Programare de sistem în C pentru platforma Linux (IV)

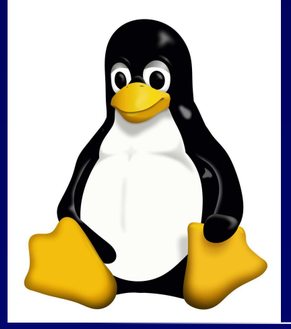
Gestiunea proceselor, partea I-a:

Crearea și sincronizarea proceselor – primitivele `fork()` și `wait()`

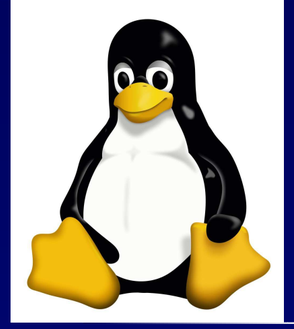
Cristian Vidrașcu

cristian.vidrascu@info.uaic.ro

Aprilie, 2024



Sumar



Introducere

Definiție

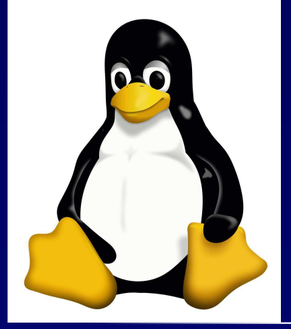
"Un *program* este un *fișier executabil*" (evident, obținut prin compilare dintr-un fișier sursă), aflat pe un suport de stocare (*i.e.*, păstrat pe o memorie secundară – *e.g. harddisk*, SDD, stick USB, etc.).

Definiție

"Un *proces* este un program aflat în curs de execuție."

Mai precis, un *proces* este o *instanță de execuție* a unui program, fiind caracterizat de :

- o "durată de viață" (*i.e.*, perioada de timp în care se execută acel program)
- memorie alocată, pentru a stoca codul + datele + stivele procesului
- timp procesor alocat de către planificatorul SO-ului, pentru execuția programului
- alte resurse necesare pentru execuție



Agenda

Introducere

Noțiuni generale despre procese

- Tabela tuturor proceselor active în sistem
- Primitive ce oferă anumite informații despre procese
- Primitive pentru suspendarea execuției programului
- Primitive pentru terminarea execuției programului

Crearea proceselor

- Primitiva `fork`
- Caracteristicile noului proces după `fork`

Așteptarea terminării unui fiu

- Sincronizarea proceselor
- Primitiva `wait`

Demo: programe cu `fork` și `wait`

- Alte programe demonstrative

Referințe bibliografice

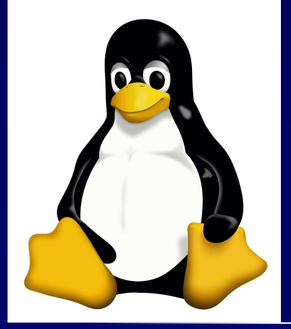
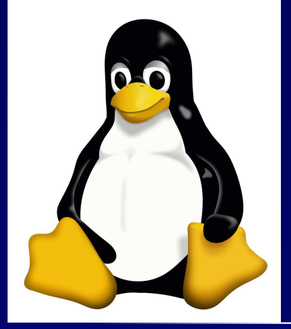


Tabela tuturor proceselor active în sistem

Nucleul sistemului de operare păstrează evidența proceselor din sistem cu ajutorul unei tabele a proceselor active.

