

# Tutoriat 2 (Sisteme de Operare)

## Procese

### Contents

<b>1</b>	<b>Notiuni introductive</b>	<b>2</b>
<b>2</b>	<b>Planificarea proceselor (process scheduling)</b>	<b>4</b>
2.1	Cozi de planificare (scheduling queues) . . . . .	4
<b>3</b>	<b>Lucrul cu procese in terminal si in C</b>	<b>5</b>
3.1	Listarea informatiilor despre procese in terminal . . . . .	5
3.2	Crearea unui proces . . . . .	6
3.3	Inlocuirea unui proces cu un alt program . . . . .	9
<b>4</b>	<b>Comunicarea intre procese (IPC)</b>	<b>9</b>
4.1	Memorie partajata . . . . .	10
4.2	Message passing . . . . .	11
4.2.1	Naming . . . . .	11
4.2.2	Synchronization . . . . .	12
4.2.3	Buffering . . . . .	12
4.2.4	Pipes . . . . .	13
4.2.5	Sockets . . . . .	14

# 1 Notiuni introductive

Un **program** de sine statator este un fisier executabil de pe disk (o *entitate pasiva*). Un **proces** este un **program in executie** (un fisier executabil incarcata in memorie; o *entitate activa*), avand un **program counter** (un registru ce retine adresa urmatoarei instructiuni de executat) si o multime de resurse asociate (cat timp ii este alocat CPU-ul, memorie, fisiere, dispozitive I/O). Majoritatea sistemelor de operare atribuie proceselor niste valori intregi unice, numite **process identifiers (PIDs)**.

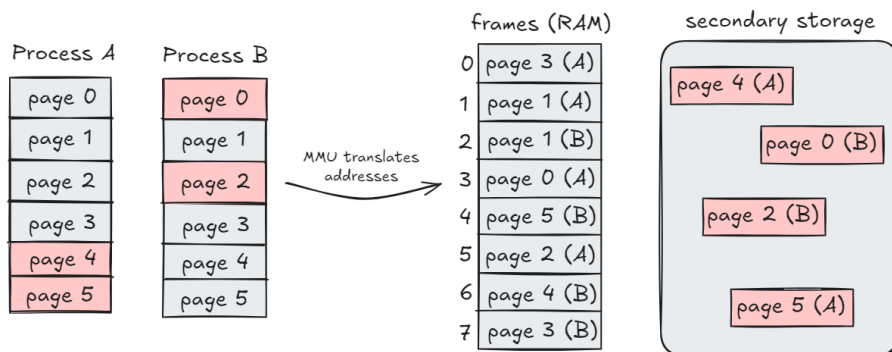


Figure 1: Maparea adreselor virtuale la adrese fizice

O **adresa de memorie fizica** reprezinta o locatie in RAM. O **adresa de memorie virtuala** este o zona *teoretica* de memorie (nu exista fizic), specifica unui proces. **Spatiul de adresa** al unui proces reprezinta gama de *adrese virtuale* cu care lucreaza. Cum adresele virtuale pentru fiecare proces incep de la 0, inseamna ca o anumita adresa **X** poate fi gasita la mai multe procese, dar este unica pentru fiecare proces in parte.

Un **page** este un bloc de dimensiune fixa de *memorie virtuala*, iar un **frame** este un bloc de dimensiune fixa de *RAM*; acestea au aceeasi dimensiune (4KB, 16KB, etc.), deoarece frame-urile sunt utilizate pentru a stoca page-uri. **Spatiul de adresa** al unui proces este impartit in page-uri.

Adresele virtuale sunt folosite cu scopul de a simplifica calculele; procesele vor avea impresia ca le sunt asociate blocuri neintrerupte de memorie. In realitate, multe din page-urile unui proces sunt distribuite peste tot in frame-uri, iar atunci cand nu mai este loc in RAM, restul page-urilor sunt stocate pe disc. Cateodata, anumite page-uri din RAM ale unui proces nu sunt utilizate frecvent, fiind interschimbate cu alte page-uri din secondary storage.

**MMU (memory management unit)** este o componenta hardware care se ocupa de traducerea adreselor virtuale in adrese fizice. Modul in care functioneaza, impreuna cu alte concepte (**TLB** = Translation Lookaside Buffer, **page table**, **page faults**, etc.) vor fi discutate pe viitor la alte tutoriate.

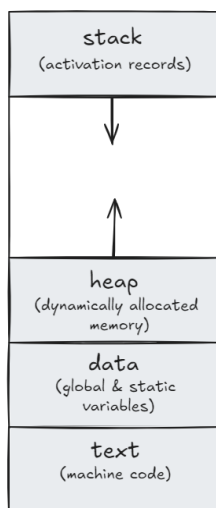


Figure 2: Structura unui proces in memorie

Cand se apeleaza o functie, este alocat un bloc de memorie numit **activation record**, unde sunt stocate toate datele necesare pentru executia apelului respectiv. Aceste date includ *parametrii functiei*, *variabilele locale* si *adresa de retur* (adresa de memorie a instructiunii unde se intoarce programul odata ce se incheie apelul respectiv).

Un proces are urmatoarea structura:

- **Stack (dimensiune dinamica):** contine **activation records**. La fiecare apel de functie se face *push*, iar la finalizarea unui apel se face *pop*.
- **Heap (dimensiune dinamica):** memoria alocata si dealocata dinamic la runtime.
- **Data section (dimensiune fixa):** variabilele globale si statice initializate.
- **Text section (dimensiune fixa):** codul masina compilat al programului respectiv.

