Laboratorul 2 - Elemente de sincronizare în Pthreads

Responsabili: Radu Ciobanu, Andrei Damian, Delia Stuparu, Dragoș Cocîrlea

Introducere

Când vorbim de programarea paralelă, putem avea situații în care mai multe fire de execuție care rulează în paralel vor să acceseze simultan aceleași resurse. De exemplu, putem avea situația următoare. Avem două thread-uri (**T0** și **T1**) care au acces partajat la o variabilă întreagă *a* inițializată cu 0. În funcția de thread, atât **T0** cât și **T1** incrementează *a* cu 2. În mod normal, ne-am aștepta ca valoarea variabilei *a* după execuția programului nostru să fie 4, pentru că avem două incrementări ale variabilei în cele două thread-uri.

În realitate, situația nu este întotdeauna așa. Dacă am traduce incrementarea unei variabile întregi în cod de asamblare, această operație ar putea arăta în felul următor (în exemplul de mai jos, *eax0* reprezintă registrul *eax* al thread-ului **T0**, iar *eax1* desemnează registrul *eax* al thread-ului **T1**):

T0	T1
load(a, eax0)	load(a, eax1)
eax0 = eax0 + 2	eax1 = eax1 + 2
write(a, eax0)	write(a, eax1)

Putem avea următorul scenariu:

- **T0** citește valoarea lui *a* (0) în propriul registru *eax0*
- în același timp, **T1** citește valoarea lui *a* (tot 0) în propriul registru *eax1*
- **T0** incrementează valoarea lui *eax0*, care devine 2
- **T1** face același lucru, iar *eax1* devine tot 2
- **T0** scrie valoarea din *eax0* în a, care devine 2
- **T1** scrie valoarea din *eax1* în a, care rămâne tot 2.

Se poate deci observa că, în funcție de modul în care thread-urile **T0** și **T1** sunt planificate, este posibil ca rezultatul secvenței de mai sus să fie 2 sau 4. Acest lucru se numește **race condition** și este cauzat de faptul că rezultatul calculului este condiționat de modul de planificare a unor evenimente necontrolabile. Operația de incrementare a lui *a* cu 2 nu este **atomică**, fiind compusă din mai multe operații care se pot intercala atunci când rulăm pe mai multe fire de execuție.

Mutex

Un **mutex** (abreviere de la „mutual exclusion”) este o primitivă de sincronizare prin care putem proteja accesul la date atunci când avem scrieri (potențial) concurente. El funcționează ca un „zăvor” ce protejează accesarea unor resurse partajate.

Un mutex se folosește pentru a delimita o **regiune critică**, adică o zonă a programului în care se poate afla cel mult un thread la un moment dat de timp. Dacă un thread **T1** încearcă să intre într-o regiune critică atunci când alt thread **T0** este deja acolo, **T1** se va bloca până când **T0** va ieși din regiunea critică.

În cadrul exemplului de mai sus, am putea folosi un mutex pentru a defini o regiune critică în jurul operației de incrementare a lui *a*, lucru care ar face imposibilă intercalarea operațiilor celor două thread-uri. Primul thread care intră în regiunea critică va incrementa *a* în mod exclusiv la 2, iar cel de-al doilea thread nu va putea incrementa *a* decât atunci când el este deja 2.

Un mutex are două operații principale: închidere (**lock**) și deschidere (**unlock**). Prin închidere, un thread marchează intrarea în zona critică, adică specifică faptul că orice alt thread care va încerca să facă o operație de închidere va trebui să aștepte. Prin deschidere, se marchează ieșirea din zona critică și deci permisiunea ca un alt thread să intre în

zona critică.

În Pthreads, o secvență tipică de folosire a unui mutex arată în felul următor:

1. se creează și se inițializează o variabilă de tip mutex
2. mai multe thread-uri încearcă să închidă mutexul (adică să intre în zona critică)
3. unul singur dintre ele reușește acest lucru și ajunge să „dețină” mutexul (adică se află în regiunea critică)
4. thread-ul aflat în zona critică realizează diverse operații pe datele protejate
5. thread-ul care deține mutexul iese din zona critică (deschide mutexul)
6. alt thread intră în zona critică și repetă procesul
7. la final, variabila de tip mutex este distrusă.

În Pthreads, un mutex se reprezintă printr-o variabilă de tip *pthread_mutex_t*, și se inițializează folosind următoarea funcție:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

Primul parametru reprezintă o referință la variabila mutex, iar al doilea parametru specifică atributele mutexului nou-creat (dacă se dorește un comportament implicit, parametrul *attr* se poate lăsa NULL).