Această tabelă conține câte o intrare pentru fiecare proces existent în sistem, intrare numită *PCB* (*Process Control Block*) și care conține o serie de informații despre procesul asociat acesteia :

- PID-ul = identificatorul de proces – este un întreg pozitiv, de tipul `pid_t` (definit în `sys/types.h`)
- `PPID`-ul : PID-ul procesului *părinte*
- UID-ul proprietarului *real* al procesului (proprietarul real este utilizatorul care l-a lansat în execuție)
- GID-ul grupului proprietar *real* al procesului
- `EUID`-ul și `EGID`-ul = UID-ul și GID-ul proprietarului *efectiv* (adică acel utilizator ce determină drepturile procesului de acces la resurse)
  Notă: dacă bitul *setuid* este 1, atunci, pe toată durata de execuție a acelui fișier, proprietarul efectiv al procesului va fi proprietarul fișierului, și nu utilizatorul care îl execută; similar pentru bitul *setgid*.
- terminalul de control
- linia de comandă (*i.e.*, parametrii cu care a fost lansat în execuție)
- variabilele de mediu transmise de către părinte
- *starea curentă* a procesului – poate fi *ready*, *running*, *waiting* sau *finished* (aka *zombie*)
- contextul hardware, ș.a.



Primitive ce oferă anumite informații despre procese

- Primitive pentru aflarea PID-urilor procesului curent și a părintelui acestuia : `getpid` și `getppid`.
  Interfețele acestor funcții :

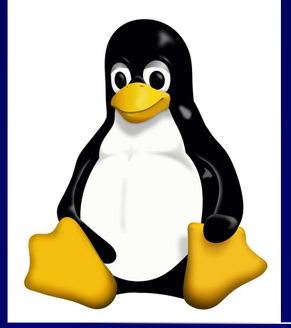
```
pid_t getpid(void);
pid_t getppid(void);
```

- Primitive pentru aflarea UID-urilor proprietarului real / efectiv al procesului curent și a GID-urilor grupului proprietar real / efectiv al acestuia : `getuid` / `geteuid` și, respectiv, `getgid` / `getegid`.
  Interfețele acestor funcții :

```
uid_t getuid(void);
gid_t getgid(void);
uid_t geteuid(void);
gid_t getegid(void);
```

- Primitivele `getrlimit` și `setrlimit` – permit aflarea și, respectiv, setarea unor limite (“soft” și “hard”) pentru diverse tipuri de resurse. Pentru mai multe detalii, consultați documentația `man 2 getrlimit` .
- Funcția `sysconf` – permite aflarea unor constante de configurare a sistemului. Pentru mai multe detalii, consultați documentația `man 3 sysconf` .

  Demo: a se vedea programul demonstrativ `get_limits_ex.c`, descris detaliat în exemplul [FourthDemo – get_limits_ex] prezentat în suportul de laborator #10.



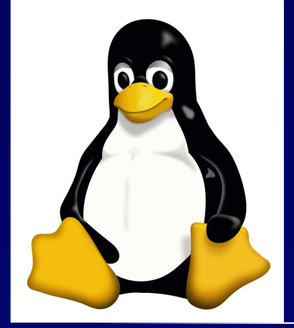
Primitive pentru suspendarea execuției programului

Primitive de suspendare a execuției programului, pe o durată de timp specificată ([5]) :

- Primitiva `sleep`, cu o precizie de ordinul secundelor. Interfața acestei funcții :
  `unsigned int sleep(unsigned int` *nr_secunde*`);`
  Demo: a se vedea programul demonstrativ `info_ex.c`, ilustrat în exemplul [FirstDemo – info_ex] prezentat în suportul de laborator #10.
- Primitiva `usleep`, cu o precizie de ordinul microsecundelor. Interfața acestei funcții :
  `int usleep(useconds_t` *nr_microsecunde*`);`
- Primitiva `nanosleep`, cu o precizie de ordinul nanosecundelor și care este mai eficientă, a fost introdusă în versiuni mai recente ale standardului POSIX. Interfața acestei funcții :
  `int nanosleep(const struct timespec *`*req*`, struct timespec *`*rem*`);`

Atenție: durata efectivă de pauză în execuția programului poate depăși valoarea specificată în aceste apeluri (!). Măsurarea timpului scurs depinde (și) de precizia ceasului utilizat, i.e. de suportul hardware oferit; detalii despre măsurarea timpului în `Linux` puteți afla citind documentația man 7 time.

