Programare concurentă în C (III) :

Gestiunea proceselor, partea I-a: Crearea și sincronizarea proceselor – primitivele `fork()` și `wait()`

Cristian Vidrașcu

vidrascu@info.uaic.ro

Sumar

- Noțiuni generale despre procese
- Primitive utile referitoare la procese
- Crearea proceselor – primitiva `fork()`
- Terminarea proceselor
- Sincronizarea proceselor – primitiva `wait()`
- Șabloane de cooperare și sincronizare

Noțiuni generale despre procese

Program = un *fișier executabil* (evident, obținut prin compilare dintr-un fișier sursă), aflat pe un suport de memorare extern (*e.g. harddisk*).

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 timp (*i.e.* perioada de timp în care se execută acel program), o zonă de memorie alocată (zona de cod + zona de date + stiva), timp procesor alocat, ș.a.

Noțiuni generale despre procese (cont.)

Nucleul sistemului de operare păstrează evidența proceselor din sistem prin intermediul unei tabele a proceselor active. Aceasta conține câte o intrare pentru fiecare proces existent în sistem, intrare ce conține o serie de informații despre acel proces:

- `PID`-ul = identificatorul de proces – este un întreg pozitiv, de tipul `pid_t` (tip definit în fișierul *header* `sys/types.h`)
- `PPID`-ul : `PID`-ul procesului *părinte*
- terminalul de control
- `UID`-ul utilizatorului proprietar *real* al procesului (proprietarul real este utilizatorul care l-a lansat în execuție)
- `GID`-ul grupului proprietar *real* al procesului

Noțiuni generale despre procese (cont.)

- `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 fișierului respectiv, proprietarul efectiv al procesului va fi proprietarul fișierului, și nu utilizatorul care îl execută; similar pentru bitul *setgid*.

- *starea procesului* – poate fi una dintre următoarele:
  - *ready* = pregătit pentru execuție
  - *running* = în execuție
  - *waiting* = în așteptarea producerii unui eveniment (e.g., terminare op. I/O)
  - *finished* = terminare normală
- linia de comandă (parametrii cu care a fost lansat în execuție)
- variabilele de mediu transmise de către părinte
- ș.a.

Primitive utile referitoare la procese (1/3)

- Primitive pentru aflarea `PID`-urilor unui proces și a parintelui acestuia: `getpid`, `getppid`. Interfețele acestor funcții:

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

- Primitive pentru aflarea `ID`-urilor proprietarului unui proces și a grupului acestuia: `getuid`, `getgid` și `geteuid`, `getegid`. Interfețele acestor funcții:

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

Primitive utile referitoare la procese (2/3)

- Primitive de suspendare a execuției pe o durată de timp specificată: `sleep` și `usleep`. Interfețele acestor funcții:
  `unsigned int sleep(unsigned int` *nr_secunde*`);`
  `int usleep(useconds_t` *nr_microsecunde*`);`
  Exemplu: a se vedea programul demonstrativ `info_ex.c`

  Notă: în standardul POSIX s-a introdus și primitiva `nanosleep`, cu o precizie de ordinul nanosecunde și mai eficientă.
  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; pentru detalii citiți man 7 time.

Primitive utile referitoare la procese (3/3)

- Primitiva de terminare normală a execuției unui proces: `exit`.
  Interfața acestei funcții: `void exit(int status);`
  Efect: procesul apelant își încheie execuția normal, iar valoarea *status* este "trunchiată" (i.e., *status* & `0xFF`) și returnată drept *cod_retur* către părintele procesului respectiv.
- Primitiva de terminare anormală a execuției unui proces: `abort`.
  Interfața acestei funcții: `void abort(void);`
  Efect: deblochează și apoi își livrează semnalul `SIGABRT`, ceea ce cauzează terminarea anormală a procesului.
- Funcția `system` permite lansarea de comenzi `UNIX` dintr-un program C, printr-un apel de forma: `system(comanda);`
  Efect: se creează un nou proces, în care se încarcă *shell*-ul implicit, ce va executa comanda specificată.