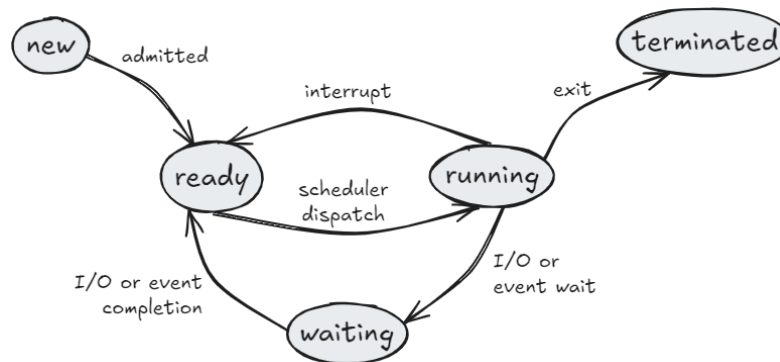


Figure 3: Diagrama starilor unui proces

In functie de ce se intampla cu un proces, acesta se poate afla in urmatoarele stari:

- **New:** tocmai a fost creat.
- **Ready:** este pregatit de executie.
- **Running:** a fost selectat de catre *CPU scheduler* sa se execute (vom discuta imediat despre scheduler).
- **Waiting:** a avut loc o cerere I/O sau un anumit eveniment, pentru care este nevoit sa astepte.
- **Terminated:** si-a finalizat executia, iar resursele sale pot fi eliberate de catre sistemul de operare.

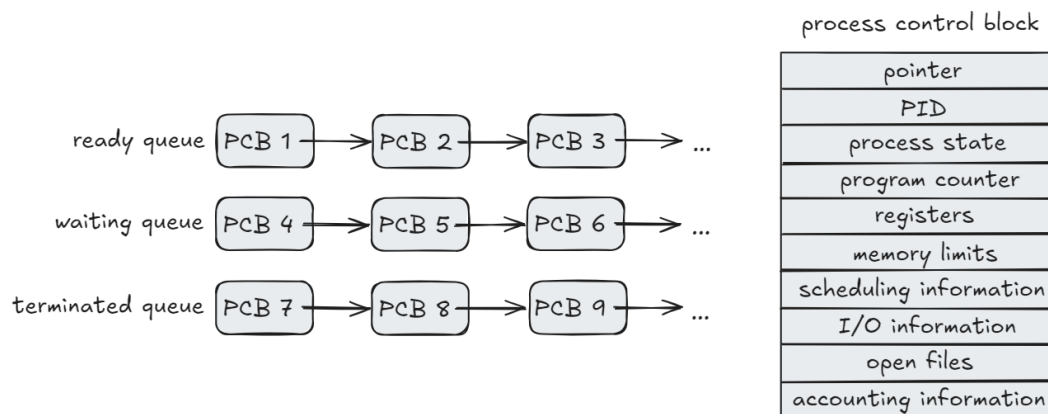


Figure 4: Structura unui PCB si niste exemple de cozi de PCB-uri

Un **PCB (process control block)** / **TCB (task control block)** este o *structura de date* folosita de catre sistemul de operare pentru a retine toate informatiile legate de un proces, inclusiv starea procesului, PID-ul,

program counter-ul, valorile registrilor CPU-ului, memoria alocata, planificarea executiei (prioritatea procesului, cozi de planificare, etc.), informatii I/O (fișiere deschise, dispozitive I/O alocate procesului), si informatii generale (timpul de executie pe CPU, limite de timp, etc.).

Cand un proces este creat, PCB-ul asociat este creat si plasat in memorie intr-o zona ce nu poate fi accesata de catre un utilizator normal. Pe masura ce se executa procesul, informatiile din PCB sunt actualizate (cum ar fi program counter-ul, starea procesului, etc.). Cand un proces este scos temporar de pe CPU, se efectueaza un **state save** (PCB-ul retine toate informatiile despre proces in momentul respectiv); odata ce procesul isi reia executia, se efectueaza un **state restore** (valorile registrilor si alte setari sunt readuse, luate din PCB).

Exista diverse *cozi* utilizate de catre OS pentru a gestiona anumite tipuri de procese. De exemplu, lista de procese care si-au finalizat executia si doresc sa isi elibereze resursele (**terminated queue**), sau lista de procese pregatite de executie, ce asteapta sa fie alocate CPU-ului (**ready queue**).

## 2 Planificarea proceselor (process scheduling)

Fiecare nucleu (core) al unui CPU poate executa o singura instructiune la un moment dat. Pentru a optimiza utilizarea resurselor, trebuie mereu sa fie un proces in executie. Cum unele procese sunt mai urgente decat altele, si de obicei exista mai multe procese decat nuclee libere, unele procese vor fi nevoite sa astepte pana la eliberarea unui nucleu; astfel, este necesara o planificare adecvata a executiei proceselor. Aceasta sarcina este indatorata unui program de sistem numit **CPU scheduler**.

Procesele au o felie de timp (masurata in milisecunde) pentru care au voie sa se execute pe CPU. Cand acest timp expira, sau procesul este temporar blocat (din cauza unei cereri I/O), sau si-a finalizat executia, **scheduler-ul** are rolul de a acorda nucleul respectiv unui alt proces aflat in asteptare.

Un scheduler poate fi **preemptive** (poate intrerupe in mod fortat un proces care se executa, chiar daca nu si-a finalizat executia) sau **nonpreemptive** (cand un proces este asignat CPU-ului, se executa pana la capat, cu exceptia cazului in care are loc o cerere I/O).

Procesele pot fi impartite in **I/O-bound** (petrec mai mult timp efectuand operatii I/O decat calcule) si **CPU-bound** (invers). Numarul de procese aflate la momentul curent in memorie, care sunt pregatite de executie, se numeste **degree of multiprogramming**.