Observație: pot exista situații în care durata efectivă de pauză în execuția programului să fie mai mică decât valoarea specificată în aceste apeluri (!). Și anume, dacă procesului îi este livrat un semnal în timpul acestei pauze, apelul returnează “imediat” `-1`, iar în variabila `errno` este scrisă valoarea `EINTR`. Deci se anulează restul de timp care mai era de petrecut în starea de suspendare a execuției programului. Doar primitiva `nanosleep` permite aflarea duratei de așteptare “pierdute” în acest fel, și astfel ea poate fi “recuperată“ printr-un nou apel `nanosleep` cu restul de timp respectiv.



Primitive pentru terminarea execuției programului

Un proces (*i.e.*, execuția unui program) se poate termina în două moduri :

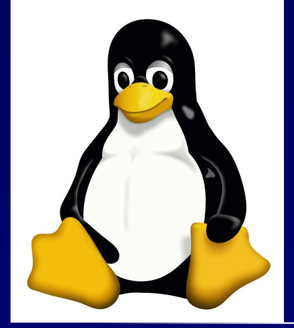
– *terminarea normală* a execuției unui proces : se petrece în momentul întâlnirii în program a apelului primitivei `exit` (sau ca urmare a instrucțiunii `return` întâlnită în funcția *main*).

  Interfața acestei funcții :

  ```
  noreturn void exit(int status);
  ```

  Efect : procesul apelant își încheie execuția normal (*i.e.*, procesul este trecut în starea *finished*, se închid fișierele deschise și se salvează pe disc conținutul *buffer*-elor folosite, se dealocă zonele de memorie alocate procesului respectiv, ș.a.m.d.), iar valoarea *status* este “trunchiată” (*i.e.*, *status* `&` `0xFF`) și salvată drept *codul de terminare normală*, în intrarea *PCB* asociată procesului din tabela proceselor, pentru a putea fi consultată ulterior de către părintele procesului respectiv. *Notă*: până la acel moment ulterior, procesul este *zombie* (*i.e.*, se mai păstrează din el doar intrarea sa *PCB*).

– *terminarea anormală* a execuției unui proces : a se vedea slide-ul următor.



Primitive pentru terminarea execuției programului (cont.)

Un proces (*i.e.*, execuția unui program) se poate termina în două moduri :

– *terminarea normală* a execuției unui proces : a se vedea slide-ul precedent.

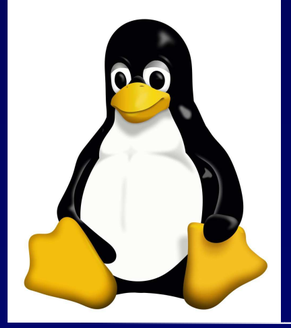
– *terminarea anormală* a execuției unui proces : se petrece în momentul primirii unui semnal de un anumit tip (*e.g.*, semnalul generat cu un apel `abort`).

Interfața acestei funcții :

```
noreturn void abort(void);
```

Efect : acest apel mai întâi deblochează semnalul `SIGABRT` și apoi livrează acest semnal procesului apelant, ceea ce cauzează terminarea anormală a procesului.

Observații:
i) vom vedea ulterior și alte tipuri de semnale ce cauzează terminarea anormală a unui proces ;
ii) și la terminarea anormală a unui proces se dealocă zonele de memorie ocupate de acel proces și se păstrează doar intrarea sa *PCB* din tabela proceselor, până când părintele său va cere codul de terminare (reprezentat în acest caz de numărul semnalului ce a cauzat terminarea anormală) .



