const char *mode);`

- asociază un descriptor deja deschis unui flux de date (stream) de tip FILE. Dupa apelul acestei functii, se poate scrie si citi formatat dintr-un descriptor folosind funcții din biblioteca stdio precum fprintf, fscanf, fread, fwrite, etc...
- chiar dacă s-a făcut asocierea cu stream de tip FILE, se mai pot folosi în continuare apelurile sistem de baza peste descriptorul fd
- Aceasta funcție se poate folosi pentru orice tip de descriptor ce este deja deschis, fie el descriptor de fișier obținut in urma unui apel sistem open() fie un descriptor de pipe obținut in urma unui apel pipe(). (practic nu exista nicio diferență între ei).
- După asocierea cu un stream de tip FILE, când descriptorul nu mai este folosit acesta trebuie închis. Acest lucru se poate realiza atât cu funcția fclose() asupra stream-ului de tip FILE cat si folosind apelul sistem close() asupra descriptorului inițial.

Comunicarea între procese

Comunicarea între procese folosind pipes

Redirectare descriptori – operațiunea prin care un descriptor oarecare poate să indice un alt fișier decât cel obișnuit inițial.

- Descriptori standard
  - Standard input – 0 – STDIN_FILENO – stdin
  - Standard output – 1 – STDOUT_FILENO – stdout
  - Standard error – 2 – STDERR_FILENO – stderr

Apelul sistem dup(): `int dup(int oldfd);`

- pagina manual: *man 2 dup*
- Realizează duplicarea descriptorului *oldfd* și returnează noul descriptor. Noul descriptor (valoarea cea mai mică disponibilă în sistem) va indica același fișier ca și *oldfd* iar *oldfd* rămâne în continuare valabil pentru fișierul inițial

Apelul sistem dup2(): `int dup2(int oldfd, int newfd);`

- Pagina de manual: *man 2 dup2*
- Similar cu dup() dar i se poate specifica explicit valoarea noului descriptor
- După apelul dup2( ), descriptorul newfd va indica același fișier ca si oldfd. Dacă înainte de operație descriptorul newfd era deschis, fișierul indicat este mai întâi închis, după care se face duplicarea.

Comunicarea între procese

Comunicarea între procese folosind pipes

Redirectare descriptori

- În codul exemplu - newfd devine oldfd
- newfd fiind 1 (stdout)
- ☾ orice scriere la stdout va ajunge în fișierul “Fisier.txt” prin descriptorul oldfd
- Ⓟ textul printat prin printf va ajunge în fișier

- Redirectarile de fisiere se păstrează chiar și după apelarea unei funcții de tip exec() chiar dacă aceasta suprascrie codul procesului, descriptorii rămân aceeași
- Redirectările se folosesc cu precădere în relație cu pipe-uri

```
...
fd=open("Fisier.txt", O_WRONLY);
...
if((newfd=dup2(fd,1))<0)
  {
          perror("Eroare la
dup2\n");
          exit(1);
  }
...
printf("ABCD");
...
```

Comunicarea între procese

Comunicarea între procese folosind named-pipes

Named pipes – pipe-uri cu nume – reprezintă un fișier special de tip pipe care poate fi accesat de orice proces care are drepturi să acceseze acel fișier. Fișierul de tip pipe se comporta ca un fișier obișnuit dar nu este stocat pe mediul de stocare și nici nu își păstrează conținutul și existența la repornirea sistemului

- Crearea unui fișier de tip named pipe:

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

- Pagina de manual: *man 3 mkfifo*
- Creează un fișier de tip pipe cu numele și calea specificat de *pathname* și cui drepturile de acces specificate de *mode*. Parametrul mode are aceeași semnificație și format ca și cel din apelul sistem open()
- Se creează doar fișierul de tip pipe dar nu se și deschide – pentru a se deschide se folosește apelul sistem *open()* sau funcții precum *fopen()* și mai departe se tratează ca un fișier obișnuit

Comanda mkfifo():

- Pagina de manual: *man 1 mkfifo*
- Similar cu funcția *mkfifo* - se creează un fișier de tip pipe la calea specificată ca argument în linie de comandă

Secțiunea IV

Fire de execuție - Threads

Fire de execuție

Concepte și definiții

Thread

- unitatea elementară de execuție – un program în execuție ce are atașat un program counter ce ține evidența codului ce se execută – thread-ul primește de fapt o cuantă de timp CPU
- Dispune de propria stivă, program counter, stare, context (regiștrii procesorului)
- Aparține unui proces – nu poate exista în afara unui proces – procesul fiind o abstractizare a unui program
- Un proces are minim un thread
- Se mai spune că este un lightweight process (LWP)
- Multithreading – conceptul în care un proces conține mai multe thread-uri ce rulează în paralel sau pseudo paralel (când există un singur CPU)

Fire de execuție

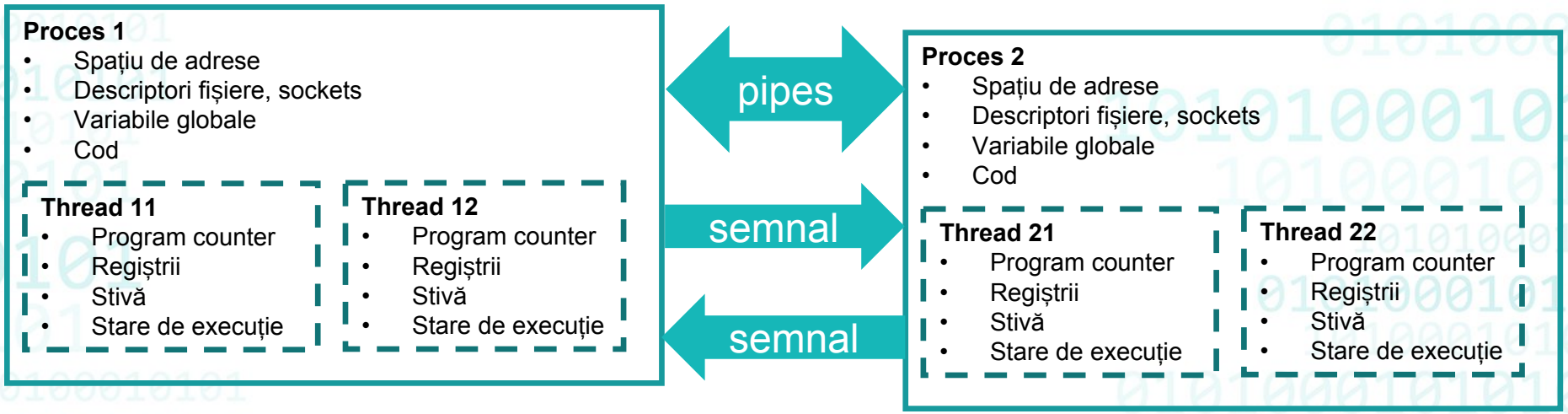
Concepte și definiții

Elementele proprii ale unui proces vs thread

Proces	Thread
• Spațiu de adrese • Variabile globale • Descriptori • Procese fiu • Sockets • Threads	• Program counter • Regiștrii • Stivă • Stare de execuție

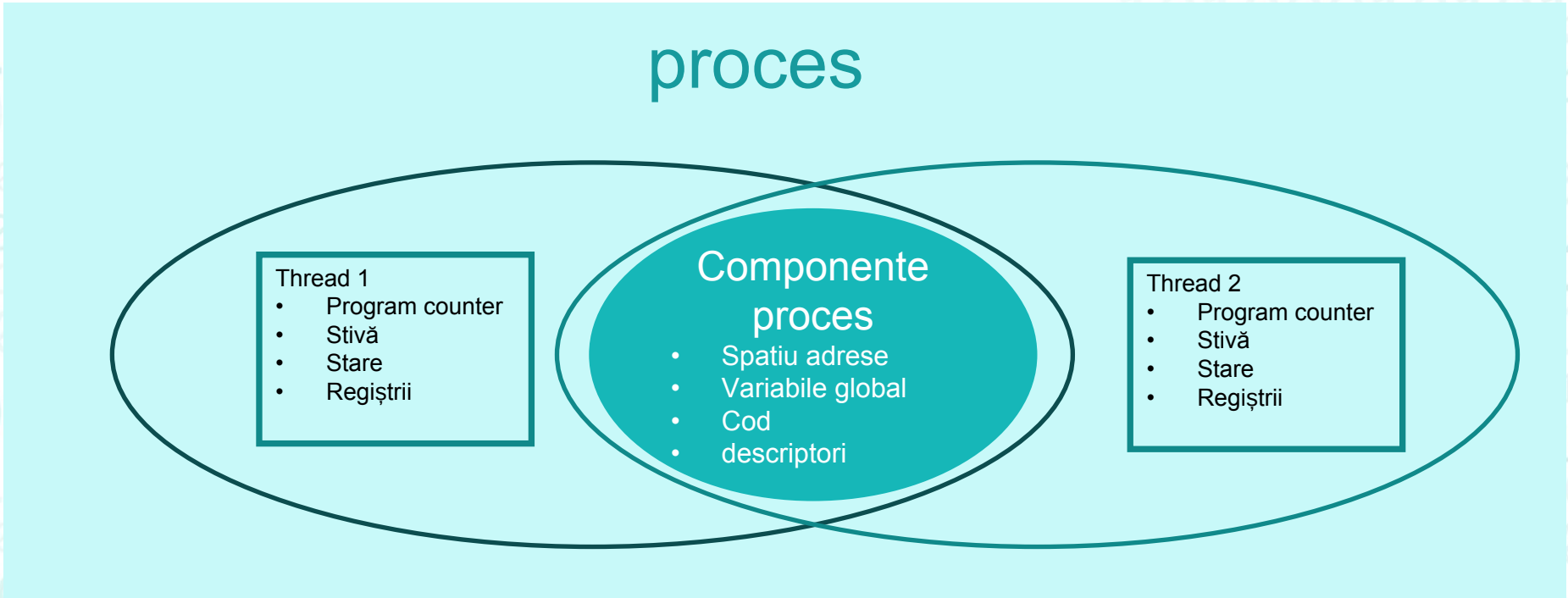
Fire de execuție

Concepte și definiții



Fire de execuție

Concepte și definiții



Fire de execuție

Concepte și definiții

- Fiecare proces are un spațiu de adrese propriu (+ descriptori, sockets, … etc) ☾ astfel ca procesele nu pot comunica decât prin pipe-uri și semnale
- Fiecare proces are minim un thread (thread-ul principal)
- Un thread are propriul Program Counter (PC) ce ține evidența codului în execuție, propria stivă, stare și propriul context
- Un proces poate avea mai multe thread-uri în componență. Toate thread-urile unui proces vor avea următoarele drepturi și accese:
  - vor avea acces la tot spațiul de adrese al procesului, la toate variabilele globale și la tot codul
  - vor avea acces la toți descriptorii, pipe-urile și socket-urile procesului
  - un thread va avea acces la propriul PC, propria stivă, stare și context dar nu și la cele ale altor threaduri din același proces
  - Un semnal trimis unui proces va fi tratat de un thread aleatoriu (thread-ul activ în execuție la momentul recepționării semnalului de către proces)
- Stările unui thread sunt aceleași cu cele ale unui proces: running, ready, blocked (logic ținând cont că un proces are minim un thread)

Fire de execuție

Modele de organizare de thread-uri în sisteme multiprocess și multithreading

- Thread-uri gestionate de user-space
  - Tabela de procese este gestionată de kernel iar tabela de thread-uri este gestionată de fiecare proces pentru thread-urile lui în user-space
  - planificarea thread-urilor are loc la nivelul procesului și poate permite implementarea unui algoritm de planificare personalizat și adaptat aplicației
  - Se aloca timp CPU pentru proces iar acesta va fi mai apoi împărțit thread-urilor
  - Schimbarea de context între thread-uri este mult mai rapidă – datele de context fiind ținute în tabela de thread-uri ale procesului
  - Dacă un thread ocupă întregul timp procesor alocat procesului celelalte thread-uri nu vor mai avea șansa să ruleze
  - Dacă un thread apelează un apel sistem ce produce blocare sau așteptare de date se va bloca întregul proces și implicit și celelalte thread-uri – în acest caz întregul proces va fi trecut în starea blocked – prezintă nu mare dezavantaj (conceptul de multithreading practic dispare) – o rezolvare a problemei o reprezintă implementare de apeluri sistem non-blocking și asincrone dar duce la complicarea codului și rescrierea apelurilor sistem

Fire de execuție

Modele de organizare de thread-uri în sisteme multiprocess și multithreading

- Thread-uri gestionate de kernel
  - Tabela de procese și tabela de thread-uri este gestionată de kernel
  - planificarea thread-urile are loc la nivelul kernel-ului și limitează implementarea unui algoritm de planificare de către utilizator
  - Se aloca timp CPU pentru fiecare thread independent – fiecare thread va avea șansa la execuție
  - La efectuarea unui apel sistem ce poate produce blocare nu se va bloca întregul proces ci doar thread-ul apelant
  - Schimbarea contextului între procese este mai costisitor dpdv al timpului, acest lucru făcându-se in kernel-space
  - doar thread-urile din kernel space sunt planificabile – luate în calcul de planificatorul sistemului de operare.