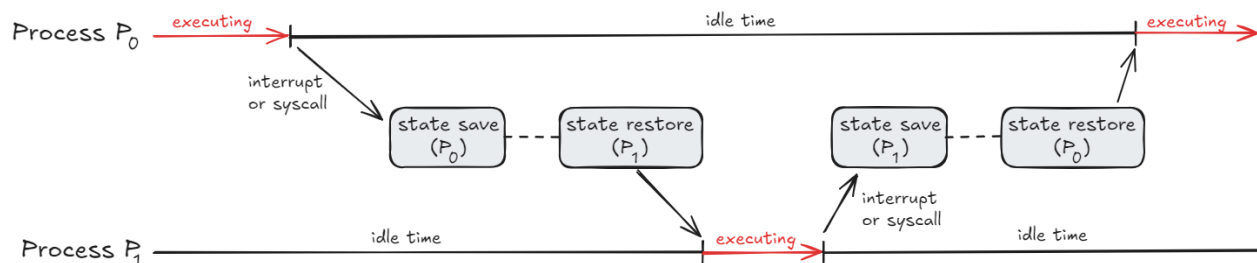


Figure 5: Context switch intre doua procese

Un **context switch** este atunci cand CPU-ul comuta intre procese. Se efectueaza un *state save* al procesului curent, dupa care are loc un *state restore* al celui alt proces si se reia din nou executia. Timpul si resursele consumate pentru realizarea unui context switch poarta numele de **context switch overhead**; prea multe context switch-uri ar putea incetini performanta sistemului, iar overhead-ul variaza de la un sistem de operare la altul.

### 2.1 Cozi de planificare (scheduling queues)

Un **scheduling queue** este o structura de date folosita de catre OS pentru a gestiona procesele care asteapta o anumita resursa. Acestea sunt cozi de PCB-uri pentru procese care se afla in anumite stari. Cateva exemple:

- **Ready queue:** procesele care sunt pregatite si doresc sa se execute pe CPU.
- **Wait queue:** procesele temporar blocate, asteptand finalizarea unei cereri I/O sau altui eveniment.
- **Job queue:** o coada mai generala, ce contine toate procesele din sistem.

- **Suspended queue:** procesele care au fost temporar scoase din RAM si stocate pe disk, din cauza lipsei de memorie.

Cand un proces se executa pe CPU, se poate intampla unul din urmatoarele lucruri:

- Are loc o cerere I/O, plasand procesul intr-un **I/O wait queue**. La finalizarea cererii, procesul va fi transferat din **wait queue** in **ready queue**.
- Procesul va crea un **proces copil (child process)**, vom discuta imediat despre acest concept); cat timp asteapta sa isi finalizeze copilul executia, **procesul parinte (parent process)** este plasat intr-un **wait queue**.
- Din cauza unei intreruperi (sau pentru ca si-a finalizat portia de timp), procesul este scos de pe CPU si plasat in **ready queue**.

### 3 Lucrul cu procese in terminal si in C

Un proces are capacitatea de a crea alte procese. Astfel, procesul principal se numeste **proces parinte (parent process)**, iar copiii sai sunt **procesele copil (child processes)**, formand un **arbore de procese (process tree)**. Un copil isi poate obtine resursele in mod direct de la OS, sau poate fi constrans la o submultime a resurselor parintelui. Procesul parinte s-ar putea sa fie nevoit sa isi imparta resursele intre copii, sau sa ofere o submultime comuna de resurse tuturor copiilor.

#### 3.1 Listarea informatiilor despre procese in terminal

Sunt cateva comenzi utile pentru listarea detaliata (sau mai putin detaliata) a informatiilor in legatura cu anumite procese (sau toate procesele) ale sistemului. De exemplu, **ps -l** (unde "ps" = process status, "-l" = long format) afiseaza informatii despre toate procesele asociate cu sesiunea curenta (terminal-ul curent):

```
harapalb@ubuntu:~/Desktop$ ps -l
F S  UID      PID     PPID  C PRI   NI ADDR SZ WCHAN  TTY          TIME CMD
0 S   1000     3568     3561  0  80    0  -  4983 do_wai pts/0        00:00:00 bash
4 R   1000     5932     3568 99  80    0  -  5612 -          pts/0        00:00:00 ps
```

Comanda poate fi rulata si fara flag-ul "-l", dar va afisa mai putine informatii. Output-ul de mai sus consta in afisarea PID-ului, starii procesului, prioritatii, numarului de pages, etc. Aceste informatii pot fi afisate despre fiecare proces existent in sistem, ruland **ps -el** (unde "-e" = extended).

```
harapalb@ubuntu:~/Desktop$ ps -el
F S  UID      PID     PPID  C PRI   NI ADDR SZ WCHAN  TTY          TIME CMD
4 S   0         1         0  0  80    0  -  5810 -          ?           00:00:05 systemd
1 S   0         2         0  0  80    0  -    0 -          ?           00:00:00 kthreadd
1 S   0         3         2  0  80    0  -    0 -          ?           00:00:00 pool_workqueue_re
1 I   0         4         2  0  60 -20 -    0 -          ?           00:00:00 kworker/R-rcu_g
1 I   0         5         2  0  60 -20 -    0 -          ?           00:00:00 kworker/R-rcu_p
1 I   0         6         2  0  60 -20 -    0 -          ?           00:00:00 kworker/R-slub_
1 I   0         7         2  0  60 -20 -    0 -          ?           00:00:00 kworker/R-netns
1 I   0         9         2  0  80    0  -    0 -          ?           00:00:00 kworker/0:1-mm_pe
1 I   0        12         2  0  60 -20 -    0 -          ?           00:00:00 kworker/R-mm_pe
1 I   0        13         2  0  80    0  -    0 -          ?           00:00:00 rcu_tasks_kthread
1 I   0        14         2  0  80    0  -    0 -          ?           00:00:00 rcu_tasks_rude_kt
1 I   0        15         2  0  80    0  -    0 -          ?           00:00:00 rcu_tasks_trace_k
```