Agenda

Introducere

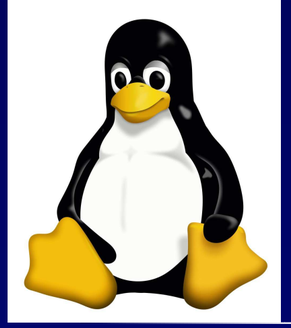
Noțiuni generale despre procese
- Tabela tuturor proceselor active în sistem
- Primitive ce oferă anumite informații despre procese
- Primitive pentru suspendarea execuției programului
- Primitive pentru terminarea execuției programului

Crearea proceselor
- Primitiva `fork`
- Caracteristicile noului proces după `fork`

Așteptarea terminării unui fiu
- Sincronizarea proceselor
- Primitiva `wait`

Demo: programe cu `fork` și `wait`
- Alte programe demonstrative

Referințe bibliografice



Primitiva `fork`

Singura modalitate de creare a proceselor în sistemele UNIX/Linux este cu ajutorul apelului de sistem `fork` ([6]). Interfața acestei funcții este următoarea :

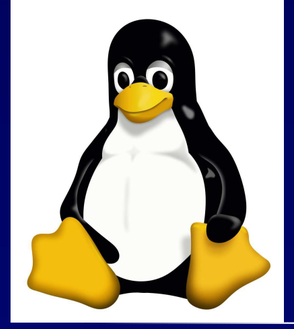
```
pid_t fork(void);
```

Efect : prin acest apel *se creează o copie a procesului apelant*, și ambele procese – cel nou creat și cel apelant – își vor continua execuția cu următoarea instrucțiune (din programul executabil) ce urmează după apelul funcției `fork`.

Procesul apelant va fi numit *părintele* procesului nou creat, iar acesta va fi numit *fiul* procesului apelant (mai exact, va fi unul dintre procesele fii ai acestuia).

Notă: practic, singura diferență la execuție dintre cele două procese va fi valoarea returnată de apelul funcției `fork`, precum și, bineînțeles, PID-urile proceselor.

Astfel, pe baza valorii returnate ce diferă în cele două procese, se poate ramifica execuția astfel încât fiul să execute altceva decât tatăl. Spre exemplificare, a se vedea programele demonstrative `fork_ex1.c` și `fork_ex2.c`, prezentate în exemplul [SecondDemo] din suportul de laborator #10.



Primitiva `fork` (cont.)

Valoarea returnată :

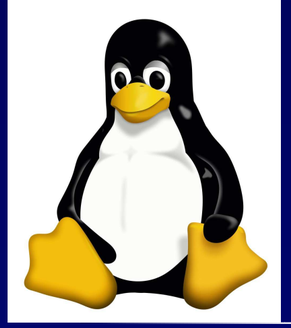
Apelul `fork` returnează valoarea `-1`, în caz de eroare (*i.e.*, dacă nu s-a putut crea un nou proces, din diverse cauze posibile), iar în caz de succes, returnează respectiv următoarele valori în cele două procese, tată și fiu :

– *n* , în procesul tată, unde *n* este PID-ul noului proces creat ;

– `0` , în procesul fiu .

Observații:

- PID-ul unui nou proces nu poate fi niciodată `0`, deoarece procesul cu PID-ul `0` nu este fiul nici unui proces, ci este rădăcina arborelui proceselor, și este singurul proces din sistem ce nu se creează prin apelul `fork`, ci este creat atunci când se *boot*-ează sistemul `UNIX/Linux` pe calculatorul respectiv. Detalii despre acest proces și cel cu PID-ul `1` puteți afla în articolul de aici .
- Procesul nou creat poate afla PID-ul tatălui cu ajutorul primitivei `getppid`, pe când procesul părinte nu poate afla PID-ul noului proces creat, fiu al său, prin altă manieră decât prin valoarea returnată de apelul `fork`.
  (*Notă*: nu s-a creat o primitivă pentru aflarea PID-ului fiului deoarece, spre deosebire de părinte, fiul unui proces nu este unic – un proces poate avea zero, unul, sau mai mulți fii la un moment dat.)