Fire de execuție

Modele de organizare de thread-uri în sisteme multiprocess și multithreading

- Thread-uri gestionate hibrid
  - Se poate realiza prin implementarea a două tipuri de thread-uri: thread-uri user-space și thread-uri kernel-space în încercarea de maximiza avantajele ambelor variante
  - Se realizează efectiv printr-o mapare a thread-urilor din user-space în thread-uri din kernel-space
  - Există 3 metode de mapare
    1. **One-to-one (1:1)** – fiecare thread din user-space este mapat într-un thread din kernel-space. Când utilizatorul creează un thread în user-space va cere sistemului de operare să creeze un thread în kernel-space ☾ rezultă un overhead și o încetinire a sistemului dar rezolvă problemele de blocare în urma apelurilor de syscalls. Se impune o limită de câte procese pot fi create.
    2. **Many-to-one (M:1)** – fiecare thread din user-space este mapat pe un singur thread din kernel-space Un model rapid dar cu multe dezavantaje precum cele de la procesele pur user-space
    3. **Many-to-Many (M:N)** – fiecare thread din user-space se mapează pe un thread din kernel-space dar pot fi mai multe thread-uri user-space mapate pe același threaduri kernel-space. Nu există restricții
    de câte procese se pot crea, nu apare blocarea procesului la blocarea într-un apel sistem (1:1). Greu de implementat

OBS: Sistemele Linux folosesc 1:1

Fire de execuție

Biblioteca Pthread

- Reprezintă o interfață de gestionare a thread-urilor – nu precizează neaparat modelul de threading folosit
- Pthread = POSIX Thread
- Standardizat în IEEE 1003.1c
- Specifică o bibliotecă de funcții pentru gestionarea thread-urilor
- În biblioteca pthreads, fiecare thread este identificat printr-un identificator unic (număr intreg) implementat prin tipul de date *pthread_t*
- Fiecare thread mai are un identificator unic în kernel numit TID = Thread ID acesta fiind direct “legat” de identificatorul din pthreads
- În cazul thread-ului principal al unui proces PID == TID
- Biblioteca este în user-space și nu conține apeluri sistem
- În Linux biblioteca implementează procese în user-space după modelul 1:1 folosind apelul sistem clone (clone – o variantă mai generală a lui fork())

Fire de execuție

Biblioteca Pthread – crearea unui thread

Funcția *pthread_create*:

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

- Are rolul de a crea și de a lansa în execuție un thread a cărui cod este reprezentat de codul funcției start_routine (printr-un pointer la o funcție existentă)
- Thread-ul se va termina atunci când funcția thread-ului start_routine va returna sau prin apelarea funcției pthread_exit(…) în codul thread-ului
- Funcția primește ca parametru un pointer către o zonă de memorie existentă de tip *pthread_t* în care va scrie identificatorul unic al thread-ului nou creat. Acest identificator va referi în mod unic acel thread pe toată durata existenței lui.
- Pagina de manual: man 3 pthread_create
- OBS: Identificatorul thread-ului dat de pthread_t reprezintă un identificator al thread-ului relativ la biblioteca pthreads care gestionează thread-urile

Fire de execuție

Biblioteca Pthread – crearea unui thread

Funcția *pthread_create*:

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

- Parametrul *attr* reprezintă un pointer constant către o structură de tip *pthread_attr_t* cu rolul de a specifica anumite proprietăți ale thread-ului ce urmează a fi creat. Acest parametru poate fi și NULL, în acest caz thread-ul se va crea cu niște proprietăți predefinite implicit
- Funcția principală a thread-ului este referită printr-un pointer denumit *start_routine*. Conform definiției pointerului la funcție *start_routine*, funcția principală a thread-ului va returna un void* și va primi un argument de tip void*. Codul acestei funcții reprezintă efectiv codul thread-ului (putem spune ca este funcția “main” a thread-ului). Programatorul NU apelează sub nici o formă în mod explicit această funcție, ci după crearea thread-ului codul acestei funcții reprezintă codul thread-ului. Funcția va avea următoarea formă (exemplu):

```
void* my_thread_function(void *arg)
{
        // function code
        return NULL;
}
```

Fire de execuție

Biblioteca Pthread – crearea unui thread

Funcția *pthread_create*:

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

- Parametrul arg reprezintă argumentul ce va fi trimis funcției referite de start_routine. Acest parametru este reprezentat de un pointer generic void* astfel oferind programatorului un grad mare de flexibilitate. Nu se face nici un fel de verificare asupra acestui parametru ci este responsabilitatea strictă a programatorului asupra modului în care folosește acest parametru. Ceea ce programatorul pune în arg la apelul de pthread_create va ajunge la funcția principala a thread-ului la momentul apelului acesteia la începerea execuției thread-ului
- Parametrul arg poate fi și NULL
- Funcția pthread_create returnează 0 în caz de succes și o valoare diferita de 0 în caz de eroare. În caz de succes, după apelul de pthread_create, thread-ul este creat și lansat în execuție rulând codul din funcția referită de start_routine. În parametrul *thread* se va scrie identificatorul unic al thread-ului nou creat
- În caz de eroare funcția pthread_create returnează codul de eroare descris in pagina de manual.

Fire de execuție

Biblioteca Pthread – crearea unui thread

Funcția *pthread_create*: Exemplu de utilizare

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void* my_thread(void *arg)
{
        printf ("Acesta este codul thread-ului !\n");
        return NULL;
}

int main(void)
{
        pthread_t th;
        if (pthread_create(&th, NULL, my_thread, NULL) != 0)
        {
                exit(-1);
        }
        // restul codului din main
        // ...
        // ...
        return 0;
}
```

Fire de execuție

Biblioteca Pthread – Terminarea unui thread

- Un thread se termină fie când se ajunge la instrucțiunea de return a funcției principale a thread-ului (start_routine) fie prin apelul funcției pthread_exit(…)
- La terminarea thread-ului, înregistrarea sa din tabela de thread-uri a procesului nu se șterge ci se așteaptă ca starea lui și valoarea returnată la terminare să fie preluată
- Această proprietate a thread-ului prin care se așteaptă citirea stării de terminare și a valorii returnate se numește joinable
- Pentru citirea stării thread-ului terminat și preluarea valorii returnate se folosește funcția pthread_join

Funcția pthread_join(…): `int pthread_join(pthread_t thread, void **retval);`

- Funcția așteaptă ca thread-ul identificat prin parametrul thread să se termine iar dacă retval este nenull atunci în această zonă de memorie va scrie valoarea returnată de funcția principală a thread-ului (valoarea returnată de *start_routine*)
- Funcția este cu blocare până la terminarea thread-ului
- Funcția returnează 0 în caz de succes și iar în caz de eroare returnează un cod nenul
- Funcția se poate apela doar pentru thread-uri cu proprietatea JOINABLE
- Proprietatea opusă proprietății joinable este DETACHED
- În cazul unui thread DETACHED la terminarea acestuia se șterge starea, valoarea returnată și intrarea sa din tabela de thread-uri a procesului de care aparține
- Dacă nu se specifică, un thread implicit se creează ca fiind JOINABLE
- Pagina de manual: man 3 *pthread_join*

Fire de execuție

Biblioteca Pthread – Terminarea unui thread

Funcția *pthread_join*: Exemplu de utilizare

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void* my_thread(void *arg)
{
        printf ("Acesta este codul thread-ului !\n");
        return NULL;
}

int main(void)
{
        pthread_t th;
        if (pthread_create(&th, NULL, my_thread, NULL) != 0)
        {
                exit(-1);
        }
        // restul codului din main
        // ...
        // ...
        if (pthread_join(th, NULL) != 0)
        {
                exit(-1);
        }
        return 0;
}
```

Fire de execuție

Biblioteca Pthread – Atributele unui thread

Atributele unui thread. Parametrul *pthread_attr_t attr*

- parametrul *attr* este practic o variabilă ce conține o serie de caracteristici ale thread-ului ce se va crea prin apelul *pthread_create*
- Tipul de date *pthread_attr_t* reprezintă o structură ce conține atributele ce vor fi transmite thread-ului. Membrii acestei structuri nu se accesează în mod explicit ci prin intermediul unor funcții dedicate

Inițializarea variabilei de tip pthread_attr_t: `int pthread_attr_init(pthread_attr_t *attr);`

- Înainte de utilizare variabila attr se va inițializa prin intermediul funcției pthread_attr_init(…)
- Funcția primește ca argument un pointer către o variabilă existentă de tip pthread_attr_t pe care o va inițializa. Funcția returnează 0 în caz de succes sau un cod de eroare nenul
- În funcție de implementare inițializarea unei astfel de structuri prin acest apel poate duce la alocare dinamică de memorie
- Pagina de manual: *man 3 pthread_attr_init*
- Efectuarea acestui apel pe o variabilă attr deja inițializată va duce la un comportament nedefinit.

Distrugerea unei variabile inițializate de tip *pthread_attr_t:* `int pthread_attr_destroy(pthread_attr_t *attr);`

- După utilizarea variabilei de tip pthread_attr_t prin apelul pthread_attr_init(…) este necesar să se distrugă folosind apelul *pthread_attr_destroy*.
- Primește ca parametru variabila pthread_attr_t ce a fost inițializată în prealabil pentru a fi distrusă.
- După distrugere, o astfel de variabilă attr poate fi refolosită printr-un alt apel ulterior de *pthread_attr_init(…)*
- Returnează 0 în caz de succes sau un cod de eroare nenul
- Distrugerea unei variabile atribut după apelul funcției *pthread_create* nu are nici un efect asupra thread-ului creat

Fire de execuție

Biblioteca Pthread – Atributele unui thread

Atributul DETACHED / JOINABLE

- Se poate seta într-o structură pthread_attr_t prin funcția pthread_attr_setdetachstate(….)

