TEMĂ TEHNICI PROGRAMARE MULTIPROCESOR

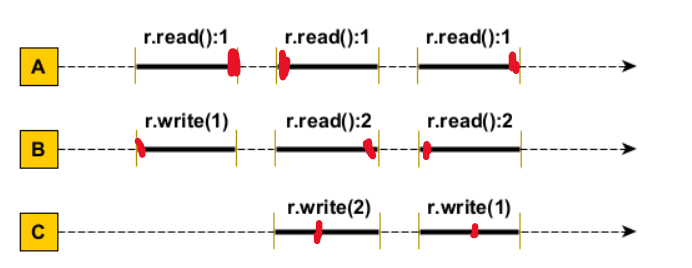
Lazurca Daniel-Vasile, Maxiniuc Diana-Alexandra

Grupa 3E2, seminar joi 14-16

Noiembrie 2024

EXERCIȚIUL 1

Secvența prezentată este linearizabilă deoarece există o ordine de execuție a celor 3 procese care să producă output-urile prezentate. Secvența fiind linearizabilă, este în consecință și consistent secvențială. Acea ordine este prezentată în poza de mai jos:



Această ordine poate reprezentată și astfel:

r.write(1)\_B -> r.read()\_A: 1 -> r.read()\_A: 1 -> r.write(2)\_C -> r.read()\_B:2 -> r.read()\_B: 2 -> r.write(1)\_C -> r.read()\_A:1

EXERCIȚIUL 2

Se preferă varianta în care instrucțiunea lock() se execută înaintea blocului de try în pofida celei în care lock() se execută în interiorul blocului try deoarece, în a doua variantă, instrucțiunea de unlock() se execută indiferent de succesul execuției lock(). Dacă lock-ul se face în interiorul blocului try și eșuează, va returna o excepție iar algoritmul va executa blocul finally în care se dă unlock, chiar dacă resursa nu a fost blocată.

De asemenea, execuția comenzii lock() înainte de blocul try face codul mai lizibil, delimitând concrect secțiunea critică de restul logicii de sincronizare a proceselor.

EXERCIȚIUL 3

Motivul pentru care algoritmul Bakery compară atât etichetele, cât și label-urile proceselor este pentru a evita fenomenele de Mutual Exclusion și Starvation.

Mai întâi, vom analiza cazul în care se compară doar indecșii. Să presupunem că avem două procese, A și B, A > B, ambele încercând să intre în secțiunea critică, apelând mai întâi funcția lock. Problema cu acest caz este una foarte clară. Cum A > B, asta inseamnă că threadul B va avea mereu prioritate față de threadul A, indiferent de ordinea în care acestea cer permisiunea de a intra în secțiunea critică, ceea ce poate duce la fenomenul de starvation pentru threadul A. Să presupunem următoarea succesiune de instrucțiun:

write\_A(flag[A]) -> write\_B(flag[B]) -> read\_A(flag[B]) (aici A vede că flag[B] este setat pe true, dar cum label[A] > label[B], așteaptă în continuare) -> read\_B(flag[A]) (aici B intră în secțiunea critică) -> unlock\_B -> write\_B(flag[B]) -> read\_A(flag[B])

Din câte putem vedea, este posibil ca A să rămână blocat în funcția de lock deoarece B are prioritate mereu față de A, având un label mai mic.

Acum, să analizăm cazul în care se compară doar label-urile. Vom presupune cele două procese A și B care încearcă simultan să intre în secțiunea critică.

Să presupunem că se face calculul max(label[0], label[1], …, label[n-1]) simultan în cele două procese iar rezultatul este același și pentru A și pentru B. În alte cuvinte, label[A] = label[B]. Astfel, în momentul în care cele două procese vor ajunge la verificarea condiției din bucla while, vor constata că aceasta returneaza false și vor intra în același timp în secțiunea critică, astfel nerespectând principiul de Mutual Exclusion.

În concluzie, algoritmul Bakery compară tupluri de forma (label, index) pentru a asigura corectitudinea între procese și desfășurarea corectă și completă a algoritmilor la care lucrează procesele.

EXERCIȚIUL 4

4.a)

NU. Răspunsul este negativ deoarece este posibil ca un *thread* sa aibă neșansa se aștepte indefinit pana să îi vină rândul a fi executat. Membrul *turn* este modificat de fiecare *thread* care vrea să intre în secțiune și este perfect posibil ca un *thread* sa rămână mereu în așteptare dacă de fiecare data după ce el execută *turn = me* (cel puțin) un alt thread să vina și să execute aceeași intructiune, blocându-i astfel primului thread șansa de a ieși din bucla *while* exterioară (care așteapta ca variabila *turn* să rămână nemodificată).