Pentru a dezaloca un mutex, se folosește următoarea funcție, care primește ca parametru un pointer la mutexul care urmează a fi dezalocat:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Pentru a se face lock pe un mutex, se folosește următoarea funcție, care primește ca parametrul mutexul:

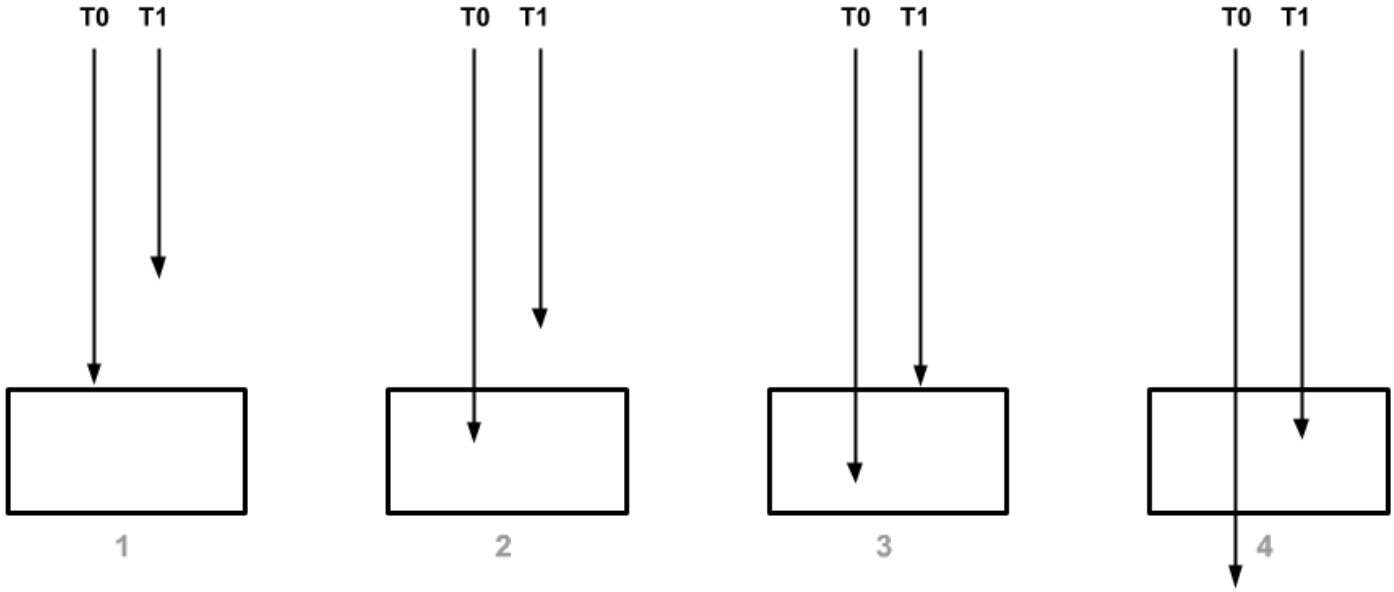
```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Operația inversă, prin care se specifică ieșirea dintr-o zonă critică (adică deschiderea mutexului), este executată prin intermediul următoarei funcții:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Toate patru funcțiile de mutex returnează 0 dacă s-au executat cu succes, sau un cod de eroare în caz contrar.

O reprezentare grafică a funcționării unui mutex se poate vedea în figura de mai jos, într-un scenariu în care avem două thread-uri (**T0** și **T1**) și o regiune critică controlată de mutex (chenarul negru din imagine). La momentul de timp **1** (în partea stângă a figurii), **T0** încearcă să intre în regiunea critică. Pentru că, la momentul respectiv de timp, niciun alt thread nu deține mutexul (adică nu se află în regiunea critică), **T0** intră în regiunea critică (momentul de timp **2**). Mai departe, atunci când **T1** ajunge la intrarea în regiunea critică (încearcă să facă lock pe mutex) la momentul de timp **3**, se blochează pentru că mutexul este curent deținut de **T0** (acesta se află în regiunea critică). Abia în momentul în care **T0** a ieșit din zona critică (la momentul de timp **4**), **T1** se va putea debloca și își va continua execuția.



Atenție! Dacă vrem să protejăm o secțiune din programul nostru folosind un mutex, atunci fiecare thread care accesează acea secțiune trebuie să facă lock și unlock pe aceeași variabilă mutex. De asemenea, dacă un thread vrea să facă unlock pe un mutex pe care nu îl deține (nu a făcut lock pe el în prealabil), va rezulta un comportament nedefinit.

Barieră

O altă primitivă de sincronizare folosită în calculul paralel este **bariera**. Ea are rolul de a se asigura că niciun thread nu poate trece mai departe de punctul în care este plasată decât atunci când toate thread-urile gestionate de barieră ajung în acel punct. Un exemplu de utilizare este atunci când împărțim un calcul pe mai multe thread-uri și vrem să nu mergem mai departe cu execuția programului decât în momentul în care fiecare thread și-a terminat propriile calcule.

În Pthreads, o barieră este reprezentată prin tipul *pthread_barrier_t* și inițializată prin următoarea funcție:

```
int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t *attr, unsigned count);
```

Primul parametru reprezintă o referință la barieră, al doilea parametru poate fi folosit pentru setarea unor atribute ale barierei (la fel ca la mutex), iar ultimul parametru denotă numărul de thread-uri care trebuie să ajungă la barieră pentru ca aceasta să se deblocheze. Acest lucru înseamnă că bariera are un contor intern care numără thread-urile care așteaptă deblocarea ei. Atunci când contorul ajunge la numărul setat la inițializarea barierei, thread-urile își pot continua execuția paralelă.

Pentru a dezaloca o barieră, se folosește următoarea funcție:

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Ambele funcții returnează 0 dacă s-au executat cu succes sau un cod de eroare în caz contrar.

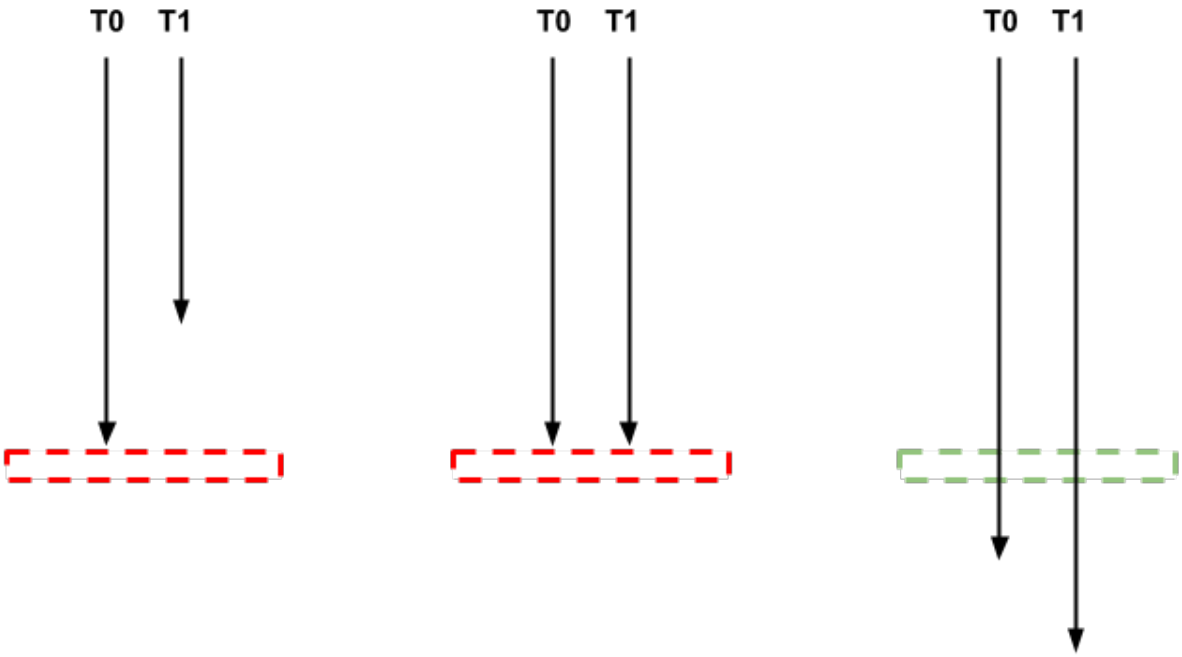
Pentru a face un thread să aștepte la o barieră (pentru a „pune o barieră” în cod), se folosește următoarea funcție:

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Funcția de mai sus va returna PTHREAD_BARRIER_SERIAL_THREAD pentru un singur thread arbitrar de la barieră și 0 pentru toate celelalte. Dacă funcția a avut vreo eroare, un cod eroare va fi returnat.

Atenție! Fiecare thread care trebuie să aștepte la barieră va apela funcția de mai sus pe aceeași variabilă de tip *pthread_barrier_t*. Dacă numărul de thread-uri care apelează **pthread_barrier_wait** este mai mic decât parametrul cu care a fost inițializată bariera, aceasta nu se va debloca niciodată.

O reprezentare grafică a modului de funcționare a unei bariere se poate vedea în figura de mai jos, unde avem o barieră inițializată cu 2. Atunci când thread-ul **T0** ajunge la barieră, acesta se blochează în așteptare, așa cum se observă în partea stângă a imaginii. La un moment dat de timp, **T1** va ajunge și el la barieră, așa cum se vede în centrul imaginii. Abia în acel moment, cele două thread-uri își vor putea continua execuția individual, cum se observă în partea dreapta a imaginii.



Mai multe informații detaliate despre API-ul Pthreads găsiți în tutorialul LLNL [https://computing.llnl.gov/tutorials/pthreads/] și în man [https://man7.org/linux/man-pages/man7/pthreads.7.html].

Exerciții

1. Compilați fișierul **mutex.c** din scheletul de laborator [https://github.com/APD-UPB/APD/tree/master/laboratoare/lab02] și rulați de mai multe ori programul obținut (puteți folosi scriptul `test_mutex.sh`). Veți observa că rezultatul nu este întotdeauna cel așteptat. Rezolvați problema de sincronizare folosind un mutex.
2. Pornind de la fișierul **barrier.c** din scheletul de laborator, folosiți o barieră pentru a vă asigura că output-ul va fi întotdeauna „1\n2”. Verificați **Hint 1** de mai jos pentru informații suplimentare.
3. Pornind de la fișierul **multiply_outer.c** din scheletul de laborator, paralelizați programul prin împărțirea iterației exterioare la mai multe thread-uri. Verificați corectitudinea și scalabilitatea programului rezultat. Verificați **Hint 2** de mai jos pentru informații suplimentare.
4. Pornind de la fișierul **multiply_middle.c** din scheletul de laborator, paralelizați doar cea de-a doua buclă de iterație. Verificați corectitudinea și scalabilitatea programului rezultat.