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

- Funcția primește ca argument un pointer către o structură pthread_attr_t unde să seteze atributul DETACHED / JOINABLE prin parametrul detachstate. Acesta poate avea valorile:
  - PTHREAD_CREATE_DETACHED – se setează atributul ca fiind un thread DETACHED
  - PTHREAD_CREATE_JOINABLE – se setează atributul ca fiind un thread JOINABLE
- Pagina de manual: *man 3 pthread_attr_setdetachstate*

- Se poate obține valoarea setată într-o structură pthread_attr_t prin funcția pthread_attr_getdetachstate(…)

```
int pthread_attr_getdetachstate(const pthread_attr_t *attr,  int *detachstate);
```

- Funcția scrie în pointerul dat de detachstate valoarea acestui atribut din structura dată prin attr
- Poate fi doar una din valorile descrise mai sus
- Pagina de manual: *man 3 pthread_attr_getdetachstate*

Fire de execuție

Biblioteca Pthread – Atributele unui thread

Atributul dimensiune stivă a thread-ului

- Se poate seta într-o structură pthread_attr_t dimensiunea stivei thread-ului prin funcția pthread_attr_setstacksize(…)

```
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

- Funcția primește ca argument un pointer către o structură pthread_attr_t unde să seteze atributul cu noua dimensiune a stivei specificată prin argumentul stacksize (in bytes)
- Pagina de manual: *man 3 pthread_attr_setstacksize*

- Se poate obține valoarea setată într-o structură pthread_attr_t a dimensiunii stivei thread-ului prin funcția pthread_attr_getdetachstate(…)

```
int pthread_attr_getstacksize(const pthread_attr_t *restrict attr, size_t *restrict stacksize);
```

- Funcția scrie în pointerul dat de stacksize valoarea curenta a dimensiunii stivei (in bytes) setată în attr
- Pagina de manual: *man 3 pthread_attr_getstacksize*

Fire de execuție

Biblioteca Pthread – Atributele unui thread

Atributul de CPU affinity a unui thread.

- CPU affinity reprezintă o listă de procesoare pe care therad-ul se dorește să ruleze. Practic dacă un sistem are 4 procesoare (sau core-uri) și CPU affinity reprezintă lista 0,2,3 rezultă că thread-ul poate rula doar pe core-urile 0,2 sau 3 și nu pe core-ul 1
- Se poate seta într-o structură pthread_attr_t folosind funcția

```
int pthread_attr_setaffinity_np(pthread_attr_t *attr,size_t cpusetsize, const cpu_set_t *cpuset);
```

- Funcția primește ca argument un pointer către o structură pthread_attr_t unde să seteze atributul cu noul set de CPU-uri pe care acesta să ruleze. Noul set este dat de parametrul cpuset iar parametrul cpusetsize reprezeintă dimensiunea lui cpuset De obicei cpusetsize = sizeof(cpu_set_t) sau cpusetsize = sizeof(*cpuset)
- Setul de CPU-uri dat prin cpuset se poate manipula doar prin intrmediul unor macro-uri precum CPU_ZERO(), CPU_SET(), CPU_CLR(), CPU_ISSET(), CPU_COUNT()
- Pagina de manual pentru aceste macro-uri este comună explică utilizarea acestora în amănunt (ex: man CPU_SET)
- Pagina de manual: *man 3 pthread_attr_setaffinity_np*

- Se poate obține valoarea setată într-o structură pthread_attr_t a setului de procesoare utilizate prin funcția

```
int pthread_attr_getaffinity_np(const pthread_attr_t *attr, size_t cpusetsize, cpu_set_t *cpuset);
```

- Pagina de manual: *man 3 pthread_attr_getaffinity_np*

OBSERVAȚIE: sufixul _np se traduce prin non-portable și semnifică faptul că aceste funcții sunt caracteristice doar pentru sistemul de operare respectiv și nu sunt neapărat valabile în standard POSIX

Fire de execuție

Biblioteca Pthread – Atributele unui thread

Atributul politică de scheduling

- Se poate seta într-o structură pthread_attr_t politica de scheduling pe care să o folosească kernelul pentru a planifica thread-ul

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

- Funcția primește ca argument un pointer către o structură pthread_attr_t unde să seteze atributul cu noua politică de scheduling. Valorile permise sunt SCHED_RR, SCHED_FIFO, SCHED_OTHER, SCHED_DEADLINE
- Atenție: conform paginii de manual, pentru ca să se schimbe politica de scheduling, este necesar să se seteze și atributul inherit-schueduler cu valoarea PTHREAD_EXPLICIT_SCHED prin funcția pthread_attr_setinheritsched(…).
- Pagina de manual: *man 3 pthread_attr_setschedpolicy*

- Se poate obține valoarea setată într-o structură pthread_attr_t a politicii de scheduling prin funcția:

```
int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr, int *restrict policy);
```

- Funcția scrie în pointerul dat de policy valoarea curenta a politicii de scheduling
- Pagina de manual: *man 3 pthread_attr_getschedpolicy*

Fire de execuție

Biblioteca Pthread – Atributele unui thread

Atributul de moștenire politică de scheduling

- Se poate seta într-o structură pthread_attr_t dacă thread-ul va moșteni politica de scheduling a thread-ului creator sau va folosi politica de scheduling dată prin setul de atribute attr.

```
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);
```

- Funcția primește ca argument un pointer către o structură pthread_attr_t unde să seteze atributul. Valorile permise sunt
  - PTHREAD_INHERIT_SCHED (valoarea implicită) – thread-ul nou creat va moșteni politica de scheduling de la thread-ul creator (cel ce apelează pthread_create)
  - PTHREAD_EXPLICIT_SCHED – thread-ul nou creat nu va moșsteni politica de scheduling de la thread-ul creator și o va folosi pe cea specificată prin *pthread_attr_setschedpolicy*
- Pagina de manual: *man 3 pthread_attr_setinheritsched*

- Se poate obține valoarea setată într-o structură pthread_attr_t a acestui atribut:

```
int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr, int *restrict inheritsched);
```

- Funcția scrie în pointerul dat de **`inheritsched`** valoarea curenta a atributului
- Pagina de manual: *man 3 pthread_attr_getinheritsched*

Fire de execuție

Biblioteca Pthread – Atributele unui thread

Obținerea atributelor unui thread creat aflat în execuție:

Funcția: `int pthread_getattr_np(pthread_t thread, pthread_attr_t *attr);`

- Funcția obține atributele curente ale unui thread creat și aflat deja în execuție identificat prin identicatorul unic al thread-ului din parametrul thread. Funcția va scrie atributele în zona de memorie referită de attr
- Atenție: Funcția nu este standard.

Obținerea valorile implicite (default) ale unei structuri de atribute. Aceste valori vor fi folosite la crearea unui thread când se apelează pthread_create cu NULL la parametrul attr.

Funcția: `int pthread_getattr_default_np(pthread_attr_t *attr);`

- Există și posibilitatea de a se seta valorile implicite folosite la crearea unui thread. Acestui lucru se face folosind:

Funcția: `int pthread_setattr_default_np(const pthread_attr_t *attr);`

Fire de execuție

Biblioteca Pthread – Atributele unui thread

- Atributele unui thread se pot schimba și prin anumite funcții dedicate fără a folosi structuri precum pthread_attr_t. Aceste funcții pot permite atât setarea anumitor atribute din exteriorul thread-ului cât și din interiorul acestuia – practic un thread își poate schimba singur atributele în timpul execuției
- Pentru aceste tipuri de funcții este în general necesar identificatorul thread-ului pthread_t (și nu TID-ul)

Schimbarea atributului DETACHED / JOINABLE

- Funcția `int pthread_detach(pthread_t thread);` - transformă thread-ul identificat prin valoarea dată ca argument într-un thread DETACHED
- Un thread o dată trecut în starea DETACHED nu mai poate fi făcut înapoi JOINABLE

Schimbarea atributului de CPU affinity

- Funcția `int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize, const cpu_set_t *cpuset);` are rolul de a schimba propritatea de afinitate în timpul execuției thread-ului identificat prin thread. Funcționeaza similar cu setarea atributului într-o structură pthread_attr_t.
- Există și funcție de a interoga starea curenta a acestui atribut: pthread_getaffinity_np(…)

Fire de execuție

Biblioteca Pthread – Alte apeluri utile pthread

Funcția pthread_self: `pthread_t pthread_self(void);`

- Returnează ID-ul pthread_t al thread-ului apelant

Funcția pthread_equal: `int pthread_equal(pthread_t t1, pthread_t t2);`

- Returnează o valoare nenulă daca cei doi identificatori de thread-uri de tip pthread_t reprezintă același thread

Funcția pthread_exit: `void pthread_exit(void *retval);`

- Are ca efect terminarea imediată a thread-ului apelant. Dacă acesta este JOINABLE, în apelul de pthread_join se va copia valoarea dată lui pthread_exit(…) ca și argument retval.
- Similară ca și logică cu apelul sistem exit(…) dar funcționează la nivel de thread și nu reprezintă un apel sistem

Apelul sistem gettid(…): pid_t gettid(void);

- Returnează TID-ul thread-ului – identificatorul unic al thread-ului la nivelul kernelului

Secțiunea V

Sincronizarea firelor de execuție

Sincronizarea firelor de execuție

Analiza cod – race-condition

- Se consideră un program în care se creează un număr de thread-uri. Thread-urile vor lucra asupra unei variabile comune. Fiecare thread va incrementa valoarea comună reprezentată de variabila shared_value, va printa rezultatul după care se va suspenda pentru o perioadă de timp prin apelul funcției usleep(…).
- Dacă se realizează mai multe experimente în care se rulează codul exemplu se constantă situația de mai jos (dreapta)

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdint.h>

#include <unistd.h>
#define THREAD_COUNT 4

uint32_t shared_value = 0;
void *thread_function(void *arg)
{
    while(1)
    {
        shared_value++;
        printf ("%d ", shared_value);
        usleep(400 * 1000);
    }

    return NULL;
}
```

```c
int main(void)
{
    int i = 0;
    pthread_t th[THREAD_COUNT];
    for (i = 0; i < THREAD_COUNT; i++)
    {
        if (pthread_create(&th[i],
NULL, thread_function, NULL) != 0)
        {
            exit(EXIT_FAILURE);
        }
    }

    for (i = 0; i < THREAD_COUNT; i++)
    {
        pthread_join(th[i], NULL);
    }


    return 0;
```

cod5-1

```
27 28 28 29 31 31 30 32
33 33 34 35 36 37 36 38
39 40 41 42 43 43 44 45
46 46 46 47 48 50 49
```

Sincronizarea firelor de execuție

Analiza cod – race-condition

- Dacă se realizează mai multe experimente în care se rulează codul exemplu se constantă o situație similară cea de mai jos:

```
27 28 28 29 31 31 30 32 33 33 34 35 36 37 36 38
39 40 41 42 43 43 44 45 46 46 46 47 48 50 49 49
```

- Se observă situația în care unele numere apar de 2,3 ori chiar și la distanță unele de altele. Acest comportament nu este conform cu codul programului
- Problema apare din cauza situației în care mai multe thread-uri accesează necontrolat resurse comune (*shared_value*)
- Situația dată se numește race-condition
- Race-condition – situația în care mai multe fire de execuție accesează prin scriere și/sau citire o resursă comună iar rezultatul
  final corect depinde de ordinea efectuării operațiilor pe acea resursă.
- Critical region (critical section) – partea de program în care sunt accesate (prin scriere/citire) resursele comune

Sincronizarea firelor de execuție

Analiza cod – race-condition

SOLUȚII:

- În cazul sistemelor uniprocesor se dezactivează întreruperile pe durata codului din critical section – se pretează de obicei la sisteme embedded – soluția nu este aplicabilă la sistemele multi-procesor și nici pentru sisteme de operare evoluate – nu se poate permite utilizatorului oprirea întreruperilor globale – poate duce la blocarea sistemului de planificare de procese
- lock variables – folosirea unei variabile comune care să marcheze când un thread execută codul din critical section – variabila de lock se pune de ex pe 1 când se intră în critical section și pe 0 când s-a ieșit din critical section. Astfel un thread poate
  verifica dacă există vreun alt thread în execuție în critical section. Soluția are aceeași problema cu situația inițială, variabila de lock fiind partajată deci poate apărea situația de race-condition
- Alternarea execuției proceselor astfel ca prin condiție software se restricționează accesul la resursă:

```
while (true)
{
        while (turn != 0);
        critical_region_code();
        turn = 0;
}
```

```
while (true)
{
        while (turn != 1);
        critical_region_code();
        turn = 1;
}
```

Soluția nu este scalabilă și consumă procesorul inutil (prin bucla de așteptare). Situația în care se așteaptă schimbarea unei variabile se numește **busy-waiting**. Variabila a cărei schimbare de valoare se așteaptă se numește **spin-lock**.

- Toate aceste potențiale soluții încearcă să limiteze accesul la resursa comună astfel încât doar un singur thread să poată accesa acea resursă comună la un moment dat, celelalte thread-uri fiind excluse în a accesa acea resursă comună în acel moment ☾ **excludere mutuală – mutual exclusion**

Sincronizarea firelor de execuție

Mutex

- Mutex (MUTual Exclusion) – reprezintă o variabilă partajată (comuna) de mai multe thread-uri ce poate avea două valori: locked și unlocked.
- În general este nevoie de un singur bit pentru a marca cele două stări dar în implementare se folosesc de obicei tipuri de date
  întregi
- Dacă un thread are nevoie să acceseze un segment de cod dintr-un critical region, acesta va încerca să treacă mutexul în starea locked. Dacă mutexul era în starea unlocked thread-ul îl va trece în starea locked și va continua execuția codului din critical region. La terminarea execuției codului din critical region, thread-ul va elibera mutex-ul și îl va trece în starea unlocked
- Pe de altă parte, dacă thread-ul nu găsește mutex-ul în starea unlocked, la încercarea de a-l trece pe acesta în starea locked apelul se va bloca în așteptarea trecerii mutex-ului în starea unlocked. Astfel, thread-ul nu va putea avansa în codul din critical region decât după eliberarea mutex-ului prin trecerea acestuia în starea unlocked de către thread-ul care l-a blocat inițial
- Doar thread-u care a blocat un mutex îl poate debloca. Nici un alt thread nu poate debloca mutexul blocat de un alt thread.
- Astfel doar un singur thread la un moment dat va putea accesa codul din critical-region.