Crearea proceselor – primitiva `fork`

Singura modalitate de creare a proceselor în `UNIX`/`Linux` este cu ajutorul apelului sistem `fork`. Prototipul lui este următorul:

```
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`.

Singura diferență dintre procese va fi valoarea returnată de funcția `fork`, precum și, bineînțeles, `PID`-urile proceselor.
Procesul apelant va fi *părintele* procesului nou creat, iar acesta va fi *fiul* procesului apelant (mai exact, unul dintre procesele fii ai acestuia).

Crearea proceselor – primitiva `fork`

Observație importantă:

Datorită acestei operații de “clonare”, imediat după apelul `fork` procesul fiu va avea aceleași valori ale variabilelor din program și aceleași fișiere deschise ca și procesul părinte. Mai departe însă, *fiecare proces va lucra pe propria sa zonă de memorie*.

Deci, dacă fiul modifică valoarea unei variabile, această modificare nu va fi vizibilă și în procesul tată (și nici invers).

În concluzie, 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`.

Crearea proceselor – primitiva `fork`

Valoarea returnată:

Apelul `fork` returnează valoarea `-1`, în caz de eroare (dacă nu s-a putut crea un nou proces), 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

Pe baza acestei valori returnate, ce diferă în cele două procese, se poate ramifica execuția astfel încât fiul să execute altceva decât tatăl.

Exemplu: a se vedea programul demonstrativ `fork_ex.c`

Crearea proceselor – primitiva `fork`

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.
- 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.)

Terminarea proceselor

Procesele se pot termina în două moduri:

- *terminarea normală*: se petrece în momentul întâlnirii în program a apelului primitivei `exit` (sau la sfârșitul funcției *main*).
  Ca efect, 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.
  Codul de terminare este salvat în intrarea corespunzătoare procesului respectiv din tabela proceselor; intrarea respectivă nu este dealocată ("șterasă") imediat din tabelă, astfel încât codul de terminare a procesului respectiv să poată fi furnizat procesului părinte la cererea acestuia.
- *terminarea anormală*: se petrece în momentul primirii unui semnal de un anumit tip (e.g., semnalul generat cu un apel `abort`).
  Notă: nu chiar toate tipurile de semnale cauzează terminarea procesului.
  Și în acest caz se dealocă zonele de memorie ocupate de procesul respectiv, și se păstrează doar intrarea sa din tabela proceselor până când părintele său va cere codul de terminare (reprezentat în acest caz de numărul semnalului ce a cauzat terminarea anormală).

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.*, program) locul unde se găsește acest punct de sincronizare.

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`).

Primitiva `wait`

Un alt exemplu de sincronizare, des întâlnită în practică:
Procesul părinte poate avea nevoie de valoarea de terminare returnată de procesul 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` (cont.)

Apelul sistem `wait` este utilizat pentru a aștepta un proces fiu să-și termine execuția.

Interfața acestei funcții:

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

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). Dacă există deja vreun fiu care s-a terminat sau a fost stopat, atunci funcția `wait` returnează imediat.

Primitiva `wait` (cont.)

Valoarea returnată:

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

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

Notă: pentru a inspecta valoarea stocată în `*`*stat_loc* pot fi folosite *macro*-urile `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFSTOPPED`, `WSTOPSIG` ș.a.

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`).

Primitiva `wait` (cont.)

Observație: dacă procesul părinte se termină înaintea vreunui proces fiu, atunci acestui fiu i se va atribui ca părinte procesul `init` (ce are `PID`-ul 1), iar acest lucru se face pentru toate procesele fii neterminate în momentul terminării părintelui lor. Iar dacă un proces se termină înaintea părintelui său, atunci el devine *zombie*.

Mai există o primitivă, cu numele `waitpid`, care va aștepta terminarea fie 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` e utilă pentru a testa fără așteptare existența vreunui fiu deja terminat).

Exemple: a se vedea programele demonstrative `wait_ex1.c` , `wait_ex2.c` și `wait_ex3.c`

Șabloane de cooperare și sincronizare

- Șablonul de cooperare 'Supervisor/workers' (aka 'Master/slaves'):
  Este un șablon de calcul paralel aplicabil atunci când avem o problemă complexă a cărei rezolvare se poate "diviza" în mai multe sub-probleme **ce pot fi apoi rezolvate, în paralel, independent una de alta**, iar la final rezultatele parțiale obținute pot fi "agregate" pentru a obține rezultatul final al problemei inițiale.
- Șablonul de sincronizare 'Token ring':
  Este un șablon de sincronizare care surprinde următoarea situație: avem un număr oarecare p de procese, fiecare având de executat, în mod repetitiv, câte o acțiune specifică A_i , cu $i = 1, ..., p$, și se cere sincronizarea execuției lor în paralel, astfel încât ordinea de execuție (i.e., *trace*-ul) să fie precis următoarea: $A_1, A_2, \ldots, A_p$, repetată de un anumit număr de ori.
- Alte șabloane: 'Producer-Consumer', 'CREW', ș.a. (pentru detalii, recitiți cursul teoretic #6).

Bibliografie obligatorie

Cap.4, §4.1, §4.2 și §4.3 din manualul, în format PDF, accesibil din pagina disciplinei “Sisteme de operare”:

- `https://profs.info.uaic.ro/~vidrascu/SO/books/ManualID-SO.pdf`

Programele demonstrative amintite pe parcursul acestei prezentări pot fi descărcate de la adresele următoare:

- `https://profs.info.uaic.ro/~vidrascu/SO/cursuri/C-programs/fork/`
- `https://profs.info.uaic.ro/~vidrascu/SO/cursuri/C-programs/wait/`