5. Pornind de la fișierul **multiply_inner.c** din scheletul de laborator, paralelizați doar bucla interioară. Verificați corectitudinea și scalabilitatea programului rezultat. Verificați **Hint 3** de mai jos pentru informații suplimentare.
6. Pornind de la fișierul **strassen.c** din scheletul de laborator, paralelizați înmulțirea unor matrice cu algoritmul Strassen folosind 7 thread-uri (de recomandat într-un fișier separat, numit **strassen_par.c**, pentru a putea testa corectitudinea folosind scriptul `test_strassen.sh`, care compară versiunea serială cu cea paralelizată). Verificați **Hint 4** de mai jos pentru informații suplimentare.

Hint 1

Pe sisteme MacOS, biblioteca de Pthreads nu conține implementarea pentru barieră. Pentru a putea face totuși acest exercițiu, există în scheletul de laborator un fișier numit *pthread_barrier_mac.h* pe care trebuie să-l includeți în fișierul

vostru sursă.

Hint 2

Pentru a putea testa corectitudinea paralelizării de la exercițiile 3, 4 și 5, găsiți în arhiva de laborator un script numit *test_multiply.sh*. Acesta face următorii pași:

1. verifică să existe un binar numit *multiply_seq* pentru implementarea secvențială a înmulțirii de matrice, pentru care aveți fișierul sursă *multiply_seq.c* în arhiva de laborator (acesta va servi drept etalon pentru corectitudinea implementării paralele)
2. verifică să existe binare pentru implemenările voastre paralele (*multiply_outer* pentru exercițiul 3, *multiply_middle* pentru exercițiul 4, *multiply_inner* pentru exercițiul 5)
3. rulează programul secvențial
4. rulează cele trei programe paralele
5. compară rezultatele rulărilor paralele cu rularea secvențială folosind *diff*; dacă nu există diferențe, scriptul nu afișează nimic; dacă implementarea paralelă este incorectă, se vor afișa diferențele dintre rularea implementării secvențiale și a celei paralele.

În mod implicit, scriptul rulează pe matrice de 1000×1000 de elemente, cu două thread-uri pentru implementarea paralelă. Dacă doriți să modificați aceste valori (și vă recomandăm să o faceți, pentru o testare cât mai completă), puteți modifica valorile variabilelor **N** și **P** din script.

Hint 3

Veți observa la acest exercițiu că, dacă paralelizați bucla interioară așa cum ați făcut la precedentele două exerciții, rezultatele nu vor fi întotdeauna corecte. De ce? Ce trebuie să faceți pentru o implementare corectă?

Hint 4

Algoritmul lui Strassen [http://stanford.edu/~rezab/classes/cme323/S16/notes/Lecture03/cme323_lec3.pdf] este un algoritm pentru înmulțire de matrice mai rapid decât metoda standard, având o complexitate de $O(N^{2.8074})$. În acest algoritm, la primul pas se definesc 7 matrice adiționale obținute prin înmulțiri ale matricelor bloc obținute din matricele inițiale. La al doilea pas, aceste 7 matrice noi sunt folosite pentru a calcula (prin operații de adunare și scădere) componentele bloc ale matricei finale.

Pentru exercițiul 6, aveți deja implementat calculul matricelor adiționale și calculul final, așa că voi trebuie doar să paralelizați aceste operații.

apd/laboratoare/02.txt · Last modified: 2021/10/22 15:55 by florin.mihalache