Sincronizarea firelor de execuție

Mutex

- În biblioteca pthread mutex-ul este implementat prin tipul de date pthread_mutex_t. Se pun la dispoziție funcții pentru inițializarea unui mutex, distrugerea, blocarea și deblocarea lui

Inițializarea unui mutex. Se realizează în doi pași: variabila de tip pthread_mutex_t se inițializează cu o valoarea prestabilită după care se apelează funcția pthread_mutex_init(…) asupra acesteia.

1. Declararea și inițializarea unei variabile de tip mutex:

```
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
```

- Variabila de tip mutex se declară astfel încât să poată fi accesată de toate thread-urile. Se poate deci declara ori global ori transmisă ca parametru funcției principale a thread-ului la crearea acestuia prin pthread_create.
- Valoarea de inițializare poate fi unul din macro-urile puse la dispoziție de biblioteca pthread

2. Inițializarea unui mutex prin apelul pthread_mutex_init(…)

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
```

- Funcția primește ca argument un pointer către o variabilă de tip mutex declarată și inițializată anterior cu o valoare printr-un macro precum și si un pointer către o structură ce conține un set de atribute sau NULL pentru atribute implicite
- Funcția returnează 0 în orice situația (conform paginii de manual)
- Pagina de manual: man pthread_mutex_init
- Pagina de manual pentru pthread nu este instalată implicit în unele sisteme de operare. Pentru Linux GNU/Debian este necesară instalarea pachetului glibc-doc

```
sudo apt install glibc-doc
```

Sincronizarea firelor de execuție

Mutex

Distrugerea unui mutex. Funcția `int pthread_mutex_destroy(pthread_mutex_t *mutex);`.

- Funcția distruge mutex-ul inițializat în prealabil prin apelul pthread_mutex_init(….) și eliberează eventuala memorie ocupată
- Este recomandat ca un mutex să fie distrus prin apelul acestei funcții atunci când acesta nu mai este utilizat

Blocarea unui mutex. Funcția `int pthread_mutex_lock(pthread_mutex_t *mutex);`

- Această funcție are ca rol blocarea mutex-ului, trecerea acestuia în starea locked
- Primește ca parametrul un pointer către un mutex (inițializat în prealabil) pe care să-l treacă în starea locked
- Dacă la apel, funcția găsește mutex-ul în starea unlocked atunci aceasta în blochează și în trece in starea locked
- Dacă la apel, funcția găsește mutex-ul deja în starea locked atunci apelul se blochează și se așteaptă eliberarea acestui mutex (trecerea lui în starea unlocked) de către thread-ul ce la blocat
- La blocarea apelului în această funcție thread-ul este suspendat din execuție de către kernel (trecut în starea blocked) până când mutex-ul este eliberat de thread-ul ce la blocat. Doar atunci thread-ul este trecut din nou de kernel în starea ready pentru a putea fi din nou lansat în execuție.
- Așadar, apelul nu blochează thread-ul într-o buclă busy-wait (busy-loop) și deci nu consumă procesorul în mod inutil
- Această funcție se apelează de obicei la intrarea intr-o zonă de tip critical section în care se accesează zone de memorie partajate (shared)
- Apelul returnează 0 în caz de succes și un cod de eroare compatibil cu errno (dar nu setează errno)
- Pagina de manual: `man pthread_mutex_lock`

Sincronizarea firelor de execuție

Mutex

Deblocarea unui mutex. Funcția `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

- Această funcție are ca rol deblocarea mutex-ului, trecerea acestuia în starea unlocked
- Primește ca parametrul un pointer către un mutex (inițializat în prealabil) pe care să-l treacă în starea unlocked
- Funcția va debloca mutex-ul doar dacă acesta a fost blocat de același thread (cu thread-ul apelant) altfel va genera o eroare și nu va debloca mutex-ul
- Această funcție se apelează de obicei la ieșirea dintr-o zonă de tip critical section în care se accesează zone de memorie partajate (shared)
- Apelul returnează 0 în caz de succes și un cod de eroare compatibil cu errno (dar nu setează errno)
- Pagina de manual: `man pthread_mutex_unlock`

Sincronizarea firelor de execuție

Mutex

- Se consideră exemplul precedent în care fiecare thread incrementează un număr dintr-o variabilă partajată în care accesul la zona partajată este controlat prin utilizarea unui mutex

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#define THREAD_COUNT 4

uint32_t shared_value = 0;
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
void *thread_function(void *arg)
{
    int r = 0;
    while(1)
    {
        if ((r = pthread_mutex_lock(&my_mutex)) != 0)
        {
            fprintf(stderr, "Mutex lock error: %s\n", strerror(r));
            pthread_exit(NULL);
        }
        // start of critical section
        shared_value++;
        printf ("%d ", shared_value);
        // end of critical section
        if ((r = pthread_mutex_unlock(&my_mutex)) != 0)
        {
            fprintf(stderr, "Mutex unlock error: %s\n", strerror(r));
            pthread_exit(NULL);
        }
        usleep(400 * 1000);
    }
    return NULL;
}

int main(void)
{
    int i = 0;
    pthread_t th[THREAD_COUNT];
                pthread_mutex_init(&my_mutex, NULL);
    for (i = 0; i < THREAD_COUNT; i++)
    {
        if (pthread_create(&th[i], NULL, thread_function, NULL) != 0)
        {
            exit(EXIT_FAILURE);
        }
    }

    for (i = 0; i < THREAD_COUNT; i++)
    {
        pthread_join(th[i], NULL);
    }
    pthread_mutex_destroy(&my_mutex);
    return 0;
}
```

Sincronizarea firelor de execuție

Semafoare (pthread)

- Inventat de Edsgar W. Dijkstra
- Spre deosebire de mutex care oferă mecanisme de excludere mutuală pentru a controla accesul la resurse partajate, semaforul reprezintă o metodă mai generală de sincronizare a thread-urilor ce nu este neapărat legată de accesul la resurse comune
- Semaforul este implementat ca o variabilă de tip întreg ce poate fi incrementată și decrementată prin utilizarea unor funcții dedicate
- Asupra unui semafor se pot face două operații:

1. Decrementare (down) – **wait** – proberen. Această operațiune presupune doi pași. În primul pas se verifică dacă semaforul este nenul. În acest caz (al doilea pas) semaforul este decrementat și operațiunea se încheie. În cazul în care semaforul are valoarea 0 operațiunea (apelul) suspendă thread-ul apelant, nu se decrementează valoarea ci se așteaptă până când semaforul devine din nou mai mare strict ca 0. Doar când semaforul are o valoarea nenulă este decrementat
2. Incrementare (up) – post – verhogen. Această operațiune presupune incrementarea unui semafor. Dacă semaforul are valoarea 0, incrementarea va duce la o valoarea nenulă ce va determina toate thread-urile ce așteaptă într-un apel de **wait.**

- Cele două operații se execută atomic și indivizibil – o operație atomică nu poate fi întreruptă. Orice implementare de semafoare trebuie să garanteze că operațiile asupra semafoarelor sunt atomice și indivizibile.
- Spre deosebire de mutex ce poate fi deblocat doar de către thread-ul care l-a și blocat, această restricție nu se aplică și semafoarelor. Un semafor poate și accesat din orice thread ce are fizic acces la el
- De asemenea, un semafor poate fi partajat nu doar de thread-uri ci și de procese. Există suport astfel încât un semafor să poată fi accesat din procese diferite – acest lucru îi lărgește mult aplicabilitatea și devine extrem de scalabil
- În Linux informații generale teoretice despre semafoare se pot obține din pagina de manual: *man 7 sem_overview*

Sincronizarea firelor de execuție

Semafoare (pthread)

- Semaforul POSIX din biblioteca pthreads – implementat printr-o variabilă de tip sem_t

Inițializarea unui semafor: `int sem_init(sem_t *sem, int pshared, unsigned int value);`

- Funcția are rolul de inițializa o variabilă de tip semafor. Este necesar ca un semafor să fie inițializat cu această funcție inainte de utilizare
- Parametrul *sem* – un pointer către o variabilă de tip semafor pe care să o inițializeze
- Parametrul *pshared* – un întreg ce stabilește tipul de semafor. Dacă acest argument are valoarea 0 atunci semaforul va fi partajat doar între thread-uri. Dacă este o valoare nenulă semaforul va fi partajat între procese.
- Parametrul *value* – un întreg ce reprezintă valoarea inițială a semaforului
- Funcția returnează 0 în caz e succes și -1 în caz de eroare cu setarea variabilei globale *errno*
- Pagina de manual: *man 3 sem_init*

Distrugerea unui semafor: `int sem_destroy(sem_t *sem);`

- Funcția are rolul de a distruge un semafor inițializat în prealabil cu sem_init(…) și de a dealoca eventualele resurse alocate pentru semaforul respectiv
- Primește ca argument un pointer către o variabilă de tip semafor
- Funcția returnează 0 în caz e succes și -1 în caz de eroare cu setarea variabilei globale *errno*
- Pagina de manual: *man 3 sem_destroy*

Sincronizarea firelor de execuție

Semafoare (pthread)

Decrementarea și testarea unui semafor: `int sem_wait(sem_t *sem);`

- Funcția întâi verifică dacă valoarea semaforului este nenulă. În cazul în care valoarea semaforului este nulă, funcția se va bloca și va aștepta ca valoarea semaforului să fie strict mai mare decât 0. Blocarea presupune suspendarea thread-ului apelant și trecerea acestuia în starea blocked. Pe de altă parte, în cazul în care la apelul funcției (sau la schimbarea valorii semaforului) valoarea semaforului este strict mai mare decât zero, funcția va decrementa valoarea semaforului și va returna din apel
- Parametrul *sem* – un pointer către o variabilă de tip semafor a cărui valoare să o decrementeze (și/sau să aștepte până când aceasta devine nenulă)
- Funcția returnează 0 în caz e succes și -1 în caz de eroare cu setarea variabilei globale *errno*. În caz de eroare valoarea semaforului rămâne nemodificată
- Pagina de manual: *man 3 sem_wait*

Incrementarea unui semafor: `int sem_post(sem_t *sem);`

- Funcția are rolul de a incrementa valoarea unui semafor
- Primește ca argument un pointer către o variabilă de tip semafor
- Funcția returnează 0 în caz e succes și -1 în caz de eroare cu setarea variabilei globale *errno.* În caz de eroare valoarea semaforului rămâne nemodificată
- În cazul în care semaforul are valoarea 0 și deci în urma acestui apel va fi incrementat, situația va genera efectul în care thread-urile blocate în apelul se *sem_wait* vor fi trecute în starea *ready* spre a fi din nou lansate în execuție
- Pagina de manual: *man 3 sem_post*

Sincronizarea firelor de execuție

Semafoare (pthread)

