Sisteme de operare

Laborator 4

Semestrul I 2025-2026

Laborator 4

- controlul proceselor
- crearea proceselor
- executia programelor
- terminarea proceselor
- race conditions

Crearea proceselor

- primul proces din ierarhia de procese *init* (*systemd* in Linux) creat de kernel, la sfarsitul secventei de boot
- apoi *init* devine "stramosul" tututor proceselor din sistem
- toate aceste procese sunt create la fel, si anume invocand apelul sistem *fork*

pid_t fork(void);

- **singurul** mod prin care sistemele de tip Unix pot crea un nou proces este atunci cand un proces **existent** apeleaza *fork*
  - procesul apelant se cheama *parinte*
  - procesul nou creat se chema *copil*
- *fork* se cheama o singura data, si se intoarce de doua ori !
- discriminare la nivel de cod, intre procesul parinte si copil pe baza valorii de retur a apelului *fork*
  - in procesul parinte *fork* intoarce valoarea PID a noului proces
  - in procesul copil *fork* intoarce valoarea 0

Crearea proceselor (cont.)

- secventa tipica de apel

```
pid_t pid;
…
if((pid = fork()) < 0)
        // cod de tratare a erorii
else if(!pid)
        // pid == 0, cod copil
    else
        // pid > 0, cod parinte
// aici e cod comun, executat atat de parinte cat si de copil
```

Crearea proceselor (cont.)

- la intoarcerea din apelul *fork*, atat parintele cat si copilul continua cu instructiunea imediat urmatoare apelului *fork*
- procesul copil este o copie a procesului parinte
  - copilul obtine o copie **separata** a sectiunilor de date, heap si stack ale parintelui
  - daca sectiunea de cod (text) e RO, poate fi partajata de cele doua procese
- copierea segmentelor de memorie e costisitoare => in practica se foloseste COW (Copy-On-Write)
  - imediat dupa *fork* cele doua procese partajeaza intreaga memorie a programului (marcata RO de catre kernel)
  - cand unul dintre procese incearca sa modifice o zona de memorie se face o copie separata a acelei zone de memorie, a.i. fiecare proces are acum propria copie a zonei modificate
  - “zona” e un termen generic, vom vedea cand vorbim de memoria virtuala ca de fapt e vorba de o *pagina de memorie*
- regula generala: dupa *fork*, nu exista nici o garantie asupra ordinii de executie a celor doua procese, parinte si copil !
- Obs: in Linux, *fork* e un wrapper pentru *clone*

