PROGRAMARE DE SISTEM ÎN C PENTRU PLATFORMA LINUX (II)

Gestiunea fișierelor, partea a II-a:

Accesul concurent sau exclusiv la fișiere. Blocaje pe fișiere

Cristian Vidrașcu
cristian.vidrascu@info.uaic.ro

Aprilie, 2025

Sumar

Introducere

Deoarece sistemele de operare din familia `UNIX` (în particular, și `Linux`-ul) sunt sisteme *multi-tasking* (*i.e.*, sisteme care suportă execuția "simultană" a mai multor programe), în mod uzual este permis *accesul concurent* la fișiere, adică mai multe procese pot accesa "simultan" în citire și/sau în scriere un același fișier, sau chiar o aceeași înregistrare dintr-un fișier.

Acest mod de acces concurent ("simultan") la un fișier de către procese diferite poate avea însă uneori și efecte nedorite (ca, de exemplu, distrugerea integrității datelor din fișier, datorită *data race*-urilor).

Din acest motiv, în sistemele din familia `UNIX` s-au implementat mecanisme care să permită și un mod de *acces exclusiv* la fișiere, adică un mod de acces în care un singur proces are, la un moment dat, permisiunea de acces la un fișier, sau chiar la o anumită înregistrare dintr-un fișier.

Agenda

Demo (1): Un exemplu de acces concurent la un fișier

Observație: d.p.d.v. al programatorului, acesta nu trebuie să utilizeze nicio tehnică suplimentară celor discutate în lecția precedentă despre accesul la fișiere, pentru a "beneficia" de accesul în mod concurent ("simultan") la un fișier. Totul se petrece la momentul execuției: dacă utilizatorul rulează în același timp două sau mai multe instanțe de programe ce accesează în mod uzual un același fișier, atunci accesele la fișier se vor petrece "simultan" (*i.e.*, aproximativ în același timp).

Iată un exemplu de program ce poate fi utilizat pentru a ilustra efectele accesului concurent la un fișier: a se vedea programul `access_v1.c` ([2]).

Mai întâi, un demo de execuție ce ilustrează *accesul secvențial la fișier*, *i.e.* un singur proces dorește să acceseze fișierul într-un anumit interval de timp.
Creăm un fișier `fis.dat` ce conține următoarea linie de text: `aaaa#bbbb#cccc#dddd#eeee`
Apoi lansăm în execuție secvențială mai multe instanțe ale acestui program, *e.g.* prin comanda:

```
UNIX> ./access_v1 1 ; ./access_v1 2 ; ./access_v1 3
```

Care va fi conținutul fișierului după terminarea execuției acestei comenzi ?

După execuția primei instanțe, fișierul va arăta astfel: `aaaa1bbbb#cccc#dddd#eeee`

După execuția instanței a doua, fișierul va arăta astfel: `aaaa1bbbb2cccc#dddd#eeee`

După execuția instanței a treia, rezultatul final va arăta astfel: `aaaa1bbbb2cccc3dddd#eeee`

Demo (1): Un exemplu de acces concurent la un fișier (cont.)

Iar acum, un demo de execuție ce ilustrează *accesul concurent la fișier*: mai multe procese (*i.e.*, instanțe ale programului) ce doresc să acceseze fișierul în același interval de timp.
"Reinițializăm" fișierul `fis.dat` cu următoarea linie de text: `aaaa#bbbb#cccc#dddd#eeee`
Apoi lansăm în execuție paralelă ("simultană") două instanțe ale acestui program, prin comanda:

```
UNIX> ./access_v1 1 & ./access_v1 2 &
```

Care va fi conținutul fișierului după terminarea execuției acestei comenzi ?

Probabil vă așteptați ca după execuție fișierul să arate astfel:
`aaaa1bbbb2cccc#dddd#eeee` sau `aaaa2bbbb1cccc#dddd#eeee`
(în funcție de care dintre cele două procese a reușit mai întâi să suprascrie primul caracter '#' din acest fișier, celuilalt proces rămânându-i al doilea caracter '#' pentru a-l suprascrie.)