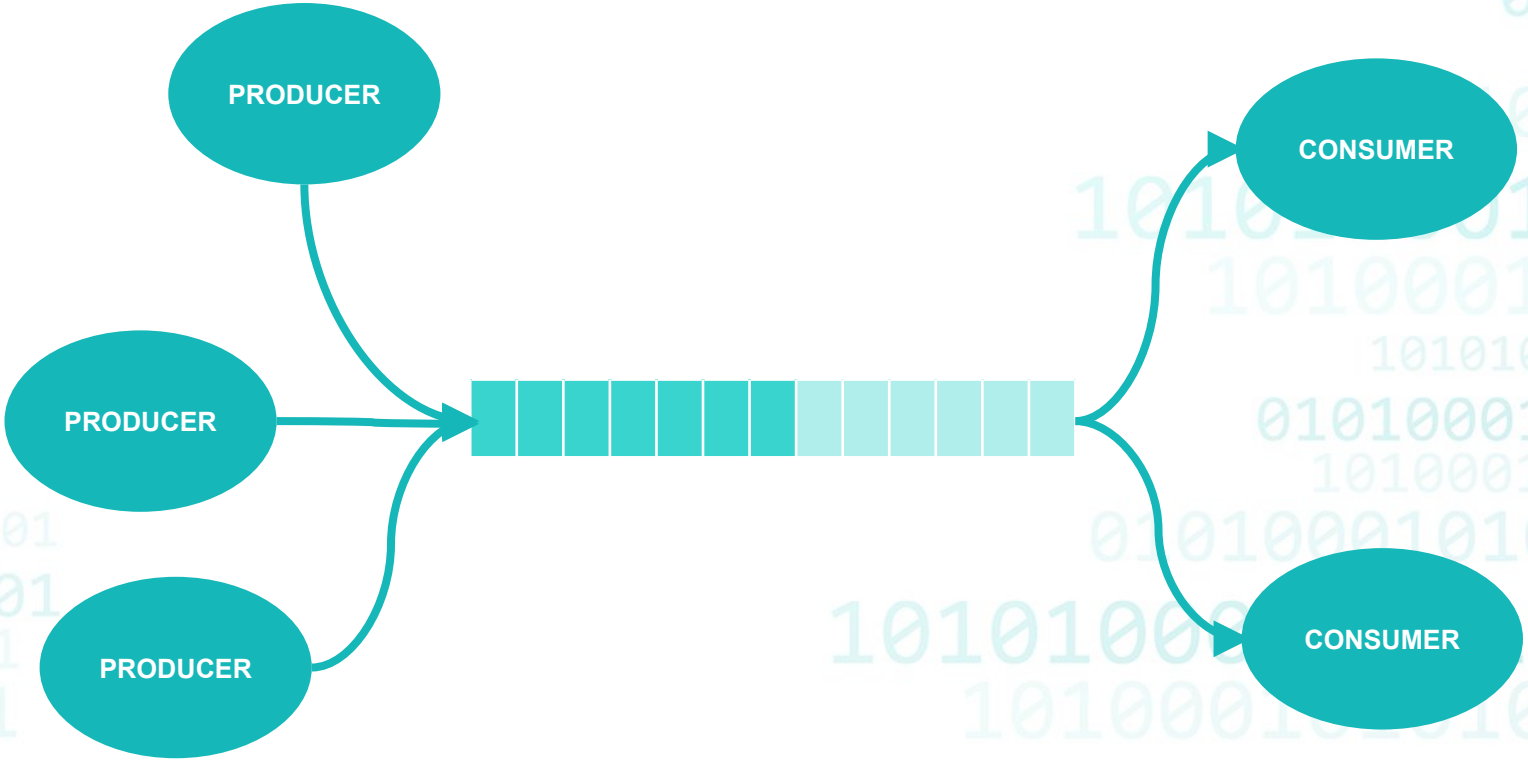
Aflarea valorii curente a unui semafor: `int sem_getvalue(sem_t *restrict sem, int *restrict sval);`

- Funcția are rolul de a obține valoarea curentă a unui semafor.
- Scopul principal al acestei funcții este în general facilitarea operațiunilor de depanare a programelor multithreading, sincronizarea thread-urilor fiind în general greu de depanat
- Parametrul *sem*: un pointer către un semafor inițializat în prealabil de la care se dorește obținerea valorii
- Parametrul *sval*: un pointer către o zonă de memorie (o variabilă) de tip întreg în care să scrie valoarea obținută de la semafor
- Returnează 0 în caz de succes și -1 în caz de eroare și setează codul de eroare în variabila *errno*

Sincronizarea firelor de execuție

Problema producător – consumator

- Se consideră o zonă de memorie mărginită în care P thread-uri produc date (adaugă date în zona de memorie) și C thread-uri care consumă date din zona de memorie (extrag datele)



Sincronizarea firelor de execuție

Problema producător – consumator

- Scopul producătorului este să producă date și să le introducă în buffer-ul comun. Scopul consumatorului este să extragă date din buffer-ul comun
- Pentru simplitate se consideră că producătorul va adăuga date într-un buffer liniar tot timpul la sfârșitul buffer-ului pe prima poziție liberă iar consumatorul va extrage date din buffer tot de la capătul buffer-ului de pe prima poziție (de la coadă) găsită conținând date ☾ de fapt buffer-ul se va accesa după principiul stivei
- Producătorul va adăuga date dacă găsește minim o poziție liberă în buffer – dacă buffer-ul nu este plin. În cazul în care buffer-ul este plin producătorul nu va adăuga date în buffer și va aștepta eliberarea a minim o poziție din buffer
- Consumatorul va extrage date din buffer doar dacă buffer-ul nu este complet gol. Dacă buffer-ul este complet gol atunci consumatorul va aștepta producere unui element în buffer
- Se vor folosi două semafoare
  - sem_buffer_empty – semaforul se va inițializa cu dimensiunea maxima a buffer-ului. Acesta va număra câte poziții sunt libere în buffer. Dacă acest semafor este 0 înseamnă că nici o poziție din buffer nu mai este goală, deci buffer-ul este de fapt plin. Producătorul va verifica și decrementa semaforul înainte de a adăuga noul element în buffer. Dacă producătorul găsește acest semafor pe 0, codul se va bloca ținând cont că nu există poziții libere în buffer și va aștepta ca vreun consumator să extrage minim un element lăsând astfel o poziție liberă în buffer. La rândul lui, consumatorul va incrementa acest semafor imediat după ce a consumat un element din buffer.
  - sem_buffer_full – semaforul se va inițializa cu 0 și va număra câte poziții sunt ocupate în buffer. Producătorul va incrementa acest semafor imediat după adăugarea unui element în buffer. La fiecare execuție a buclei principale, consumatorul va verifica și decrementa acest semafor înainte de a încerca consumarea unui element din buffer. Consumatorul va verifica întâi dacă valoarea acestui semafor este 0, acest caz însemnând că nu există nici o poziție ocupată în buffer, deci nu are ce să consume și astfel așteaptă ca un fir producător să adauge elemente în buffer. În cazul în care valoarea acestui semafor este nenulă, consumatorul are minim un element disponibil spre a fi extras.

Sincronizarea firelor de execuție

Problema producător – consumator

- Pe lângă cele două semafoare, pentru implementarea problemei mai este necesar un mutex pentru a asigura controlul asupra buffer-ului de elemente, acesta fiind o resursă comună pentru toate thread-urile consumator și producător – este nevoie de excludere mutuală.
- O implementare în pseudo-cod a algoritmilor pentru consumator și producător poate fi următoarea

Algoritm implementare thread producator

```
item ⇝ elementul ce se va produce
sem_buffer_empty ⇝ semafor pentru semnalare buffer gol
sem_buffer_full ⇝ semafor pentru semnalare buffer plin
mutex_buffer_access ⇝ mutex pentru controlul accesului la buffer
producer_running ⇝ valoare booleană – codul producătorului este în execuție
Inițializare variabile: producer_running ⇝ true
while (producer_running == true)
    item = produce_element()
    sem_wait_and_decrement(sem_buffer_empty)
    lock_mutex(mutex_buffer_access)
    adaugare_item_in_buffer(item)
    unlock_mutex(mutex_buffer_access)
    sem_increment(sem_buffer_full)
end
```

Algoritm implementare thread consumator

```
item ⇝ elementul ce se va obține din buffer
sem_buffer_empty ⇝ semafor pentru semnalare buffer gol
sem_buffer_full ⇝ semafor pentru semnalare buffer plin
mutex_buffer_access ⇝ mutex pentru controlul accesului la buffer
consumer_running ⇝ valoare booleană – codul consumatorului este în execuție
Inițializare variabile: consumer_running ⇝ true
while (consumer_running == true)
    sem_wait_and_decrement(sem_buffer_full)
    lock_mutex(mutex_buffer_access)
    item ⇝ extrage_item_din_buffer();
    unlock_mutex(mutex_buffer_access)
    sem_increment(sem_buffer_empty)
end
```

- Să se implementeze în C o problema producător-consumator considerând ca date numere întregi: cod5-3
- Codul 5-3 implementează problema și se va oprit la semnalul SIGINT (CTRL+C) prin comportamentul implicit al acestuia. S-a implementat și varianta de oprire controlată la recepționarea semnalului SIGTERM – totuși în acest caz există o problemă și uneori unele thread-uri nu se opresc și rămân blocate. De ce ? Găsiți problema. Găsiți o rezolvare a problemei !

Sincronizarea firelor de execuție

Problemă producător – consumator

- Rezolvați problema producător – consumator fără semafoare ci doar cu mutex-uri și bucle busy-wait în loc de semafoare ? Realizați o comparație între cele 2 soluții și arătați avantajele și dezavantajele celor două variante
- Adaptați și generalizați problema producător – consumator
  - Folosirea unui buffer circular în loc de un buffer liniar
  - La un buffer liniar să se adauge și consume de pe poziții aleatoare
  - Se existe nivele mai multe de producători: producători primari, producători secundari ce sunt și consumatori de elemente primare și consumatori finali. Exemplu: să se simuleze o fabrică de mașini. Se consideră că fiecare mașină are nevoie de 4 roți, o caroserie și un motor iar fiecare roată are nevoie de un cauciuc și o jantă
- **Rezolvați problema producător – consumator fără thread-uri ci doar cu procese, pipe-uri și semnale (fără semafoare și mutex-uri)

ALTE FUNCȚII UTILE:

1. int sem_trywait(sem_t *sem);
2. int sem_timedwait(sem_t *restrict sem, const struct timespec *restrict abs_timeout);
3. int pthread_mutex_trylock(pthread_mutex_t *mutex);
4. int pthread_cancel(pthread_t thread);

Secțiunea VI

Elemente de gestionarea timpului. Timer.

Reprezentarea timpului

Reprezentarea timpului în sisteme Linux

În sisteme de operare Linux (compatibile POSIX) timpul este implementat și contorizat în mai multe moduri:

- **Timpul calendaristic** (RTC – Real Time Clock) este implementat în Linux ca fiind numărul de secunde de la Epoch
- Epoch se considera a fi: 01.01.1970 ora 00:00:00 UTC (1970-01-01 00:00:00 +0000 (UTC).
  UTC – Universal Time Coordinated – reprezintă în principiu același lucru cu GMT (Greenwich Mean Time) – dar nu este doar un fus orar (time zone) ci un standard de reprezentare a timpului – Militar: Zulu Military Time / Zulu Time Zone
  Ⓟ timpul în sisteme Linux/Unix/POSIX se reprezintă ca fiind numărul de secunde de la 01.01.1970 00:00:00
  În sistemele moderne se stochează ca fiind un număr cu granularitate mai mare de o secunda ☾ milisecundă, nanosecundă
  Din punct de vedere hardware, majoritatea sistemelor conțin un circuit specializat (RTC) ce măsoară timpul independent de restul sistemului hardware și sistemului de operare. Sunt de obicei alimentate pe baterie pentru a contoriza timpul și atunci când sistemul este oprit sau deconectat de la alimentare.
- Process time – se definește ca fiind timpul în care un proces a utilizat procesorul. Este uneori divizat în user process time și system process time în funcție de ce parte de cod a fost măsurată (cod la nivelul user sau cod la nivel kernel)

Reprezentarea timpului

Reprezentarea timpului în sisteme Linux

- Tipuri de clock suportate în Linux – definite ca și MACRO-uri ce reprezintă un *clockid*
  - **CLOCK_REALTIME** – reprezintă ceasul calendaristic (real clock, wall-clock). Acest clock poate fi setat dar necesită de obicei privilegii ridicate – de obicei doar cu userul root
  - **CLOCK_MONOTONIC** – reprezintă un ceas sistem, nesetabil, monoton strict crescător, ce numără timpul dintr-un anumit moment necunoscut din trecut, de obicei fiind numărul de secunde/nanosecunde de la pornirea sistemului. Acest ceas nu este afectat de setarea ceasului sistem calendaristic dar poate suferi anumite ajustări când intervine NTP (Network Time Protocol). Acest clock nu se actualizează și pe perioadă când sistemul este in starea de suspend.
  - **CLOCK_MONOTONIC_COARSE** – reprezintă o variantă mult mai rapidă dar mai puțin exactă a lui CLOCK_MONOTONIC. Nu este disponibil pe toate sistemele, depinzând de arhitectură
  - **CLOCK_MONOTONIC_RAW** – reprezintă o variantă a lui CLOCK_MONOTONIC care nu este afectat nici de ajustările NTP și obține timpul numărat direct de hardware
  - **CLOCK_BOOTTIME** – reprezintă o variantă a lui CLOCK_MONOTINIC ce nu este afectată de starea de suspend a sistemului. Practic acest ceas este actualizat și atunci când sistemul este în starea de suspend.
  - **CLOCK_PROCESS_CPUTIME_ID** – acest clock măsoară cât timp procesor a consumat procesul incluzând timpul consumat de toate thread-urile procesului
  - **CLOCK_THREAD_CPUTIME_ID** – măsoară cât timp procesor a consumat thread-ul.