Pentru a vizualiza arborescenta proceselor, se poate folosi **pstree**. Dupa cum se poate observa, **systemd** este primul proces creat ce ruleaza cand este pornit calculatorul. De asemenea, exista anumite procese ce poarta numele de "**daemons**"; acestea sunt **procesele de fundal (background processes)**; ruleaza pe fundal, fara sa necesite vreo interactiune directa cu utilizatorul) ce opereaza continuu si au rolul de a automatiza anumite sarcini.

```

harapal@ubuntu:~/Desktop$ pstree
systemd--ModemManager--3*[{ModemManager}]
--NetworkManager--3*[{NetworkManager}]
--accounts-daemon--3*[{accounts-daemon}]
--avahi-daemon--avahi-daemon
--colord--3*[{colord}]
--containerd--10*[{containerd}]
--cron
--cups-browsed--3*[{cups-browsed}]
--cupsd
--dbus-daemon
--dockerd--21*[{dockerd}]
--gdm3--gdm-session-wor--gdm-wayland-ses--gnome-session-b--3*[{gnome-session-b}]+
--3*[{gdm3}]--3*[{gdm-session-wor}]--3*[{gdm-wayland-ses}]
--gnome-remote-de--3*[{gnome-remote-de}]
--2*[{kerneloops}]

```

## 3.2 Crearea unui proces

În primul rând, funcția **fork** este folosită pentru crearea unui proces copil, identic cu părintele, dar care va avea propriul său PID. Pentru procesul apelant (părinte), valoarea returnată de funcție este PID-ul (adică  $> 0$ ); pentru copil, valoarea este nulă ( $= 0$ ), iar în cazul unei erori, se întoarce o valoare negativă ( $< 0$ ).

Scopul funcției **exit** este de a opri executia unui proces, restituind controlul host-ului (părintelui), și semnaland succes sau eșec prin intermediul unui cod de exit. Nu este neapărat necesară includerea acestei funcții în codul unui proces copil; totuși, excluderea ei ar putea cauza anumite probleme, ce vor fi discutate imediat. Odată ce un proces își finalizează executia și resursele sale sunt recuperate de către OS, intrarea sa tot va rămâne în *tabelul de procese*, până când părintele sau apelează funcția *wait*.

Funcția **wait** este folosită de către părinte pentru a-l forța să aștepte până când unul (sau mai mulți) din copiii săi își finalizează executia, ulterior eliberându-le resursele. De asemenea, părintele poate primi codul transmis de funcția *exit* prin intermediul funcției *wait*. Dacă părintele nu apelează *wait* și rulează în continuare în timp ce copiii săi și-au finalizat executia, atunci copiii devin **proces zombie (zombie processes)**; dacă părintele nu apelează **wait** și își finalizează executia înaintea copiilor săi, atunci copiii devin **proces orfan (orphan processes)**.

Pentru a gestiona problema **proceselor orfane**, există procese ce au responsabilitatea de a deveni părinții acestora. Totuși, anumite sisteme nu permit proceselor copil să existe dacă părinții își finalizează executia; astfel, are loc o **terminare în cascada (cascading termination)**, eliminând arbori întregi de procese.

```

1  /* Creating one child process (v1) */
2
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <stdlib.h>
6  #include <sys/wait.h>
7  #include <errno.h>
8
9  int main() {
10     pid_t pid = fork();
11
12     if (pid < 0) {
13         return errno;
14     }
15     else if (pid == 0) {
16         printf("child, PID=%d (parent=%d)\n", getpid(), getppid());
17     }
18     else {
19         wait(NULL);
20         printf("parent, PID=%d\n", getpid());
21     }
22     return 0;
23 }
24
25 /* Output example:
26 child, PID = 24422 (parent=24421)
27 parent, PID = 24421
28 */

```

**Linia 10** apeleaza functia *fork* si stocheaza rezultatul in variabila *pid*. Noul proces creat isi incepe executia de la urmatoarea linie, adica va intra pe *if* si apoi pe ramura corespunzatoare (*pid == 0*). Parintele va intra pe ramura *else*. Copilul isi va executa codul in timp ce parintele va ramane blocat pe *wait*; astfel, mai intai copilul va afisa cele doua PID-uri, iar apoi parintele isi va afisa PID-ul.

Se poate observa faptul ca nu a fost inclusa functia *exit* in instructiunile copilului; acesta si-a executat sectiunea de cod (codul parintelui fiind protejat de ramura *else*), iar apoi a returnat valoarea din *main* (*return 0*). Putem rescrie exemplul in felul urmator, facand necesara includerea functiei *exit*:

```
1  /* Creating one child process (v2) */
2
3  pid_t pid = fork();
4
5  if (pid < 0) {
6      return errno;
7  }
8  else if (pid == 0) {
9      printf("child, PID=%d (parent=%d)\n", getpid(), getppid());
10     exit(0);
11 }
12
13 wait(NULL);
14 printf("parent, PID=%d\n", getpid());
15 return 0;
```

Procesul copil va trece pe ramura *else if* (*pid == 0*), isi va executa instructiunile si se va termina prin intermediul apelului *exit*, returnand valoarea 0. Nu mai este necesara includerea ramurii *else* pentru parinte. In cazul in care dorim sa creem mai mult de un proces copil, exista urmatoarea abordare:

```
1  /* Creating two child processes */
2
3  pid_t pid1 = fork();
4
5  if (pid1 < 0) {
6      return errno;
7  }
8  else if (pid1 == 0) {
9      printf("child 1, PID=%d (parent=%d)\n", getpid(), getppid());
10     exit(0);
11 }
12
13 pid_t pid2 = fork();
14
15 if (pid2 < 0) {
16     return errno;
17 }
18 else if (pid2 == 0) {
19     printf("child 2, PID=%d (parent=%d)\n", getpid(), getppid());
20     exit(0);
21 }
22
23 wait(NULL);
24 wait(NULL);
25 printf("parent, PID=%d\n", getpid());
26
27 /* output example:
28 child 2, PID=25268 (parent=25266)
29 child 1, PID=25267 (parent=25266)
30 parent, PID=25266
31 */
```

Nu este determinista ordinea in care se executa copiii; aceasta sarcina este intrebuintata *scheduler-ului*. Totusi, daca se doreste crearea unui numar mai mare de procese, ar fi mai adecvata utilizarea instructiunilor repetitive (*for loops*).