În realitate, repetând de oricâte ori execuția acestei comenzi, întotdeauna se va obține:
`aaaa1bbbb#cccc#dddd#eeee` sau `aaaa2bbbb#cccc#dddd#eeee`

Motivul: datorită apelului `sleep(5)` care provoacă o așteptare de 5 secunde între momentul depistării primei înregistrări din fișier care este '#' și momentul suprascrierii acestei înregistrări cu alt caracter.

Observație: prin eliminarea apelului `sleep(5)` din program, repetând execuția acestei comenzi de un număr suficient de mare de ori, se pot obține toate cele 4 rezultate de mai sus, cu frecvențe diferite de observare.

Demo: pentru explicații mai detaliate, a se vedea [FirstDemo] prezentat în suportul de laborator #7.

Modul de acces exclusiv la fișiere – Blocaje pe fișiere 7 / 21

Agenda

Structura de date flock pentru blocaje

Sistemele din familia UNIX furnizează programatorilor un mecanism de *blocare* (*i.e.*, de punere de "*lacăte*") pe porțiuni de fișier pentru accesul în mod exclusiv.
Prin acest mecanism se definește o zonă de *acces exclusiv* în fișier. O asemenea porțiune nu va putea fi accesată în mod concurent de mai multe procese pe toată durata de existență a blocajului.

Pentru a specifica un blocaj (*i.e.*, un "*lacăt*") pe o porțiune dintr-un fișier (sau pe întregul fișier), se utilizează structura de date `flock`, definită în fișierul header `fcntl.h` în felul următor:

```
struct flock
{
  short l_type;   // indica tipul blocarii
  short l_whence; // indica pozitia relativa (originea)
  long  l_start;  // indica pozitia de start, in raport cu originea
  long  l_len;    // indica lungimea portiunii blocate
  int   l_pid;
}
```

Observație: după ce se completează câmpurile structurii de mai sus, ulterior se va apela funcția `fcntl` pentru a pune efectiv "lacătul" pe porțiunea respectivă din fișier.

8 / 21

Structura de date flock pentru blocaje (cont.)

Semnificația câmpurilor structurii `flock`:

- câmpul `l_type` indică tipul blocării, putând avea ca valoare una dintre constantele:
  - `F_RDLCK` : blocaj în citire
  - `F_WRLCK` : blocaj în scriere
  - `F_UNLCK` : deblocaj (*i.e.*, se înlătură lacătul)
- câmpul `l_whence` indică poziția relativă (*i.e.*, originea) în raport cu care este interpretat câmpul `l_start`, putând avea ca valoare una dintre următoarele constante simbolice:
  - `SEEK_SET` (=0) : originea este BOF (*i.e.*, *beginning of file*)
  - `SEEK_CUR` (=1) : originea este CURR (*i.e.*, *current position in file*)
  - `SEEK_END` (=2) : originea este EOF (*i.e.*, *end of file*)
- câmpul `l_start` indică poziția (*i.e.*, *offset*-ul în raport cu originea `l_whence`) de la care începe porțiunea blocată.
  Observație: `l_start` trebuie să fie negativ pentru `l_whence=SEEK_END`.
- câmpul `l_len` indică lungimea în octeți a porțiunii blocate.
- câmpul `l_pid` este gestionat de funcția `fcntl` care pune blocajul, fiind utilizat pentru a memora PID-ul procesului proprietar al acelui lacăt.
  Observație: are sens consultarea acestui câmp doar atunci când funcția `fcntl` se apelează cu parametrul `F_GETLK`.

9 / 21

Primitiva fcntl pentru blocaje

Interfața funcției `fcntl` ([5] – una dintre ele, cea pentru blocaje):
`int fcntl(int df, int cmd, struct flock* p_flock)`

- *df* = descriptorul de fișier deschis pe care se pune lacătul
- *p_flock* = adresa structurii `flock` ce definește acel lacăt
- *cmd* = indică modul de punere, putând lua una dintre valorile:
  - `F_SETLK` : permite punerea unui lacăt pe fișier, în citire sau în scriere, sau scoaterea unuia deja pus (funcție de tipul specificat în structura `flock`).
    Observație: în caz de eșec datorită conflictului cu alt lacăt deja pus, se setează variabila `errno` la valoarea `EACCES` sau `EAGAIN`.
  - `F_SETLKW` : permite punerea lacătelor în mod "blocant", adică se așteaptă (*i.e.*, funcția nu returnează) până când se poate pune lacătul. Motivul posibil de așteptare: se încearcă blocarea unei zone deja blocate de un alt proces.
  - `F_GETLK` : permite extragerea informațiilor despre un lacăt pus pe fișier.
- valoarea returnată este `0` pentru blocaj reușit, sau `-1` în caz de eroare.