Caracteristicile noului proces după `fork`

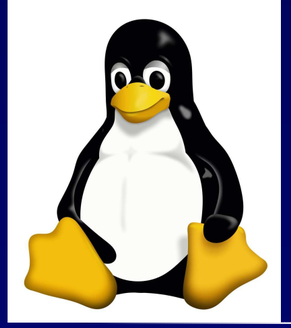
Prin “clonarea” unui proces, fiul nou creat “moștenește” o serie de caracteristici ale părintelui, și anume :

- are același proprietar, real și efectiv, și același grup proprietar, real și efectiv ;
- are aceeași linie de comandă și aceleași variabile de mediu transmise de către părinte ;
- are același set de descriptori de fișiere deschise și, mai mult, fiecare descriptor referă, în ambele procese, aceeași intrare în tabela globală de fișiere deschise (*i.e.*, fiecare sesiune de lucru cu un fișier este “partajată” de procesele tată și fiu) ; de asemenea, descriptorii “partajați” se pot referi atât la fișiere de orice tip, cât și la alte tipuri de resurse accesibile prin astfel de descriptori (*e.g.*, obiecte de tip *shared memory*, create cu apelul `shm_open`) ;
- are același set de mapări, atât persistente (*i.e.*, mapări în memorie ale fișierelor de pe disc), cât și ne-persistente (anonime și cu nume) .

Concluzie importantă: așadar, datorită operației de “clonare”, *imediat după apelul `fork`* procesul fiu va avea aceleași valori ale variabilelor din program și aceleași referințe către resurse (*i.e.*, fișiere deschise, mapări de fișiere, etc.) ca și procesul părinte. Mai departe însă, *fiecare proces va lucra pe propria sa zonă de memorie* și va afecta în felul său resursele partajate. Prin urmare, dacă fiul modifică valoarea unei variabile, această modificare nu va fi vizibilă și în procesul tată (și nici invers).

În concluzie, în mod implicit nu avem memorie partajată (*shared memory*) între procesele părinte și fiu.

Observație: în `Linux`, apelul de sistem `fork` este implementat folosind pagini COW (*copy-on-write*), ceea ce optimizează timpul de creare a fiului, util mai ales când fiul apelează imediat o funcție `exec`.



Agenda

Introducere

Noțiuni generale despre procese

- Tabela tuturor proceselor active în sistem
- Primitive ce oferă anumite informații despre procese
- Primitive pentru suspendarea execuției programului
- Primitive pentru terminarea execuției programului

Crearea proceselor

- Primitiva `fork`
- Caracteristicile noului proces după `fork`

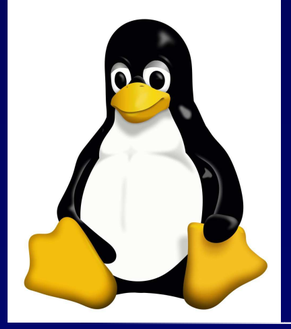
Așteptarea terminării unui fiu

- Sincronizarea proceselor
- Primitiva `wait`

Demo: programe cu `fork` și `wait`

- Alte programe demonstrative

Referințe bibliografice

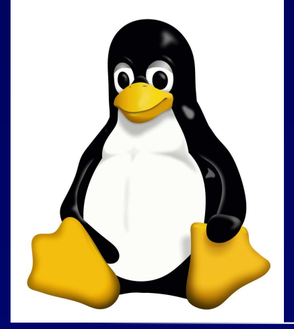


Sincronizarea proceselor

În programarea concurentă există noțiunea de *punct de sincronizare* a două procese : este un punct din care cele două procese au o execuție simultană (*i.e.*, este un punct de așteptare reciprocă). Punctul de sincronizare nu este o noțiune dinamică, ci una statică (o noțiune fixă) : este precizat în algoritm (*i.e.*, în program) locul unde se găsește acest punct de sincronizare.

i) Primitiva `fork` este un exemplu de punct de sincronizare :
Cele două procese – procesul apelant al primitivei `fork` și procesul nou creat de apelul acestei primitive – își continuă execuția simultan din acest punct (*i.e.*, punctul din program în care apare apelul funcției `fork`).

ii) Un alt exemplu de sincronizare, des întâlnită în practică :
Procesul părinte poate avea nevoie de valoarea de terminare returnată de un proces fiu.
Pentru a realiza această facilitate, trebuie stabilit un punct de sincronizare între sfârșitul programului fiu și punctul din programul părinte în care este nevoie de acea valoare, și apoi trebuie transferată acea valoare de la procesul fiu la procesul părinte.