In exemplul de mai jos vor fi create 10 procese. De asemenea, fiecare copil se va executa pentru un anumit numar de secunde cu ajutorul functiei **sleep**; se va stagna terminarea copiilor pentru a putea observa arborescenta proceselor in terminal.

```

1  /* Creating 10 child processes */
2
3  int num_children = 10;
4  for (int i = 0; i < num_children; ++i) {
5      pid_t pid = fork();
6
7      if (pid < 0) {
8          return errno;
9      }
10     else if (pid == 0) {
11         printf("child %d, PID=%d\n", i, getpid());
12         sleep(i + 5);
13         exit(0);
14     }
15 }
16
17 printf("parent, PID=%d\n", getpid());
18 for (int i = 0; i < num_children; ++i) {
19     wait(NULL);
20 }
21 return 0;

```

Un exemplu de output pentru acest program nedeterminist ar arata in felul urmatoar:

```

harapalb@ubuntu:~/Desktop$ ./p
child 0, PID=26307
child 1, PID=26308
child 3, PID=26310
child 2, PID=26309
parent, PID=26306
child 4, PID=26311
child 5, PID=26312
child 7, PID=26314
child 9, PID=26316
child 6, PID=26313
child 8, PID=26315

```

Cum procesele copil stagneaza cateva secunde dupa *print*, exista destul timp pentru a putea deschide un al doilea terminal si a vizualiza arborescenta proceselor. Se ruleaza programul, se identifica PID-ul parintelui si este rulata (in celalalt terminal) comanda **ps tree -p <parent\_PID>**.

```

harapalb@ubuntu:~/Desktop$ ps tree -p 26393
p(26393)
├── p(26394)
├── p(26395)
├── p(26396)
├── p(26397)
├── p(26398)
├── p(26399)
├── p(26400)
├── p(26401)
├── p(26402)
└── p(26403)

harapalb@ubuntu:~/Desktop$ ps tree -p 26393
p(26393)
├── p(26400)
├── p(26401)
├── p(26402)
└── p(26403)

```

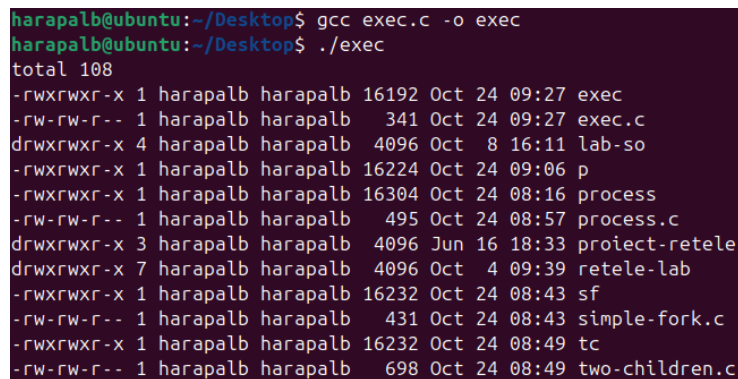
Rularea comenzii in primele 5 secunde afiseaza tot arborele cu cei 10 copii. Dupa cateva secunde, unele dintre aceste procese ies din *sleep* si isi finalizeaza executia; astfel, cand comanda este rulata din nou dupa o scurta pauza, arborele este mai mic.



### 3.3 Inlocuirea unui proces cu un alt program

Comanda **execve** are scopul de a inlocui un proces cu un alt program. Aceasta primeste 3 argumente: drumul catre programul respectiv, un array de string-uri ce contine argumentele programului (ultimul element fiind *NULL*, actionand ca si un simbol terminator), si un array de environment variables (in principal e *NULL* pentru a pastra acelasi environment). Mai jos este un exemplu ce presupune crearea unui proces copil si inlocuirea acestuia cu programul care executa comanda **ls**, afisand continutul directorului curent:

```
1  /* Create a new process to run the "ls" command */
2
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <stdlib.h>
6  #include <sys/wait.h>
7  #include <errno.h>
8
9  int main() {
10     char* const child_args[] = {"ls", "-l", NULL};
11     pid_t pid = fork();
12
13     if (pid < 0) {
14         return errno;
15     }
16     else if (pid == 0) {
17         execve("/usr/bin/ls", child_args, NULL);
18         exit(0);
19     }
20
21     wait(NULL);
22     return 0;
23 }
```



```
harapalb@ubuntu:~/Desktop$ gcc exec.c -o exec
harapalb@ubuntu:~/Desktop$ ./exec
total 108
-rwxrwxr-x 1 harapalb harapalb 16192 Oct 24 09:27 exec
-rw-rw-r-- 1 harapalb harapalb 341 Oct 24 09:27 exec.c
drwxrwxr-x 4 harapalb harapalb 4096 Oct 8 16:11 lab-so
-rwxrwxr-x 1 harapalb harapalb 16224 Oct 24 09:06 p
-rwxrwxr-x 1 harapalb harapalb 16304 Oct 24 08:16 process
-rw-rw-r-- 1 harapalb harapalb 495 Oct 24 08:57 process.c
drwxrwxr-x 3 harapalb harapalb 4096 Jun 16 18:33 proiect-retele
drwxrwxr-x 7 harapalb harapalb 4096 Oct 4 09:39 retele-lab
-rwxrwxr-x 1 harapalb harapalb 16232 Oct 24 08:43 sf
-rw-rw-r-- 1 harapalb harapalb 431 Oct 24 08:43 simple-fork.c
-rwxrwxr-x 1 harapalb harapalb 16232 Oct 24 08:49 tc
-rw-rw-r-- 1 harapalb harapalb 698 Oct 24 08:49 two-children.c
```