- Aceste MACRO-uri sunt folosite de mai multe funcții de bibliotecă și apeluri sistem pentru a se utiliza de un anumit ceas al sistemului

Reprezentarea timpului

Structuri de date de reprezentare a timpului

```
struct timespec {
               time_t tv_sec;   /* Seconds */
               time_t tv_nsec;  /* Nanoseconds [0, 999'999'999] */
};

#include <time.h>

typedef /* ... */  time_t;

#include <sys/types.h>

typedef /* ... */  suseconds_t;
typedef /* ... */  useconds_t;
```

```
struct tm {
    int         tm_sec;    /* Seconds          [0, 60] */
    int         tm_min;    /* Minutes          [0, 59] */
    int         tm_hour;   /* Hour             [0, 23] */
    int         tm_mday;   /* Day of the month [1, 31] */
    int         tm_mon;    /* Month            [0, 11]  (January = 0)
*/
    int         tm_year;   /* Year minus 1900 */
    int         tm_wday;   /* Day of the week  [0, 6]   (Sunday = 0) */
    int         tm_yday;   /* Day of the year  [0, 365] (Jan/01 = 0) */
    int         tm_isdst;  /* Daylight savings flag */
    long        tm_gmtoff; /* Seconds East of UTC */
    const char *tm_zone;   /* Timezone abbreviation */
};
```

- tip de date de un tip întreg, folosit pentru a reprezenta timpul sub forma de număr de secunde de la Epoch
- **`struct timespec`** – tip de date pentru reprezentarea unei perioade de timp în secunde și nanosecunde – folosit pentru reprezentarea rezoluțiilor diferitelor tipuri de ceasuri precum și pentru definirea intervalelor la timere
- **`struct tm`** – tip de date folosit pentru reprezentarea timpului în forma calendaristică

Reprezentarea timpului

Funcții de lucru si manipulare a structurilor de timp

Funcția mktime(…): `time_t mktime(struct tm *tm);`

- Funcția primește ca argument un pointer către o structură struct tm pe care o normalizează.
  Normalizare: calculul valorilor astfel încât să rezulte o dată calendaristică validă. Ex: 32 august va fi normalizat ca fiind 1 sept, 63 de minute va fi normalizat ca fiind 1h:03min
- Funcția returnează -1 în caz de eroare sau valoarea de timp de la Epoch ca datei calendaristice din structură dată ca și argument
- Atenție: parametrul tm este un parametru de intrare-ieșire.

Funcția localtime(…): struct tm *localtime(const time_t *timep);

- Primește ca argument un pointer către o valoare de timp de la Epoch și returnează o dată calendaristică validă sub forma unei structuri de tip struct tm
- Valoarea returnată reprezintă o structură alocată static și nu dinamic. Această valoare poate și va fi suprascrisă cu alte valori la apeluri succesive ale acestei funcții sau a altor funcții înrudite.
- OSB: Este greșit a se dealoca pointer-ul asignat cu valoarea returnată a acestei funcții

Funcția asctime(…): char *asctime(const struct tm *tm);

- Primește ca argument un pointer către o structură struct tm ce conține o dată calendaristică și returnează un string ce reprezintă data într-o formă textuală.
- Valoarea returnată este un string alocat static și nu dinamic astfel că este greșită utilizarea unei operații de free(…) asupra acestui pointer după apelul acestei funcții
- Atenție: această funcție nu normalizează data din tm

Reprezentarea timpului

Funcții de lucru si manipulare a structurilor de timp

Funcția clock_getres(…): `int clock_getres(clockid_t clockid, struct timespec *_Nullable res);`

- Funcția are rolul de a obține rezoluția unui ceas (clock). Rezoluția reprezintă durata de timp a unui tick ☾ diferența dintre două valori consecutive
- Primește ca și prim argument clock_id ce reprezintă identificatorul unui clock si poate fi una din valorile definite de macro-urile discutate anterior: CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_MONOTONIC_COARSE, ….)
- Primește ca și al doilea argument un pointer către o structură de timp struct timespec unde va stoca rezoluția clock-ului
- Retunează 0 în caz de succes și -1 în caz de eroare cu setarea errno cu codul de eroare corespunzător

Funcția clock_gettime(…): `int clock_gettime(clockid_t clockid, struct timespec *tp);`

- Funcția returnează valoarea ceasului la momentul apelului, reprezentat prin clockid, valoarea pe care o scrie în structura struct timespec referită de pointer-ul tp.
- Primește ca și prim argument clock_id ce reprezintă identificatorul unui clock si poate fi una din valorile definite de macro-urile discutate anterior: CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_MONOTONIC_COARSE, ….)
- Primește ca și al doilea argument un pointer către o structură de timp struct timespec unde va stoca valoarea clock-ului cerut
- Retunează 0 în caz de succes și -1 în caz de eroare cu setarea errno cu codul de eroare corespunzător

Funcția clock_settime(…): `int clock_gettime(clockid_t clockid, const struct timespec *tp);`

- Similară cu funcția gettime dar cu scopul de a seta clock-ul referit de clockid cu valoarea referită de tp.
- Retunează 0 în caz de succes și -1 în caz de eroare cu setarea errno cu codul de eroare corespunzător

Funcția clock(…): `clock_t clock(void);`

- Funcția returnează cât timp procesor a fost utilizat de programul apelant la momentul apelului
- Conform pagini de manual, pentru a se obține valoarea în secunde, valoarea returnată se va împărți la macro-ul: CLOCKS_PER_SEC – exprimă numărul de incrementări de clock pe secundă (clock ticks per second)
- Reprezintă un caz particular al apelului de clock_gettime(….) folosind clock-ul CLOCK_PROCESS_CPUTIME_ID

COD EXEMPLU: utilizare clock_getrest() pentru obținerea rezoluției pentru principalele tipuri de clock dintr-un sistem Linux: cod6-1

Reprezentarea timpului

POSIX Timer

- Timer-ul reprezintă un mecanism bazat pe un numărător care poate să genereze un eveniment la expirarea timpului setat. Se poate folosi pentru generarea de intervale de timp periodice, intervale de timeout, …
- În POSIX un timer se poate gestiona prin apeluri sistem dedicate. Fiecare timer este identificat printr-un ID, denumit in general timerid, unic la nivelul procesului. Clock-ul unui timer poate fi unul din cele descrise anterior reprezentate prin MACRO-uri precum: CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_MONOTONIC_COARSE …
- ID-ul unic, timerid este implementat prin tipul de date *time_t* – la

Crearea unui timer. Funcție timer_create:

```
int timer_create(clockid_t clockid,
                 struct sigevent *_Nullable restrict sevp,
                 timer_t *restrict timerid);
```

- Funcția timer_create este un apel sistem ce creează un timer la nivelul unui proces.
- Parametrul clockid reprezintă sursa de clock folosită pentru timer (un MACRO de clock)
- Parametrul sevp reprezintă o structură ce descrie comportamentul timer-ului în momentul expirării acestuia
- Parametrul timerid reprezintă un parametru de ieșire a apelului în care acesta va scrie ID-ul unic al timer-ului creat
- Funcția returnează 0 în caz de succes iar în caz de eroare -1 cu setarea variabilei errno cu codul de eroare corespunzător. În caz de eroare apelul nu scrie nimic în zone de memorie referită de timerid.
- Timer-ul nou creat nu are nici valoare de numărare și este oprit (disarmed). În modul de numărare (fired) timer-ul numără descrescător de la valoarea inițială spre 0 și, în funcție de comportamentul stabilit, va emite (sau nu) o notificare în momentul în care ajunge la valoarea 0

Reprezentarea timpului

POSIX Timer

Structura **`struct sigevent`**

- Se folosește în apelul funcției timer_create pentru a defini tipul de notificare în momentul expirării timer-ului
- Structura are următoarea formă (man 7 sigevent)

```
struct sigevent {
    int             sigev_notify;                   /* Notification type */
    int             sigev_signo;                    /* Signal number */
    union sigval    sigev_value;                    /* Signal value */
    void          (*sigev_notify_function)(union sigval);   /* Notification function */
    pthread_attr_t *sigev_notify_attributes;        /* Notification attributes */
    pid_t           sigev_notify_thread_id;         /* ID of thread to signal */
};
```

- Câmpul sigev_notify definește tipul notificării
  - SIGEV_NONE – nu se trimite nici o notificare în momentul expirării timer-ului
  - SIGEV_SIGNAL – se notifică procesul prin trimiterea unui semnal în momentul expirării timer-ului. În cazul în care se folosește flag-ul SA_SIGINFO la instanțierea tratării semnalului cu sigaction(…), conținutul câmpului sigev_value va fi transmis împreună cu semnalul în câmpul si_value al structurii siginfo_t. Câmpul este un union și astfel se poate trimite ori un întreg ori un pointer

```
union sigval {            /* Data passed with notification */
    int     sival_int;    /* Integer value */
    void   *sival_ptr;    /* Pointer value */
};
```

timer-ului se creează un thread a cărui cod este reprezentat de sigev_notify_function. Atributele noului thread creat sunt reprezentate de sigev_notify_attributes de tip pthread_attr_t

  - SIGEV_THREAD_ID – similar cu SIGEV_SIGNAL dar se trimite către un thread identificat prin ID-ul său din kernel (obținut prin apelul gettid()). ID-ul thread-ului se scrie în câmpul sigev_notify_thread_id.

Reprezentarea timpului

POSIX Timer

Setarea valorii unui timer. Funcția timer_settime:

```
int timer_settime(timer_t timerid,
                  int flags,
                  struct itimerspec *restrict new_value,
                  struct itimerspec *_Nullable restrict old_value);
```

Funcția timer_settime are rolul de a seta valoarea de numărare a unui timer. De asemenea, funcția are ca afect si armarea timer-ului, adică pornirea acestuia (timer-ul este oprit după ce a fost creat)

- Parametrul flags specifică anumite proprietăți ale timer-ului. De obicei se folosește cu 0 sau cu valoarea TIMER_ABSTIME
- Parametrul new_value – reprezintă un pointer către o structură de tip struct itimerspec și reprezintă noua valoarea de numărare a timer-ului
- Parametrul old_value – reprezintă un pointer către o structură de tip struct itimerspec în care funcția va scrie vechea valoarea de numărare a timer-ului înainte de fi suprascrisă cu valoarea new_value de către apelul curent

Structura struct itimerspec:

```
struct itimerspec {
    struct timespec  it_interval;  /* Interval for periodic timer */
    struct timespec  it_value;     /* Initial expiration */
};
```

- ui. Când timer-ul este pornit această valoarea va fi decrementată până la
- it_interval – dacă această valoare este setată, în momentul în care valoarea it_value ajunge la 0, timer-ul va folosi această valoare pentru a re-arma timer-ul și o copiază în valoarea it_value. Acest câmp poate fi folosit pentru a da un comportament recurent unui timer Dacă această valoare este 0 timer-ul va da un eveniment o singură data, va fi de tip one-shot

Reprezentarea timpului

POSIX Timer

Obținerea valorii curente a unui timer. Funcția timer_gettime:

```
int timer_settime(timer_t timerid, struct itimerspec * curr_value);
```

Funcția timer_gettime are rolul de a obține valoarea curentă a numărătorului timer-ului identificat prin timerid și o va scrie în structura referită de parametrul curr_time.
Funcția returnează 0 în caz de succes și -1 în caz de eroare cu setarea valorii errno

Ștergerea unui timer. Funcția timer_delete

```
int timer_delete(timer_t timerid);
```

- Funcția va șterge timer-ul referit prin timerid. Se recomandă ștergerea unui timer ce nu mai este folosit înainte de terminarea programului pentru evitarea situațiilor de tip memory-leaks și/sau pentru eliberarea resurselor folosite în kernel
- Funcția returnează 0 în caz de succes și -1 în caz de eroare cu setarea valorii ernno

Reprezentarea timpului

POSIX Timer

Exemplu de utilizare a timer-ului: cod6-2

În această secvență de cod se stabilește un timer ciclul cu rearmare automată care trimite semnalul SIGUSR1 la fiecare expirare. Semnalul este tratat prin printarea unui mesaj. Programul principal așteaptă un text de la stdin pentru terminarea programului prin apelul fgets(…).
Pentru ca acest comportament să funcționeze se setează flag-ul SA_RESTART la sigaction.