Primitiva `wait`

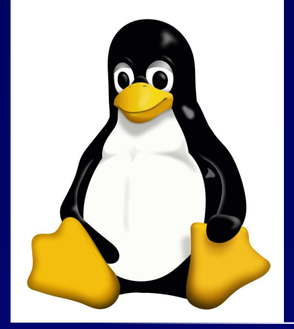
Apelul de sistem `wait` este utilizat pentru a aștepta un proces fiu să-și termine execuția ([6]). Interfața acestei funcții este următoarea :

```
pid_t wait(int* wstatus);
```

Efect : apelul funcției `wait` suspendă execuția procesului apelant până în momentul în care unul dintre fiii acelui proces (oricare dintre ei), se termină sau este stopat (*i.e.*, terminat anormal printr-un semnal). Iar dacă există vreun fiu care deja s-a terminat sau a fost stopat, atunci funcția `wait` returnează imediat.

Notă: mai există și primitiva `waitpid` ([6]), care va aștepta terminarea fie a unui anumit fiu (specificat prin PID-ul său dat ca argument), fie a oricărui fiu (dacă se specifică `-1` drept PID), și are un argument suplimentar care influențează modul de așteptare (*e.g.*, opțiunea `WNOHANG` este utilă pentru a testa fără așteptare existența vreunui fiu deja terminat).

Observații: i) dacă un proces se termină înaintea părintelui său, atunci el devine *zombie* (*i.e.*, i se păstrează în sistem doar intrarea sa *PCB* din tabela proceselor) ; ii) iar dacă procesul părinte se termină înaintea vreunui proces fiu, atunci acelui fiu i se va atribui ca părinte procesul `init` (ce are PID-ul `1`), iar acest lucru se face pentru toate procesele fii neterminate încă în momentul terminării părintelui lor .



Primitiva `wait` (cont.)

Valoarea returnată :

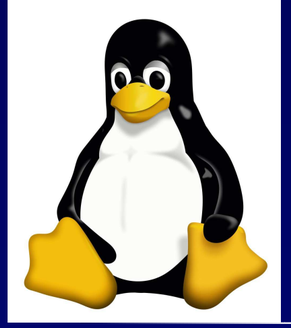
Apelul `wait` returnează ca valoare PID-ul acelui proces fiu, iar în locația referită de pointerul *wstatus* este salvată următoarea valoare :

- codul de terminare a acelui proces fiu (și anume, în octetul *high* al acelui `int`), dacă funcția `wait` returnează deoarece un fiu s-a terminat normal ;
- tipul semnalului (și anume, în octetul *low* al acelui `int`), dacă funcția `wait` returnează deoarece un fiu a fost stopat de un semnal .

Pentru a inspecta valoarea stocată în $*$*wstatus* pot fi folosite *macro*-urile următoare : `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFSTOPPED`, `WSTOPSIG`, ș.a.

Observație: dacă procesul apelant nu are procese fii, atunci funcția `wait` returnează valoarea `-1`, iar variabila `errno` este setată în mod corespunzător pentru a indica eroarea (*i.e.*, `ECHILD`).

Demo: a se vedea programele demonstrative `wait_ex1.c`, `wait_ex2.c` și `wait_ex3.c`, prezentate în exemplul [ThirdDemo] din suportul de laborator #10.