10 / 21

Primitiva fcntl pentru blocaje (cont.)

Observații:

- Pentru a putea pune un lacăt în citire, respectiv în scriere, pe un descriptor de fișier, acesta trebuie să fi fost anterior deschis în citire, respectiv în scriere.
- Blocajul este scos automat atunci când procesul care l-a pus închide acel fișier, sau își termină execuția.
- Scoaterea (deblocarea) unui segment dintr-o porțiune mai mare anterior blocată poate produce două segmente blocate.
- Câmpul `l_pid` din structura `flock` este actualizat de funcția `fcntl`.
- Lacătele nu se transmit proceselor fii în momentul creării acestora cu funcția `fork`.
  Motivul: fiecare lacăt are în structura `flock` asociată PID-ul procesului care l-a creat (și care este deci proprietarul lui), iar procesele fii au, bineînțeles, PID-uri diferite de cel al părintelui.
- În `Linux` mai există alte două interfețe ce oferă lacăte pe fișiere ([5]):
  - funcția `flock` → pentru detalii consultați documentația: `man 2 flock`
  - funcția `lockf` → pentru detalii consultați documentația: `man 3 lockf`
- Există și două comenzi utile pentru lacăte: `flock` și `lslocks` ([6]).

11 / 21

Demo (2): Un exemplu de acces exclusiv la un fișier

Putem rescrie programul anterior, adăugând utilizarea de lacăte în scriere pentru a "inhiba" accesul concurent la fișier: a se vedea programul `access_v2.c` ([2]).

"Reinițializăm" fișierul `fis.dat` cu următoarea linie de text: `aaaa#bbbb#cccc#dddd#eeee`

Apoi lansăm în execuție paralelă ("simultană") două instanțe ale acestui program, prin comanda:

`UNIX> ./access_v2 1 & ./access_v2 2 &`

Care va fi conținutul fișierului după terminarea execuției acestei comenzi ?

De data aceasta, oricâte execuții s-ar face, întotdeauna se va obține rezultatul urmărit:

`aaaa1bbbb2cccc#dddd#eeee` sau `aaaa2bbbb1cccc#dddd#eeee`

Observație: în programul de mai sus apelul de punere a lacătului este neblocant (*i.e.*, cu parametrul `F_SETLK`). Se poate face și un apel blocant, *i.e.* funcția `fcntl` nu va returna imediat, ci va sta în așteptare până când reușește să pună lacătul.

A se vedea programul `access_v2w.c`

Lansând simultan în execuție două instanțe ale acestui program, se va constata că obținem același rezultat ca și în cazul variantei neblocante.

Demo: pentru explicații mai detaliate, a se vedea [SecondDemo] prezentat în suportul de laborator #7.

Caracteristici ale blocajelor pe fișiere

- Important: lacătele în scriere (*i.e.*, cele cu tipul `F_WRLCK`) sunt *exclusive*, iar cele în citire (*i.e.*, cele cu tipul `F_RDLCK`) sunt *partajate*, în sensul CREW ("Concurrent Read or Exclusive Write").
  Cu alte cuvinte: în orice moment, pentru orice porțiune dintr-un fișier, cel mult un proces poate deține un lacăt în scriere pe acea porțiune (și atunci nici un proces nu poate deține concomitent vreun lacăt în citire), sau este posibil ca mai multe procese să dețină lacăte în citire pe acea porțiune (și atunci nici un proces nu poate deține concomitent vreun lacăt în scriere).
- Important: blocajele puse pe fișiere sunt *advisory*, nu sunt *mandatory*!
  Cu alte cuvinte: funcționarea corectă a lacătelor în scriere se bazează pe *cooperarea proceselor* pentru asigurarea accesului exclusiv la fișiere, *i.e.* toate procesele care vor să acceseze mutual exclusiv un fișier (sau o porțiune dintr-un fișier) vor trebui să folosească lacăte în scriere pentru accesul respectiv.