Teme, analiza, studiu individual:

- Transformați codul exemplu astfel încât la fiecare expirare de timer să se creeze un thread care să printeze mesajul la stdin dar fără a se modifica comportamentul timer-ului
- Transformați codul exemplu astfel încât la fiecare expirare de timer să se creeze un thread care să printeze mesajul la stdin prin modificarea comportamentului timer-ului cu SIGEV_THREAD.
- Transformați codul exemplu astfel încât semnalul trimis să se trimită către un thread anume prin folosirea SIGEV_THREAD_ID
- Transformați codul exemplu astfel încât la sigaction să se folosească câmpul sa_sigaction pe post de handler. Să se modifice codul astfel încât timer-ul să transmită o informație (variantă cu întreg dar și cu pointer – separat). Insepectați conținutul structurii siginfo_t trimisă handler-ului semnalului (prin printare la stdout)
- Transformați codul exemplu astfel încât să nu se folosească nici un mod de notificare (să se utilizeze în modul SIGEV_NONE). Pentru detectarea evenimentului când timer-ul ajunge la 0 se va face prin metoda polling cu verificarea valorii curente a timer-ului prin apelul timer_gettime(…) intr-o buclă while până când valoarea curentă a timer-ului este 0. Analizați și comparați utilizarea procesor a acestei variante și a codului exemplu

Secțiunea VII

Scheduling – Noțiuni de planificare de task-uri

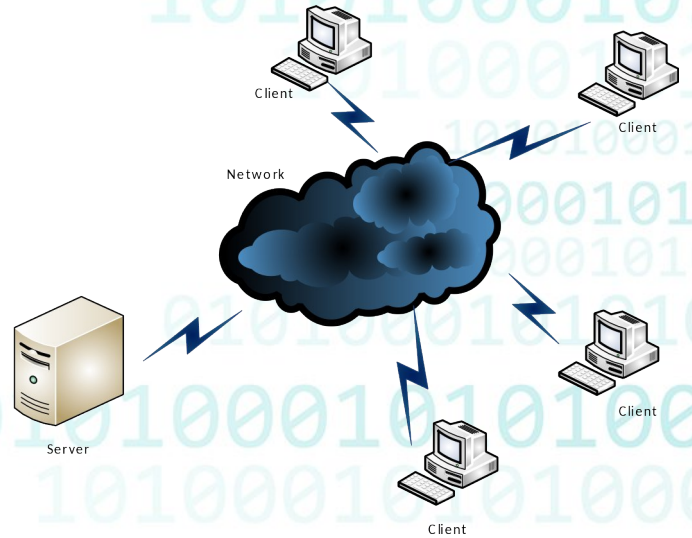
Secțiunea VIII

Aplicații distribuite. Comunicarea prin socket

Comunicarea prin socket

Socket. Definiții

- Socket – reprezintă un capăt (endpoint) de comunicație identificat printr-un descriptor asupra căruia se pot efectua operații de scriere și citire folosind apeluri sistem. Operațiile de scriere și citire se vor translata mai departe în operații specifice protocoalelor de comunicație folosite mai departe (i.e. TCP, UDP, X.25, …)
- Se folosește pentru comunicarea între procese
- Există mai multe tipui de socket, dar cele mai utilizate sunt UNIX socket (UNIX domain) sau network socket (Internet domain).
  - UNIX socket – mai multe procese de pe aceeași mașină pot comunica printr-un fișier de tip UNIX socket creat în sistemul de fișiere. Adresa într-o comunicație de tip UNIX socket este reprezentată de o cale din sistemul de fișiere
  - Network socket – mai multe procese de pe mașini diferite, conectate la o rețea pot comunica prin protocoale de rețea. Adresa într-o comuinicație de tip Internet socket este reprezentată printr-o adresa IP și port
- Modelul de comunicație este modelul client-server
  - Server – reprezintă o abstractizare a unui serviciu furnizat unor client
  - Client – reprezintă consumatorul serviciului oferit de server
  - Comunicarea se realizează după paradigma request-response:
    Clientul se conectează la server
    Clientul face o cerere către server (request) cu un mesaj
    Serverul procesează și validează cererea clientului
    Serverul răspunde cererii cu un mesaj pe post de răspuns (response)
    Clientul se deconectează de la server

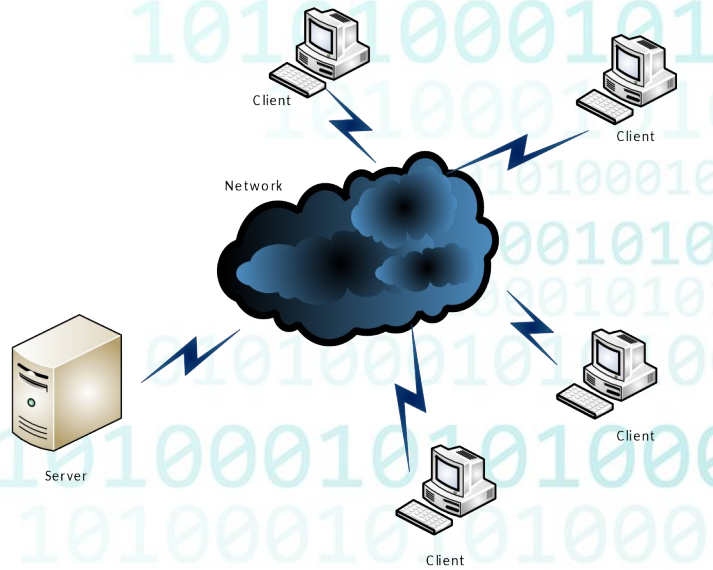


Comunicarea prin socket

Socket. Definiții

- **Internet socket (Network Socket)** – se folosește când se dorește o comunicare între mai multe procese aflate pe mașini fizice diferite
- Modelul de comunicare este client – server
- Protocolul de rețea folosit: IP – TCP/UDP

- Socket-ul serverului se atașează (bind) de una din adresele IP prezente în sistem și ascultă (listen) conexiuni pe un anumit port (număr întreg între 0 și 65535)
- Clientul creează și el un socket cu care se conectează la adresa server-ului pe portul pe ce acesta ascultă
- Când clientul s-a conectat, server-ul acceptă conexiunea (accept) și creează un nou socket ce va reprezenta end-point-ul propriu zis de comunicare cu clientul



Comunicarea prin socket

Socket TCP

Utilitarul nc – netcat - TCP/IP swiss army knife

- Permite crearea unui server sau client pe TCP sau UDP și transferul de date text între sistemele conectate
- Pagina de manual: man 1 nc
- Parametrii:
  - -l – listen – crează un server TCP – dacă lipsește va crea un client TCP
  - -v – verbose – printează mesaje despre conexiuni
  - -p <port> – specifică portul
  - -s <ip> – specifică adresa IP pentru bind

- Server TCP

```
nc -l -p 3000 -s 127.0.0.1 -v
```

à Se creează un server TCP pe portul 3000 cu bind pe adresa de localhost 127.0.0.1

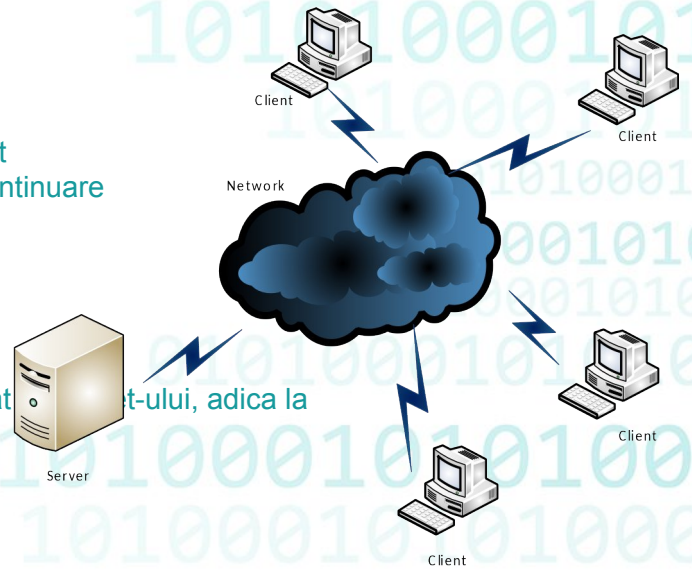
à După execuția comenzii tot ceea ce se scrie la stdin se va trimite pe socket, evident după o conexiune cu un client. Tot ceea ce se va primi de la client se va afișa în continuare la stdout

- Client TCP

```
nc 127.0.0.1 3000
```

à Se creaza un client TCP care se va conecta la adresa de localhost 127.0.0.1 pe portul 3000

à După execuția comenzii tot ceea ce se va scrie la stdin se va trimite la celalalt capat [illegible]t-ului, adica la server. Tot ceea ce se va primi de la server se va afișa în continuare la stdout



Comunicarea prin socket

Socket TCP

Utilitarul netstat

- Permite afișarea socket-ilor, conexiunilor active, tabelele de rutare din sistem precum și alte informații și statistici despre interfețele de rețea
- Pagina de manual: man 1 netstat
- Parametrii importanți:
  - -a – afișarea tuturor socket-ilor deschiși
  - -n – afișarea adreselor și a porturilor sub formă numerică în loc de interpretarea lor textuală
  - -p – afișarea PID-ului procesului ce folosește socket-ul
  - -t – toate conexiunile și sockets TCP
  - -u – toate conexiunile și scokets UDP

Comunicarea prin socket

Socket TCP

Utilizarea de socket TCP client-server folosind POSIX API

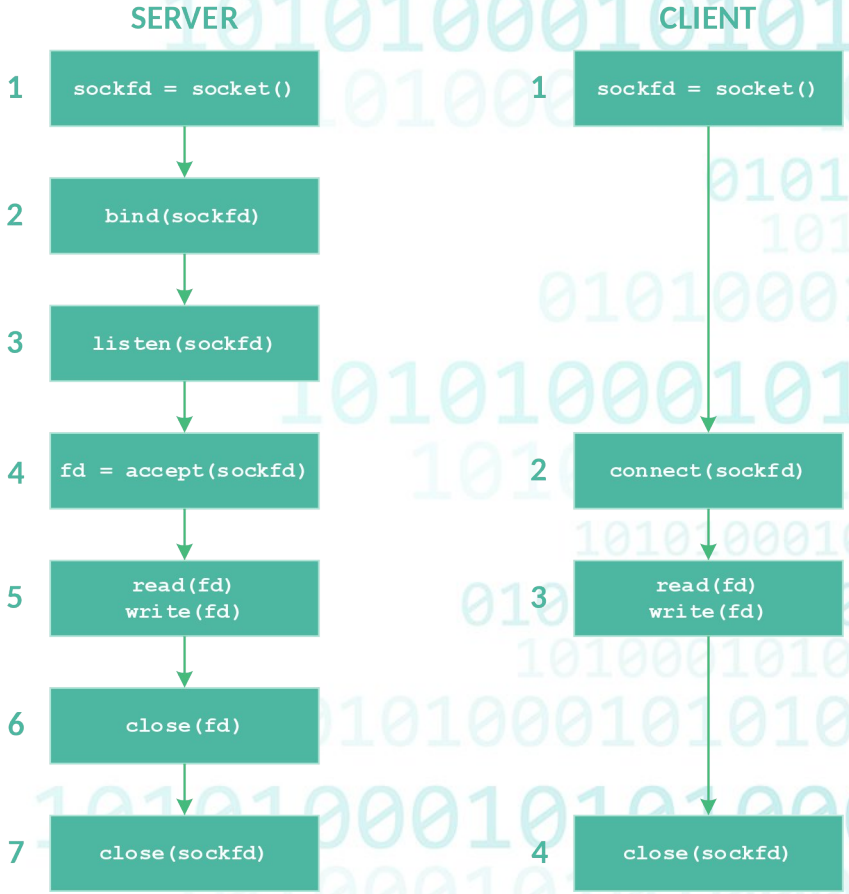
- Funcții: socket(), bind(), listen(), accept(), connect(), read(), write(), close()