Partajarea fisierelor dupa fork

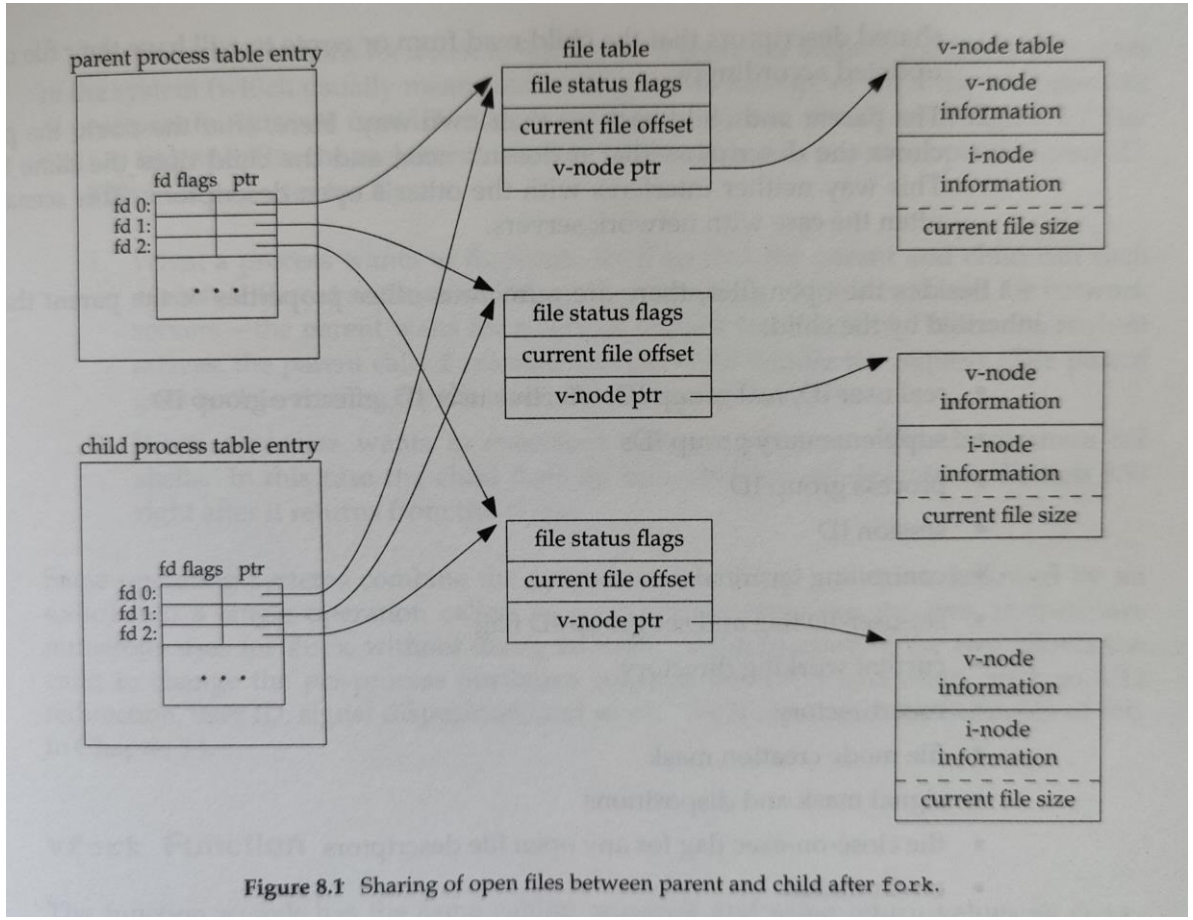

Figure 8.1 Sharing of open files between parent and child after `fork`.

Partajarea fisierelor dupa fork (cont.)

- in general, dupa *fork* exista doua variante
  - parintele asteapta terminare executiei copilului
    - parintele nu trebuie sa faca nimic cu descriptorii de fisiere, offset-urile partajate sunt automat actualizate de copil
  - parintele si copilul continua independent
    - de regula, fiecare proces inchide descriptorii de fisiere pe care nu ii foloseste pentru a nu interfera cu activitatea celuilalt proces
- alte informatii mostenite de copil de la parinte (pe langa descriptorii de fisiere deschise)
  - ID-uri reale si efective
  - directorul curent de lucru
  - root directory
  - umask
  - variabilele de mediu
  - samd.

Utilizarea fork

- doua moduri principale de lucru

1. un proces se duplica pentru ca parintele si copilul sa poata executa concurent diferite sectiuni ale codului
   - tipic pt. serverele de retea:
     - parintele asteapta o cerere client
     - la sosirea cererii creaza un nou proces cu *fork*
     - copilul tratateaza cererea
     - parintele continua sa astepte noi cereri client
2. un proces vrea sa execute un program diferit
   - tipic shell-urilor: imediat dupa *fork*, copilul executa un apel sistem *exec* care incarca un nou program de pe disc

- unele SO combina scenariul *fork-exec* intr-o singura operatie *spawn*

Vfork

- scenariul *fork-exec* ridica problema copierii inutile a sectiunilor de memorie ale parintelui
  - copilul incarca un nou program, fara legatura cu procesul parinte
- optimizare: apelul sistem *vfork*
  - aceeasi semnatura ca si *fork*, dar semantica diferita
  - dupa *vfork*, procesul copil partajeaza spatiul de adrese al parintelui pana cand cheama fie *exec* fie *exit*
  - parintele asteapta pana cand copilul cheama fie *exec* fie *exit*

  => e garantat ca dupa *vfork* copilul ruleaza primul !
  - consecinta grava: daca procesul copil se blocheaza inainte de a chema *exec* sau *exit* si deblocarea depinde de procesul parinte => **deadlock** !

Terminarea proceselor

- terminare normala
  - procesul executa instructiunea *return* a functiei *main*
  - procesul apeleaza explicit *exit/_exit*
- terminare anormala
  - prin apelul explicit al functiei *abort* (de fapt genereaza SIGABRT)
  - cand procesul primeste anumite semnale (ex: SIGSEGV, SIGFPE, SIGKILL, etc)
- indiferent de tipul terminarii procesului, codul kernel elibereaza resursele detinute de proces:
  - inchide fisierele deschise, elibereaza memoria alocata, samd
- procesul parinte este notificat printr-un cod de stare de tipul terminarii procesului
  - codul de stare este fie valoarea de return din *main*, fie parametrul lui *exit*, fie generat de kernel in cazul terminarii anormale
- procesul parinte recupereaza acest cod de stare cu ajutorul apelurilor sistem de tip *wait/waitpid*

Terminarea proceselor (cont.)