  Altfel, spre exemplu, dacă un proces scrie direct un fișier, sau o porțiune dintr-un fișier (și are permisiunile de acces necesare), apelul său de scriere NU va fi împiedicat de un eventual lacăt în scriere sau în citire pus pe acel fișier, sau pe acea porțiune de fișier, de către un alt proces.

  Și lacătele în citire au un caracter *advisory*, *i.e.* putem scrie (și citi) fișierul (dacă avem permisiunile de acces necesare) în timp ce un alt proces deține un lacăt în citire pe acel fișier.

Demo (3): Ilustrarea caracterului *advisory* al blocajelor

Iată o justificare a observației anterioare despre caracterul *advisory* al blocajelor:

"Reinițializăm" fișierul `fis.dat` cu linia de text: `aaaa#bbbb#cccc#dddd#eeee`

și apoi rulăm următoarea comandă:

```
UNIX> ./access_v2 1 & sleep 2 ; echo "xyxyxy" > fis.dat
```

Care va fi conținutul fișierului după terminarea execuției acestei comenzi ?

Răspuns: la finalul execuției acestei comenzi, fișierul `fis.dat` va conține linia de text: `xyxy1y` , ceea ce ne demonstrează că suprascrierea executată de comanda `echo` în fișier s-a petrecut în intervalul de timp al celor 5 secunde în care instanța `access_v2` deținea blocajul pe fișier!

* * *

```
UNIX> ./access_v2 1 & sleep 2 ; cat fis.dat
```

Rulând această comandă, vom observa că citirea efectuată de comanda `cat` reușește, deși fișierul era blocat la momentul citirii.

Demo: pentru explicații mai detaliate, a se revedea ultima parte din [SecondDemo].

Demo (4): Un exemplu de acces exclusiv *optimizat* la un fișier

Observație importantă: a doua versiune a programului demonstrativ (ambele variante, și cea neblocantă, și cea blocantă) nu este optimă:

Practic, cele două procese (*i.e.*, cele două instanțe ale programului executate în paralel) își fac treaba *secvențial*, unul după altul, și nu concurent, deoarece de abia după ce se termină acel proces care a reușit primul să pună lacăt pe fișier, va putea începe și celălalt proces să-și facă treaba (*i.e.*, parcurgerea fișierului și înlocuirea primului caracter '#' întâlnit).

* * *

Această observație ne sugerează că putem *îmbunătăți timpul total de execuție* permițând celor două procese să se execute într-adevăr concurent, iar pentru aceasta trebuie să punem lacăt doar pe un singur caracter (și anume pe prima poziție din fișier la care întâlnim caracterul '#') și să păstrăm blocajul doar pe durata minimă necesară pentru a face suprascrierea, în loc să blocăm tot fișierul, tot timpul – încă de la început și până la finalul execuției programului.

Demo (4): Un exemplu de acces exclusiv *optimizat* la un fișier (cont.)

Versiunea a treia, cu blocaj la nivel de caracter și de durată minimală:

Implementarea acestei optimizări: programul va trebui sa facă următorul lucru – când întâlnește primul caracter '#' în fișier, pune lacăt pe el (*i.e.*, pe exact un caracter) și apoi îl rescrie: a se vedea programul (în varianta blocantă) `access_v3.c` ([2]).

În acest caz, care credeți că va fi conținutul fișierului după terminarea execuției în paralel a două instanțe ale acestei versiuni a programului ?

* * *

Observație: ideea de rezolvare aplicată în programul `access_v3.c` nu este întrutotul corectă, în sensul că nu se va obține întotdeauna rezultatul scontat, deoarece între momentul depistării primei poziții a unui caracter '#' în fișier și momentul reușitei blocajului există posibilitatea ca acel '#' să fie suprascris de cealaltă instanță executată în paralel !

Notă: tocmai pentru a forța apariția unei situații care cauzează producerea unui rezultat nedorit, am introdus în program acel apel `sleep(5)` între punerea blocajului pe caracterul '#' și rescrierea lui.

Cum se poate remedia acest neajuns al programului `access_v3.c` ? $\rightarrow$ $\rightarrow$ $\rightarrow$

Demo (4): Un exemplu de acces exclusiv *optimizat* la un fișier (cont.)