SERVER

1. Se creează un socket ce va fi returnat în descriptorul sockfd
2. Se atașează socket-ul creat, prin descriptorul sockfd, unei adrese IP existente (a unei interfețe de rețea activă) și unui port
3. Se marchează socket-ul ca fiind pasiv în sensul că va fi folosit doar pentru a “asculta” și a accepta conexiuni și nu va fi folosit efectiv pentru transfer de date. Tot prin acest apel se stabilește și dimensiunea cozii de așteptare de clienți
4. Se așteaptă conexiuni. Apelul se va bloca până la apariția unei conexiuni. La apariția unei noi conexiuni funcția va returna un descriptor ce va fi folosit efectiv pentru transferul
   de date
5. Folosind apeluri sistem precum read() write() se efectuează transferurile de date necesare
6. Se închide conexiunea prin închiderea descriptorului returnat de accept()
7. La final, se închide socket-ul deschis inițial de server

CLIENT

1. Se creează un socket ce va fi returnat în descriptor sockfd. Acest descriptor va fi folosit pentru transferul de date efectiv
2. Se apelează connect folosind descriptorul creat și datele de identificare a serverului în rețea (adresa IP și port).
3. Folosind socket-ul creat inițial se vor efectua apelurile sistem read() write() pentru a realiza transferurile de date cu serverul
4. Se va inchide conexiunea prin închiderea descriptorului creat inițial



Comunicarea prin socket

Socket TCP

Apelul sistem connect()

```
int socket(int domain, int type, int protocol);
```

- Permite crearea unui socket ce poate fi folosit atât în componenta de tip server cât și în componenta de tip client.
- Pagina de manual: man 2 socket
- Parametrul domain specifică domeniul de comunicație. Se folosesc niște macro-uri specializate cum ar fi:
  - AF_UNIX – UNIX socket – comunicație locală pe aceeași mașină
  - AF_INET – socket IPv4 – comunicație TCP/UDP pe IPv4
  - AF_INET6 – socket IPv6 – comuinicație TCP/UDP pe IPv6
- Parametrul type reprezintă tipul de socket. Se folosesc niște macro-uri specializate cum ar fi:
  SOCK_STREAM – comunicație bidirecțională, sigură cu asigurarea secvențierii corecte a pachetelor de date. Folosit pentru comunicație orientată pe conexiuni (TCP) – connection-based communication
  SOCK_DGRAM – comunicație fără asigurarea unei secvențe corecta a pachetelor de date. Pachetele de date sunt de dimensiune maximă fixă. Comunicarea este nesigură și nu se garantează și ajungerea mesajelor la destinație. Comunicația nu e este orientată pe conexiuni (UDP) – connectionless comunication
- Parametrul protocol specifică dacă există vreun protocol special ce se aplică socket-ului. În mod uzual se folosește valoarea 0
- Returnează un descriptor ce reprezintă socket-ul creat. La componenta client acest descriptor se folosește efectiv pentru transferul de date pe când la componenta server se folosește doar pentru a aștepta conexiuni, pentru transferul de date folosindu-se un descriptor adițional

Comunicarea prin socket

Socket TCP

Apelul sistem connect()

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

- Permite conectarea unui socket de tip client la un server specificat prin adresa dată de addr de lungime addrlen.
- Pagina de manual: man 2 connect
- Se folosește doar în componenta de tip client
- Parametrul sockfd reprezintă un socket format inițial printr-un apel socket()
- Parametrul addr reprezintă un pointer către o structură de tip struct sockaddr. Această structură este generică și în practică nu se folosește în mod explicit. Se poate folosi prin typecast și un pointer către o structură de unul din tipurile următoare în funcție de tipul de conxiune (domain) specificat în apelul sistem socket()

socket domain	sockaddr structure	Addrlen
AF_INET	struct sockaddr_in	sizeof(struct sockaddr_in)
AF_INET6	struct sockaddr_in6	sizeof(struct sockaddr_in6)
AF_UNIX	struct sockaddr_un	Sizeof(struct sockaddr_un)

- Parametrul addren reprezint
- Pentru componenta client, a server-ului la care se dorește să se realizeze conexiunea

Comunicarea prin socket

Socket TCP

Structurile struct sockaddr – structura sockaddr_in

```
struct sockaddr_in {
        sa_family_t     sin_family;     /* AF_INET */
        in_port_t       sin_port;       /* Port number */
        struct in_addr  sin_addr;       /* IPv4 address */
    };
```

- sa_family sin_family – va lua obligatoriu valoarea AF_INET
- sin_port – reprezintă portul – un număr întreg pe 16 biți fără semn. Nu se scrie explicit cu o valoarea de port ci doar prin intermediul funcțiilor de convesie precum htons, htonl. Aceste funcții convertesc un număr din reprezentarea endianess a host-ului în reprezentarea folosită în rețea, aceasta fiind MSB first
- sin_addr – adresa ip encodată ca un număr întreg. Se poate transforma din reprezentarea ca string în reprezentarea necesară folosind funcții ajutătoare precum: inet_addr
- Se recomandă inițializarea cu 0 întregii structuri înainte de utilizare
- Pagina de manual: man struct sockaddr
- Exemplu de atribuire a câmpurilor:

```
struct sockaddr_in my_addr_struct;
memset(&my_addr_struct, 0, sizeof(struct sockaddr_in);
my_addr_struct.sin_family = AF_INET;
my_addr_struct.sin_port = htons(4555);
my_addr_struct.sin_addr = inet_addr("127.0.0.1");
```

Comunicarea prin socket

Socket TCP

Exemplu de realizarea a unei componente client tcp – se conectează la un server pe localhost (adresa 127.0.0.1) pe portul 4555. Programul citește apoi de la intrarea standard câte o linie și o trimite peste conexiune la server. Programul se oprește la închiderea intrării standard. Comunicarea este implementată doar unidirecțional de la client spre server. Programul client nu citește date de pe socket.

Pentru testare, înainte de rularea programului se lansează un server tcp pe adresa localhost pe portul 4555 folosind utilitarul netcat astfel:

```
nc -l -p 4555 -s 127.0.0.1 -v
```

După ce serverul a fost lansat folosind utilitarul netcat se lansează programul exemplu. La lansarea programului exemplu în momentul realizării conexiunii cu serverul, utilitarul netcat va printa la ieșirea standard un mesaj. În continuare, toate mesajele trimise de pe consola programului exemplu client vor ajunge la serverul gestionat de netcat. Comunicația funcționează doar în acest sens în sens invers nefiind implementată în codul exemplu

Codul exemplu: cod7-1.c

Comunicarea prin socket

Socket TCP

Apelul sistem bind()

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

- Se folosește doar în componenta server
- Permite asignarea la o adresa existentă în sistem a unui socket creat în prealabil anterior cu apelul sistem socket(). Un socket este necesar să se asigneze la o adresă ip deja existentă în sistem, adica la o adresa ip asignată unei plăci de rețea din sistem (iar placa de rețea să fie conectată folosind adresa ip respectivă)
- Pe lângă o adresă ip a unei plăci de rețea fizice se poate folosi și adresa de localhost (127.0.01). Limitarea în acest caz este că astfel se pot realiza doar conexiuni interne pe aceeași mașină fizică, adresa de localhost nefiind vizibilă în exteriorul acesteia
- Adresa de ip se specifică printr-o structură din familia struct sockaddr cu semnificațiile prezentate anterior.
- Adresa de ip specificată prin structura addr trimisă apelului bind poate fi înlocuită și cu MACRO-ul INADDR_ANY care specifică adresa ip generică 0.0.0.0 astfel că socket-ul se va asigna dinamic la toate adresele ip existente în sistem (practic serverul nou creat va asculta pe toate adresele)
- Pagina de manual: man 2 bind
- Parametrul sockfd reprezintă un socket format inițial printr-un apel socket()
- Exemplu de asignare a unei structuri pentru bind() pe toate adresele pe portul 4555, serverul va asculta pe toate adresele pe portul 4555

```
struct sockaddr_in my_server_bind;
memset(&my_server_bind, 0, sizeof(struct sockaddr_in);
my_server_bind.sin_family = AF_INET;
my_server_bind.sin_port = htons(4555);
my_server_bind.sin_addr = INADDR_ANY;
```

Comunicarea prin socket

Socket TCP

Apelul sistem listen()

```
int listen(int sockfd, int backlog);
```

- Se folosește doar în componenta server după efectuarea apelului sistem bind()
- Are rolul de a marca socket-ul sockfd ca fiind un socket pasiv ce nu va fi folosit pentru transfer de date ci va fi folosit pentru acceptarea de conexiuni venite de la clienți
- Parametrul backlog este un intreg reprezintă dimensiunea maximă a cozii de așteptare a clienților. Dacă un client se conectează la server acesta va intra în coada de așteptare până la realizarea efectivă a conexiunii. Dacă în schimb coada de așteptare definită de acest parametru este plină, în momentul în care un client se conectează acesta va fi automat respins de către server și deconectat
- Pagina de manual: man 2 listen
- Parametrul sockfd reprezintă un socket format inițial printr-un apel socket()

Comunicarea prin socket

Socket TCP

Apelul sistem accept()

```
int accept(int sockfd, struct sockaddr *_Nullable restrict addr, socklen_t *_Nullable restrict addrlen);
```

- Se folosește doar în componenta server după efectuarea apelului sistem listen() și doar pentru conexiuni TCP de tipul SOCK_STREAM
- Are rolul de a aștepta conexiuni din partea clienților. Apelul sistem de blochează până când un client realizează o conexiune.
- Apelul folosește un socket (sockfd) creat (socket()), asignat unei adrese (bind())și marcat ca și pasiv (listen())
- În momentul în care se realizează o conexiune din partea unui client, apelul returnează un nou descriptor ce va fi folosit exclusiv pentru transferul de date dintre server și clientul conectat. De asemenea, în parametrul addr apelul accept() va scrie adresa ip a clientului ce s-a conectat și portul efemer sursă al acestui iar în parametrul addrlen va scrie dimensiunea în bytes a datelor scrise în structură.
- Parametrii addr și addrlen se interpretează ca și în cazurile anterioare.
- Returnează un nou descriptor ce va folosit pentru transferul de date prin apelurile sistem read() și write(). Este necesar ca acest descriptor să fie închis cu apelul sistem close() când se dorește terminarea conexiunii
- Pagina de manual: man 2 accept

Comunicarea prin socket

Socket TCP

Exemplu de realizarea a unei componente server tcp – se instanțiază un server TCP care ascultă pe toate adresele din sistem pe portul 4555. În momentul în care se realizează o conexiunne programul printează la ieșirea standard un mesaj ce conține adresa ip a clientului ce s-a conectat

Programul citește apoi de la intrarea standard câte o linie și o trimite peste conexiune la server. Programul se oprește la închiderea intrării standard. Comunicarea este implementată doar unidirecțional de la client spre server. Programul u citește date de pe socket.

Pentru testare, după rularea programului se lansează un client tcp pe adresa localhost pe portul 4555 folosind utilitarul netcat astfel:

```
nc 127.0.0.1 4555
```

În momentul în care utilitarul netcat s-a conectat la serverul implementat în acest cod exemplu, programul va printa la ieșirea standard un mesaj ce conține adresa ip a clientului conectat. În continuare, toate mesajele trimise de pe consola programului exemplu server vor ajunge la clientul gestionat de netcat. Comunicația funcționează doar în acest sens în sens invers nefiind implementată în codul exemplu.

Codul exemplu: cod7-2.c

Comunicarea prin socket

Socket TCP

ATENTIE: Exemplele realizate sunt doar demonstrative si au multe limitări precum:

- Nu tratează decât transmisia
- Tratează doar o singură conexiuni
- În mod obișnuit atât un client cât și un server (mai ales) este necesar să fie implementate fie multioproces fie multithreading

RECOMANDĂRI: Adăugați implementărilor elemente lipsă:

- Serverul necesită o implementare multithreading sau multiproces. Exemplu: in momentul în care se realizează o nouă conexiune aceasa va fi tratată de un thread/proces separat
- Pentru fiecare conexiune diferită se pot implementa câte două procese/threaduri diferite: unul pentru recepție și unul pentru transmisie
- Partea de accept() din server se recomandă să fie implementată printr-un proces diferit (sau thread)
- Pe partea de client se recomandă tot o implementare mutlithreading sau multiproces în momentul în care se realizează o conexiune.