## 4 Comunicarea intre procese (IPC)

Un proces poate fi ori **independent** (nu afecteaza si nu se bazeaza pe rezultatele altor procese) ori **cooperant** (afecteaza sau este afectat de catre lucrul altor procese). Exista cateva beneficii ale proceselor cooperante:

- **Partajarea datelor:** este necesar ca anumite date sa fie partajate intre mai multe aplicatii.
- **Modularitate:** o sarcina complexa poate fi impartita in sarcini mai mici.
- **Eficientizarea calculelor:** fiecare sarcina mai mica poate rula in paralel.

Procesele cooperante au nevoie de un mod de a partaja date, adica un **mecanism de comunicare intre procese** (**interprocess communication mechanism**). Doua solutii fundamentale la aceasta problema sunt **memoria partajata** (**shared memory**) si **transmiterea mesajelor** (**message passing**).

## 4.1 Memorie partajata

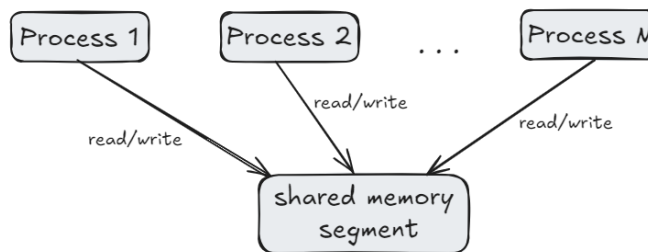


Figure 6: N procese accesand un segment de memorie partajata

O tehnica pentru partajarea datelor intre mai multe procese o constituie lucrul cu **memoria partajata**. Unul din procesele cooperante stabileste o regiune de memorie (un **segment de memorie partajata**) prin intermediul unui syscall; acea regiune poate fi atasata de spatiile de adresa ale proceselor cooperante, permitandu-le sa citeasca si sa scrie date. Acest tip de comunicare este gestionat de catre procesele cooperante, nu de catre OS.

Memoria partajata poate reprezenta o solutie pentru **problema producator-consumator (producer-consumer problem)**: un **proces producator (producer process)** creeaza obiecte care sunt consumate de catre un **proces consumator (consumer process)**. Ambele procese citesc si scriu date intr-o regiune de memorie partajata. Obiectele produse sunt stocate intr-un **buffer** (o zona de memorie pentru stocarea temporara a datelor, atunci cand sunt mutate dintr-un loc in altul) care poate fi de doua tipuri:

- **Unbounded buffer**: nu exista o limita asupra numarului de obiecte ce pot fi produse. Totusi, consumatorul trebuie sa astepte daca nu exista obiecte pe buffer.
- **Bounded buffer**: exista un numar maxim de obiecte ce pot fi stocate pe buffer. Astfel, producatorul trebuie sa astepte cand buffer-ul este plin, iar consumatorul trebuie sa astepte cand buffer-ul este gol.

Pentru a implementa o solutie folosind memoria partajata, consideram urmatoarele variabile (date) partajate, ce vor fi accesate de catre procesele cooperante:

```
1 #define BUFFER_SIZE 10
2
3 typedef struct {
4     ...
5 } item;
6
7 item buffer[BUFFER_SIZE];
8 int in = 0;
9 int out = 0;
```

Buffer-ul este implementat ca un *array circular*. Variabila **in** pointeaza catre *urmatoarea pozitie libera* din buffer, in timp ce variabila **out** pointeaza catre *prima pozitie ocupata*. In cazul in care buffer-ul este **gol** (e.g. la inceput cand  $in = 0$  si  $out = 0$ ), este indeplinita conditia **in == out**; in cazul in care este **plin**, avem  $((in + 1) \% BUFFER\_SIZE) == out$ .

### Proces producator

```
1 item next_produced;
2
3 while (true) {
4     /* produce an item, store it in
5      next_produced */
6
7     while (((in + 1) \% BUFFER_SIZE) == out);
8     /* do nothing */
9
10    buffer[in] = next_produced;
11    in = (in + 1) \% BUFFER_SIZE;
12 }
```

### Proces consumator

```
1 item next_consumed;
2
3 while (true) {
4     while (in == out); /* do nothing */
5
6     next_consumed = buffer[out];
7     out = (out + 1) \% BUFFER_SIZE;
8
9     /* consume the item stored in
10    next_consumed */
11 }
```

O mica problema: solutia de mai sus permite stocarea a maxim  $BUFFER\_SIZE - 1$  obiecte; daca atingem aceasta limita, buffer-ul este considerat ca fiind "plin" si avem  $((in + 1) \% BUFFER\_SIZE) == out$ . Daca mai adaugam un obiect, obtinem  $in == out$ , indicand faptul ca buffer-ul este *gol* (cand el are, de fapt,  $BUFFER\_SIZE$  obiecte). Pentru a corecta aceasta problema, putem folosi o variabila **count** = 0, ce indica numarul de obiecte aflate la momentul current in buffer.

