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 in secțiune si este perfect posibil ca un *thread* sa ramana mereu in asteptare daca de fiecare data după ce el executa *turn = me* (cel putin) un alt thread sa vina si sa execute aceeasi intructiune, blocandu-i astfel primului thread șansa de a ieși din bucla *while* exterioara (care asteapta ca variabila *turn* sa ramana 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 ramane in 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… |  |  |

Exemplu concret:

* Dacă un thread rapid (să zicem T1) eliberează și recaptură lock-ul în mod repetat, poate seta used = true înainte ca alte thread-uri să aibă șansa să preia lock-ul.
* În timpul în care T1 este în secțiunea critică, alte thread-uri (T2, T3 etc.) stau în bucla interioară și setează turn la propriul ID.
* Când T1 eliberează lock-ul, există o competiție pentru a seta *used = true*. Datorită diferențelor de planificare și timpi de execuție, T1 poate prelua din nou lock-ul înaintea celorlalți.
* Astfel, thread-urile T2, T3 pot fi întârziate indefinit, neavând garanția că vor obține lock-ul într-un timp finit.

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.

Daca totusi exista threaduri mai „lenese” care ajung la linia 8 dupa ce un alt thread a setat deja campul *getWhite*, numerele obtinute din presupunerea anterioară vor fi mai mici deoarece pentru aceste threaduri se va returna "white".