Agenda

Introducere

Noțiuni generale despre procese

- Tabela tuturor proceselor active în sistem
- Primitive ce oferă anumite informații despre procese
- Primitive pentru suspendarea execuției programului
- Primitive pentru terminarea execuției programului

Crearea proceselor

- Primitiva `fork`
- Caracteristicile noului proces după `fork`

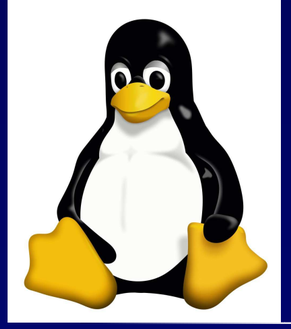
Așteptarea terminării unui fiu

- Sincronizarea proceselor
- Primitiva `wait`

Demo: programe cu `fork` și `wait`

- Alte programe demonstrative

Referințe bibliografice



Alte programe demonstrative

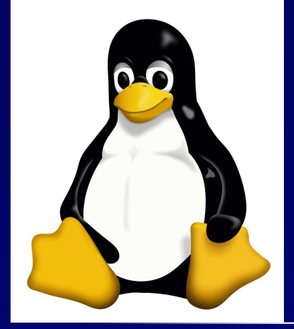
Demo: exercițiile rezolvate [N children] și [A list of processes #1 & #2], prezentate în suportul online de laborator ([3]), ilustrează două exemple de programe care creează câte o anumită ierarhie de procese (un părinte singur cu o serie de copii și, respectiv, un arbore 1-ar de procese) și care, totodată, prelucrează statusul execuției proceselor nou create.

* * *

Demo: exercițiul rezolvat ['Supervisor-workers' pattern #1: A coordinated distributed sum #1], prezentat în suportul online de laborator ([3]) , ilustrează o aplicare a șablonului de cooperare 'Supervisor / workers' pentru realizarea unui calcul paralel / distribuit.

* * *

Demo: exercițiile rezolvate [Demo 'data race'_shmem #1 & #2 : . . .], prezentate în suportul online de laborator ([3]) , ilustrează cum putem "agrega" cele două programe producător și consumator prezentate în două exerciții cu nume similare din suportul de laborator precedent, într-un singur program ce va îngloba ambele funcționalități.
În plus, se ilustrează folosirea unei mapări anonime partajate, în locul unui fișier mapat în memorie, pentru implementarea *buffer*-ului partajat (utilizat pentru comunicația dintre cele două procese producător și consumator), precum și implementarea unei soluții de sincronizare fără semafoare pentru aceste două instanțe particulare ale problemei producător-consumator.



Bibliografie obligatorie

[1] Cap. 4, §4.1–§4.3 din cartea “*Sisteme de operare – manual pentru ID*”, autor C. Vidrașcu, editura UAIC, 2006. *Notă*: este accesibilă, în format PDF, din pagina disciplinei “Sisteme de operare”:

- https://edu.info.uaic.ro/sisteme-de-operare/SO/books/ManualID-SO.pdf

[2] Programele demonstrative amintite pe parcursul acestei prezentări pot fi descărcate de la:

- https://edu.info.uaic.ro/sisteme-de-operare/SO/lectures/Linux/demo/fork/
- https://edu.info.uaic.ro/sisteme-de-operare/SO/lectures/Linux/demo/wait/

[3] Suportul de laborator online asociat acestei prezentări:

- https://edu.info.uaic.ro/sisteme-de-operare/SO/support-lessons/C/suport_lab10.html

Bibliografie suplimentară:

[4] Cap. 24, 25 și 26 din cartea “The Linux Programming Interface : A Linux and UNIX System Programming Handbook”, autor M. Kerrisk, editura No Starch Press, 2010.

- https://edu.info.uaic.ro/sisteme-de-operare/SO/books/TLPI1.pdf

[5] POSIX API for sleeping : `man 3 sleep`, `man 3 usleep`, `man 2 nanosleep`, `man 7 time`.

[6] POSIX API for cloning & waiting : `man 2 fork`, `man 2 wait`, `man 2 waitpid`.