#### Producer process

```

1  item next_produced;
2
3  while (true) {
4      /* produce an item, store it in
       next_produced */
5
6      while (count == BUFFER_SIZE); /* do
       nothing */
7
8      buffer[in] = next_produced;
9      in = (in + 1) % BUFFER_SIZE;
10     count++;
11 }

```

#### Consumer process

```

1  item next_consumed;
2
3  while (true) {
4      while (count == 0); /* do nothing */
5
6      next_consumed = buffer[out];
7      out = (out + 1) % BUFFER_SIZE;
8      count--;
9
10     /* consume the item stored in
       next_consumed */
11 }

```

In teorie, solutia este corecta, dar mai apare o problema: doua sau mai multe procese ce acceseaza in acelasi timp niste date partajate ar putea corupe datele respective. De exemplu, consideram instructiunea **count++**; implementarea acesteia in cod masina s-ar face astfel (similar pentru **count--**):

```

1  register1 = count
2  register1 = register1 + 1
3  count = register1

```

Executia simultana a instructiunilor **count++** si **count--** produce o ordonare aleatoare a instructiunilor la nivel de cod masina. Mai jos poate fi observat un exemplu de o ordonare aleatoare (consideram **count = 5**):

1	register1 = count	[register1 = 5]
2	register1 = register1 + 1	[register1 = 6]
3	register2 = count	[register2 = 5]
4	count = register1	[count = 6]
5	register2 = register2 - 1	[register2 = 4]
6	count = register2	[count = 4]

In final, valoarea variabilei *count* este 4 in loc de 5. Aceasta problema poarta denumirea de **conditie de cursa** (**race condition**), si va fi studiata in detaliu la alt tutorial.

## 4.2 Message passing

Aceasta metoda de comunicare presupune **transmiterea de mesaje** (unde un **mesaj** este un *pachet de date*). Cele doua operatii fundamentale sunt **send(message)** si **receive(message)**, iar mesajele pot fi de dimensiune fixa sau variabila. Daca doua procese doresc sa isi transmita mesaje, vor avea nevoie de o **legatura de comunicare** (**communication link**), ce poate avea mai multe implementari.

### 4.2.1 Naming

Acest concept se refera la felul in care se identifica procesele cooperante intre ele. Comunicarea poate avea loc in mod **direct** sau **indirect**. In **comunicarea directa**, procesele sunt nevoite sa se identifice in mod explicit pentru a putea transmite mesaje; astfel, operatiile devin:

- **send(message, P)**: trimite un mesaj catre procesul P.
- **receive(message, Q)**: primeste un mesaj de la procesul Q.

Proprietatile legaturilor de comunicare sunt urmatoarele:

- Intre fiecare pereche de procese exista o *singura* legatura de comunicare. Astfel, procesele doar trebuie sa isi cunoasca identitatile pentru a putea comunica.

- O legatura este asociata cu exact o singura pereche de procese cooperante.

Aceasta abordare este considerata **simetrica**, deoarece expeditorul si destinatarul trebuie sa se numeasca explicit pentru a putea comunica. Se poate considera si o abordare **asimetrica**; operatiile devin:

- **send(message, P)**: trimite un mesaj catre procesul P.
- **receive(id, message)**: primeste un mesaj de la un proces oarecare, al carui identificator este retinut in variabila **id**.

In cazul **comunicarii indirecte**, mesajele sunt trimise si primite prin intermediul unor **cutii postale (mailbox-uri)**; niste zone de memorie gestionate de catre kernel, folosite pentru IPC). Operatiile se schimba in felul urmator:

- **send(A, message)**: trimite un mesaj catre mailbox-ul A.
- **receive(A, message)**: primeste un mesaj din mailbox-ul A.

In acest caz, proprietatile legaturilor de comunicare se modifica astfel:

- O legatura este stabilita intre doua procese numai daca procesele respective au un mailbox in comun.
- O legatura poate fi asociata cu mai multe procese.
- Fiecare pereche de procese poate avea mai multe legaturi.
- Legatura poate fi *unidirectionala* sau *bidirectionala*.

Apare o problema: presupunem ca procesele  $P_1$ ,  $P_2$  si  $P_3$  au in comun mailbox-ul A. Procesul  $P_1$  trimite un mesaj catre A, iar  $P_2$  si  $P_3$  executa *receive* in acelasi timp. Care dintre cele doua procese primeste mesajul respectiv? Raspunsul depinde de strategia abordata:

- Permite ca o legatura sa fie asociata cu maxim doua procese.
- Permite numai unui singur proces sa poata executa operatia *receive* la un moment dat.
- Lasam sistemul sa aleaga in mod aleator (sau folosind un anumit algoritim) procesul ce primeste mesajul.

#### 4.2.2 Synchronization

Acest concept se refera la felul in care expeditorul si destinatarul isi coordoneaza comunicarea; aceasta poate avea loc in mod **sincron (synchronous / blocking)** sau **asincron (asynchronous / non-blocking)**. Intr-un **mediu sincron**, operatiile se desfasoara astfel:

- **Blocking send**: expeditorul trimite un mesaj si se blocheaza pana cand destinatarul sau mailbox-ul primeste mesajul.
- **Blocking receive**: destinatarul este blocat pana cand apare un mesaj.

Intr-un **mediu asincron**:

- **Non-blocking send**: expeditorul trimite mesajul si isi continua executia.
- **Non-blocking receive**: destinatarul primeste un mesaj valid sau un obiect *NULL*.

#### 4.2.3 Buffering

Acesta este mecanismul pentru stocarea temporara a mesajelor; acestea sunt plasate intr-o coada, care poate fi implementata in trei moduri:

- **Zero capacity**: mesajele nu sunt stocate. Expeditorul si destinatarul trebuie sa comunice *sincron*.
- **Bounded capacity**: exista un numar maxim de mesaje ce se pot afla in coada. Odata ce limita este atinsa, expeditorul se blocheaza pana cand se mai elibereaza coada (*blocking send*).
- **Unbounded capacity**: nu este impusa o limita asupra numarului maxim de mesaje ce pot fi stocate (*non-blocking send*).