- cand procesul parinte se termina inainte de procesul copil, *init* devine parintele copilului *orfan*
  - spunem ca *init* mosteneste procesul copil
  - cand un proces se termina, kernelul verifica daca are copii si, in caz afirmativ, toti sunt mosteniti de *init*, a.i. ID-ul procesului lor parinte devine 1
- daca procesul copil dispare inainte de procesul parinte
  - apare posibilitatea ca parintele sa nu mai astepte dupa copil
  - kernelul salveaza informatie minimala despre procesele copil terminate
    - PID, cod de terminare, timpul CPU al procesului terminat
    - elibereaza resursele detinute (inchide fisierele deschise, elibereaza memoria, etc)
    - aceste informatii pot fi recuperate de procesul parinte daca apeleaza *wait/waitpid*
  - daca parintele nu apeleaza niciodata *wait/waitpid*, procesul copil intra in starea de *zombie*
    - procesele *zombie* trebuie terminate explicit din shell cu comanda *kill*

  Obs: copii lui *init* nu ajung niciodata *zombie*, *init* se ingrijeste sa apeleze *wait* pentru fiecare dintre copii sai care termina executia

Asteptarea terminarii proceselor

- cand un proces se termina, kernelul trimite procesului parinte (eventual *init*) un semnal SIGCHLD
  - semnal = notificare a unui eveniment asincron
- parintele poate ignora semnalul, sau ii poate asocia un handler
  - actiunea implicit asociata SIGCHLD este sa fie ignorat
- pentru a recupera informatiile legate de terminarea procesului copil, parintele trebuie sa apeleze *wait/waitpid*

  *pid_t wait((int *status);*

  *pid_t waitpid(pid_t pid, int *status, int options);*

- consecinte posibile ale apelurilor *wait*:
  - procesul apelant se blocheaza (daca toti copiii sai ruleaza)
  - procesul se intoarce imediat din apel daca un proces copil s-a terminat si asteapta sa-i fie recuperata starea
  - intoarce eroare daca procesul apelant nu are copii

Asteptarea terminarii proceselor (cont.)

- diferente *wait* vs *waitpid*
  - *wait* poate bloca apelantul (daca toti copiii ruleaza), *waitpid* are o optiune de apel neblocant
  - *wait* asteapta dupa primul proces copil care se termina, *waitpid* are o serie de optiuni care dicteaza terminarea carui proces copil o asteapta
- *wait* se intoarce imediat daca un proces copil e *zombie*, iar argumentul apelului contine codul de stare al terminarii, altfel se blocheaza pana cand un copil termina
  - valoarea de retur este PID-ul procesului copil care s-a terminat
  - daca sunt mai multi copii in rulare, se asteapta terminarea primului
- Q: cum asteptam terminarea unui proces anume?
- A: cu *wait* si testam valoarea de retur intr-o bucla, sau cu *waitpid*

Semantica waitpid

- *pid* == -1 => asteapta terminarea oricarui proces copil
- *pid* > 0 => asteapta terminarea procesului *pid*
- *pid* == 0 => asteapta terminarea oricarui copil al carui GID de proces este egal cu cel al procesului apelant (parinte)
- *pid* < -1 => asteapta terminarea oricarui copil al carui GID de proces este egal cu *abs(pid)*
- optiuni
  - 0 sau un OR logic cu urmatoarele constante
  - WNOHANG: *waitpid* nu se blocheaza daca procesul specificat de *pid* nu s-a terminat
  - WUNTRACED: daca procesul specificat de *pid* e stopat (job control), starea sa e retunata daca daca nu a fost raportata de cand s-a oprit
    - macro-ul WIFSTOPPED specifica daca valoarea de retur este a unui proces copil oprit

Race conditions

- situatie in care executia intretesuta (interleaved) a mai multor procese care acceseaza o resursa partajata produce rezultate nedeterministe
  - accesul mai multor procese la date partajate genereaza rezultate care depind de ordinea in care procesele ruleaza
- dupa *fork*, nu se poate face nici o presupunere cu privire la ce proces ruleaza primul, parintele sau copilul
- in general, nu se pot face presupuneri cu privire la ordinea de rulare a proceselor
  - depinde de incarcarea curenta a sistemului si deciziile planificatorului de procese din kernel
- Obs: apeluri de tip *sleep* nu garanteaza nici ele ordinea executiei
  - ex: daca sistemul este foarte incarcat, e posibil ca procesul care a apelat *sleep* sa se trezeasca inaintea altor procese care n-au rulat intre timp
- solutie generala: mecanisme de sincronizare a proceselor