|  |  |  |
| --- | --- | --- |
| T1 | Tn | Obs: |
| me = ThreadId.get();  turn = me; |  | me = 1  turn = 1 |
|  | me = ThreadId.get();  turn = me; | me = 2  turn = 2 |
| used = true; |  |  |
|  | used = true; |  |
| while (turn != me); |  | turn != 1, deci T1 rămâne în bucla |
|  | while (turn != me); | turn == 2, deci T2 intra in sectiunea critica |
| do {  turn = me;  } while (used); |  | turn = 1  asteapta ca T2 sa faca unlock(). |
|  | used = false; | T2 face unlock(). |
|  | me = ThreadId.get();  turn = me; | me = 3  turn = 3  T3 e mai rapid decat T1. |
| used = true; |  |  |
|  | used = true; |  |
| while (turn != me); |  | turn != 1, deci T1 ramane in bucla |
|  | while (turn != me); | turn == 3, deci T3 intra in sectiunea critica |
| do {  turn = me;  } while (used); |  | turn = 1  asteapta ca T3 sa faca unlock(). |
|  | used = false; | T3 face unlock(). |
| Etc… |  |  |

4.b)

Nu, algoritmul prezentat **nu asigură excluderea mutuală**. Există scenarii de execuție în care două sau mai multe thread-uri pot intra simultan în secțiunea critică, încălcând astfel principiul de excludere mutuală.

**Problema cheie:**

Algoritmul se bazează pe presupunerea că, dacă un thread vede că *x == me*, atunci nu există alt thread care ar putea intra în secțiunea critică în același timp fără a apela *lock.lock()*. Însă această presupunere nu este întotdeauna validă din cauza concurenței și a modului în care variabilele x și y pot fi modificate de mai multe thread-uri aproape simultan.

Exemplu concret în care excluderea mutuală este încălcată:

|  |  |  |
| --- | --- | --- |
| T1 | T2 | Obs: |
| int me = ThreadId.get();  x = me;  while (y != 0) {}; |  | me = 1  x = 1  nu intra in bucla while deoarece y = 0 |
|  | int me = ThreadId.get();  x = me;  while (y != 0) {}; | me = 2  x = 2  nu intra in bucla while deoarece y = 0 |
| y = me; |  | y = 1 |
|  | y = me; | y = 2 |
| if (x != me) {  lock.lock();  } |  | x != me, deci T1 se va baza pe lock.lock() pentru asigurarea excluderii mutuale, dar lock este liber ceea ce **ii va asigura lui T1 acces la zona critica,** blocând totodată lock-ul. |
|  | if (x != me) | x == me, deci va intra in sectiunea critica fara sa verifice lock-ul. |
| T1 este in secțiunea critica. | T2 este în secțiunea critică, dar fără a deține lock. | Atât T1, cât și T2 sunt în secțiunea critică simultan, încălcând excluderea mutuală. |

4.c)

Demonstrație că cel mult un *thread* poate obține "red":

Pentru ca un *thread* să obțină "red", trebuie să indeplineasca următoarele condiții:

* Să ajungă la linia 8 cu *getWhite == false*, adică să nu fi fost setat de alt thread înainte.
* La linia 11, condiția *last == me* trebuie să fie adevărată.

Argumentare:

Primul *thread* care ajunge la linia 8 și găsește *getWhite == false* va continua și va seta *getWhite = true* la linia 10.

La linia 11, dacă niciun alt *thread* nu a modificat last între linia 7 și linia 11, atunci *last == me* va fi adevărat, iar *thread*-ul va returna "red".

Dacă alt *thread* modifică *last* între linia 7 și linia 11, atunci *last != me*, iar *thread*-ul va returna "black".

Doar un *thread* poate trece de la linia 7 la linia 11 fără ca *last* să fie modificat de alt *thread*.

Dacă mai multe *thread*-uri ajung la linia 10 simultan, doar unul dintre ele poate avea *last == me* la linia 11. Acest lucru se datorează faptului că *last* este suprascris de fiecare *thread* la linia 7, iar *interleaving*-urile pot face ca *last* să fie diferit de *me* pentru celelalte *thread*-uri.

Deci, cel mult un thread poate obține "red", deoarece doar unul poate îndeplini condiția *last == me* la linia 11 după ce a setat *getWhite = true*.

Demonstrație că cel mult un thread poate obține "black":

Pentru ca un thread să obțină "black", trebuie să indeplineasca următoarele condiții:

* Ajung la linia 8 cu *getWhite == false*, adică înainte ca *getWhite* să fie setat la true.
* La linia 11, găsesc că *last != me*, deci returnează "black".

Pentru a obtine cifrele maxime, presupunem ca toate cele n threaduri executa (aproape) simultan liniile 5-10.

Totalul thread-urilor care pot ajunge la linia 10 cu *getWhite == false* este n, deoarece toate *thread*-urile pot începe înainte ca *getWhite* să fie setat la *true*. Dintre acestea, unul obtine "red", conform demonstrației anterioare.

Restul de n-1 thread-uri, care nu au îndeplinit condiția *last == me*, vor returna "black" la linia 11.

Dacă totuși există threaduri mai „leneșe” care ajung la linia 8 după ce un alt thread a setat deja câmpul *getWhite*, numerele obținute din presupunerea anterioară vor fi mai mici deoarece pentru aceste threaduri se va returna "white".