#### 4.2.4 Pipes

Un **pipe** (conducta) reprezinta o implementare pentru tehnica de **message passing**. Un pipe este creat de catre un proces cu ajutorul apelurilor de sistem. Mecanismul de implementare este simplu, dar trebuie luate in considerare urmatoarele lucruri:

- Pipe-ul permite cumva comunicare **unidirectionala** (conecteaza output-ul unui proces de input-ul altui proces, facilitand comunicarea dintre un **writer process** si un **reader process**) sau **bidirectionala**?
- In cazul in care avem comunicare bidirectionala, este cumva **half-duplex** (datele pot merge intr-o singura directie la un moment dat) sau **full-duplex** (datele pot merge in ambele directii in acelasi timp)?
- Ar trebui sa existe o relatie (e.g. *parinte-copil*) intre procesele cooperante?
- Pot sa comunice pipe-urile peste o retea, sau trebuie ca procesele sa se afle pe acelasi calculator?

##### 4.2.4.1 Ordinary pipes

Un **ordinary pipe** permite **comunicare unidirectionala**: *procesul producator* scrie la un capat al pipe-ului (**write end**), iar *procesul consumator* citeste de la celalalt capat (**read end**). Daca se doreste **comunicare bidirectionala**, se vor folosi doua pipe-uri, fiecare transmitand date in directii opuse.

Odata ce un proces creeaza un ordinary pipe, alte procese din exterior nu pot accesa pipe-ul respectiv. Totusi, cum un pipe este un fisier special, acesta poate fi mostenit de catre procesele copil. Astfel, este necesara o relatie **parinte-copil** intre procesele cooperante; asta inseamna si ca procesele trebuie sa se afle pe acelasi dispozitiv pentru a putea comunica. Odata ce se incheie comunicarea, se finalizeaza si existenta pipe-ului respectiv.

Crearea unui ordinary pipe se realizeaza cu ajutorul syscall-ului **pipe** (`int fd[]`), unde `fd` este un array ce contine doi **descriptori de fisier** (**file descriptors**). Primul este folosit de catre *procesul copil* pentru a *citi* date, iar celalalt este folosit de catre *procesul parinte* pentru a *scrie* date.

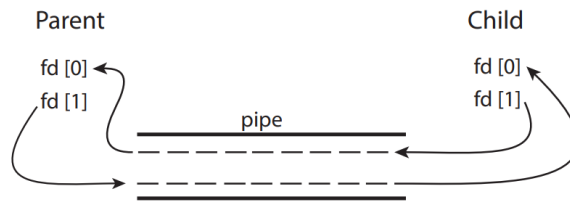


Figure 7: Un proces copil si un proces parinte comunicand folosind un ordinary pipe

##### 4.2.4.2 Named pipes

Un **named pipe** (cunoscut si sub numele de **FIFO** pe sistemele UNIX) permite **comunicare bidirectionala** si nu necesita o relatie intre procesele cooperante. Pot exista mai multe procese care scriu (sau citesc) date in acelasi timp. Un asemenea pipe nu depinde de existenta proceselor; acestea pot incheia comunicarea sau se pot termina, iar pipe-ul o sa functioneze in continuare.

##### 4.2.4.3 Concluzie la pipes

Un **ordinary pipe** indeplineste urmatoarele patru proprietati:

- Comunicarea este **unidirectionala**.
- Implicit, este si **half-duplex**.
- Este necesara o relatie **parinte-copil** intre procesele cooperante.
- Procesele pot comunica doar pe acelasi calculator.

Un **named pipe** indeplineste urmatoarele proprietati:

- Comunicarea este **bidirectionala**.
- Tipul de duplex depinde de sistem (**half-duplex** pentru UNIX, **full-duplex** pentru Windows).
- Nu este necesara o relatie intre procesele cooperante.
- Pe sistemele UNIX, comunicarea poate avea loc doar pe acelasi calculator; pe Windows, comunicarea poate avea loc pe o retea de dispozitive.

#### 4.2.5 Sockets

O **adresa IP** este un identificator atribuit unui calculator conectat la o retea. Un **port** este un identificator pentru un serviciu specific sau o aplicatie de pe un calculator. Impreuna, adresa si port-ul formeaza un **socket**. O legatura de comunicare bidirectionala poate fi obtinuta folosind doua socket-uri (ce reprezinta capetele legaturii).

Consideram o **arhitectura client-server**. Un server care implementeaza un anumit serviciu precum **SSH**, **FTP** sau **HTTP** va deschide un socket folosind port-ul corespunzator serviciului respectiv (e.g. port 80 pentru HTTP). Punand impreuna IP-ul server-ului si port-ul serviciului, se obtine un socket din partea server-ului (e.g. pentru IP-ul 161.25.19.8 si port-ul 80 avem 161.25.19.8:80). Clientii pot trimite cereri server-ului pe port-ul respectiv, iar fiecare client va avea propriul sau socket (port-ul clientului este atribuit de catre sistemul de operare si trebuie sa fie mai mare decat 1023; e.g. pentru IP-ul 146.86.5.20 si port-ul 1625 se obtine socket-ul 146.86.5.20:1625 corespunzator clientului).

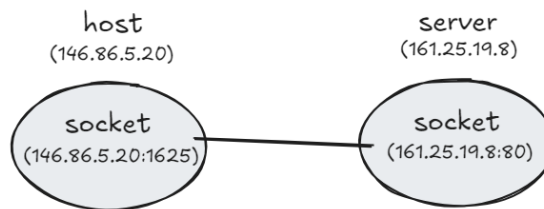


Figure 8: Comunicarea folosind socket-uri