$\rightarrow$ $\rightarrow$ Acest neajuns al programului `access_v3.c` se poate corecta astfel:

După punerea blocajului, se verifică din nou dacă acel caracter este într-adevăr '#' (pentru că între timp s-ar putea să fi fost rescris de cealaltă instanță executată în paralel) și, dacă nu mai este '#', atunci trebuie scos blocajul și reluată bucla de căutare a primului caracter '#' întâlnit în fișier.

v4 $\rightarrow$ *Temă*: adăugați această corecție la programul `access_v3.c` .

* * *

Rezolvare: dacă nu reușiți să corectați singuri programul, iată soluția: `access_v4.c` .

Demo: pentru explicații mai detaliate despre această variantă mai eficientă a programului demonstrativ, a se vedea [ThirdDemo] prezentat în suportul de laborator #7.

* * *

Suplimentar, a se vedea exemplele de utilizare a comenzii `lslocks` pentru observarea lacătelor active la diverse momente din execuția jobului paralel respectiv, prezentate în [SecondDemo] și [ThirdDemo] din suportul de laborator #7.

Aplicație: implementarea unui semafor binar

Cum am putea implementa un semafor binar folosind blocaje pe fișiere ?

O posibilă implementare ar consta în următoarele idei:

Inițializarea semaforului s-ar realiza prin crearea unui fișier de tip normal, de către un proces cu rol de *supervizor* (acesta poate fi oricare dintre procesele cooperante ce vor folosi acel semafor, sau poate fi un proces separat). Noul fișier va avea un nume unic prin care să se identifice semaforul în cadrul grupului de procese cooperante.

Acest proces *supervizor* va scrie inițial în fișier 1 octet oarecare (nu este importantă lungimea fișierului, deoarece vom pune blocaj în scriere pe întreg fișierul pentru a "simula" un semafor binar).

Operația `wait` va consta în punerea unui blocaj în scriere pe fișier, cu un apel blocant (*i.e.*, utilizând operația `F_SETLKW` în apelul funcției `fcntl`).
Operația `signal` va consta în scoaterea blocajului de pe fișier.

Temă: implementați în C un semafor binar pe baza ideilor de mai sus și scrieți un program demonstrativ în care să utilizați semaforul astfel implementat pentru asigurarea excluderii mutuale a unei secțiuni critice de cod (pentru "inspirație" în scrierea programului demonstrativ, revedeți problemele de sincronizare discutate în cursurile teoretice #5 și #6).

Durata de execuție a unui program

Agenda

19 / 21

Metode de măsurare a timpului de execuție

Există mai multe posibilități de măsurare a duratei de execuție a unui program:

- Comanda internă `time` a interpretorului de comenzi `bash`:
  `UNIX> time ./MyProgram parameters`
  Output: durata totală de execuție, precum și timpul consumat în starea *running*, în *user-mode* și *kernel-mode*, cu precizie de milisecunde.
- Comanda externă `/usr/bin/time`:
  `UNIX> /usr/bin/time ./MyProgram parameters`
  Output: durata totală de execuție, timpul consumat în starea *running*, în *user-mode* și *kernel-mode*, precum și diverse alte informații statistice despre tipurile de resurse consumate pentru execuția programului, specificate prin intermediul opțiunii `--format`.
- Apelul de sistem `gettimeofday()`, având interfața următoare:
  `int gettimeofday(struct timeval *tv, struct timezone *tz)`
  Returnează în primul parametru ora curentă, cu o precizie de microsecunde.
- Apelul de sistem `clock_gettime()`, având interfața următoare:
  `int clock_gettime(clockid_t clockid, struct timespec *tp)`
  Apelat cu ceasul `CLOCK_REALTIME` în primul parametru, returnează în al doilea parametru ora curentă, cu o precizie de nanosecunde.

Observație: pentru explicații suplimentare despre aceste metode de măsurare a timpului, a se citi ultima secțiune din suportul de laborator #7.

Bibliografie obligatorie

[1] Cap. 3, §3.2 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/flock/

[3] Suportul de laborator online asociat acestei prezentări:

- https://edu.info.uaic.ro/sisteme-de-operare/SO/support-lessons/C/suport_lab7.html

Bibliografie suplimentară:

[4] Cap. 55 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 : `man 2 fcntl` , `man 2 flock` și `man 3 lockf` .

[6] Documentația comenzilor pentru lacăte: `man 1 flock` și `man 8 lslocks` .

21 / 21