Exclusion et section critiques

PCO

3 - Exlusion et Section critique

Résumé du document

Definition

Table des matières

1. Ressource critique

Une ressource critique est une ressource non partageable qui peut-être accédée par plusieurs tâches.

- Ressources logiques
  - Une variable globale (lecture et ecriture) accédée par des tâches
  - Une ressource physique (périphérique) accédée par des tâches

```
static int counter = 0;
const int NB_ITERATIONS = 1000000;

void run() {
  for(int i = 0; i < NB_ITERATIONS; i++) {
    counter = counter + 1;
  }
}

int main(int argc, char *argv[])
{
  std::vector<PcoThread*> threads;
  for(int i = 0; i < 2; i++)
    threads.push_back(new PcoThread(run));
  for(int i = 0; i < 2; i++)
    threads[i]->join();

  std::cout << "Fin des taches : counter = " << counter
    << " (" << 2 * NB_ITERATIONS << ")" << std::endl;

  return 0;
}
```

Dans cet exemple on voit que la ressource `int counter` va être accédée simulatanément par deux threads.

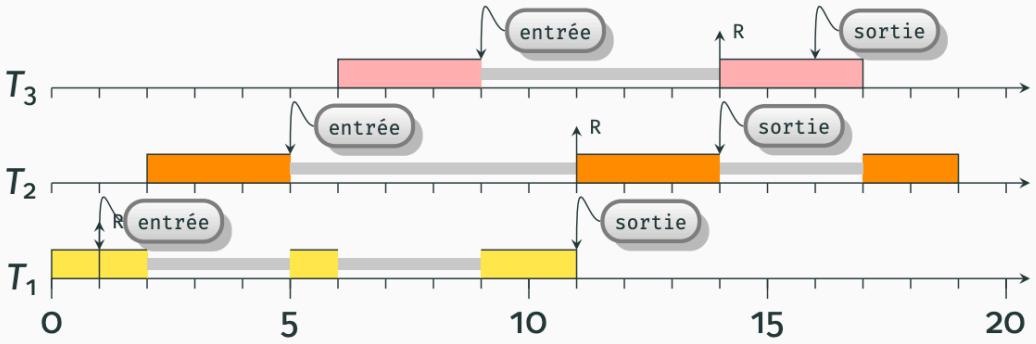
1.1. Section critique

- Une ressource critique doit être exécutée en exclusion mutuelle (c.-à-d. les accès à la ressource s'excluent mutuellement)
- La portion de code contenant un accès à une ressource critique est appelée **section critique**
- L'exclusion mutuelle doit être assurée dans une section critique
- L'accès à une section critique est géré par un algorithme d'exclusion mutuelle en deux parties:
  - protocole d'entrée
  - protocole de sortie

```
void T(void* arg) {
  déclarations locales;
  instructions;
  while (true) {
    instructions;
    <prélude>           // Protocole d'entrée
    <section critique>  // Accès à la RC
    <postlude>          // Protocole de sortie
    instructions;
  }
}
```

2. Propriétés des algorithme

Les tâches doivent pouvoir avancer (**liveness**) et éviter les interblocage (**deadlock**).

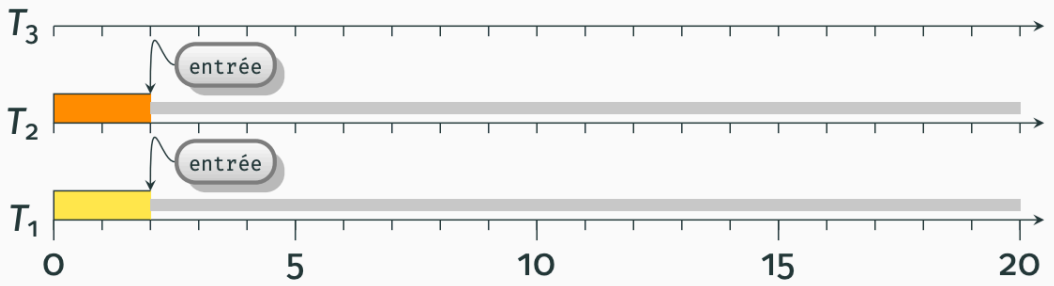


Légende

- Gris: tâche suspendue
- Couleur: tâche s'exécute (1 seul coeur)
- R: obtention de la ressource en accès exclusif

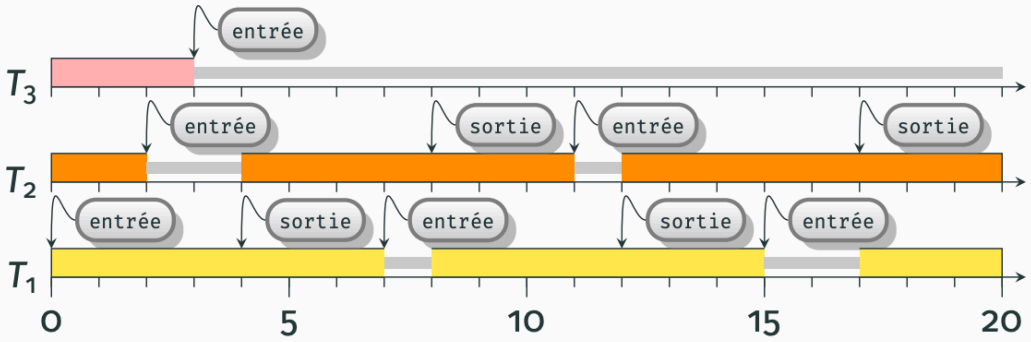
2.1. Interblockage

Dans le cas d'un code ne gérant pas correctement les accès aux sections critiques on peut se retrouver dans la situation d'un interblocage.



2.2. Famine

Dans le cas ou plusieurs opérations auraient besoin de la même ressources alors il se peut que certains thread ne puissent plus avancer. Pour cela il faut que l'ordonnonceur arbitre de manière équitable pour que toutes les ressources puissent avancer.



Légende

- Couleur: la tâche s'exécute (multiple coeurs)

2.3. Protocole d'entrée et attente

Dans la situation ou plusieurs tâches demandent accès à une ressource commune, les types d'attentes sont:

- FIFO (First In First Out)
- Linéaire: une tâche ne peut accéder deux fois à la ressource si une autre est en attente
- Bornée par une fonction $f(n)$: pour n tâches, une tâche en attente ne peut se faire dépasser par $f(n)$ tâches
- Finie: l'attente n'est pas bornée mais non plus infinie

2.4. Propriété des algorithmes

1. A tout instant, **une seule tâche** peut se trouver en **section critique**
2. Si plusieurs tâches sont bloquées en attente d'entrer en section critique alors qu'aucune tâche ne s'y trouve, l'une d'entre elles doit pouvoir y accéder au bout d'un temps fini (pas d'interblocage)
3. Le comportement d'une tâche en dehors de la section critique et des protocoles qui en gèrent l'accès n'a aucune influence sur l'algorithme d'exclusion mutuelle
4. Aucune tâche ne joue de rôle privilégié, la solution est la même pour toutes

3. Différents algorithmes

3.1. Peterson

```
bool intention[2] = {false,false};
int tour = 0; // ou 1

void T0()
{
  while (true) {
    intention[0] = true;
    tour = 1;
    while (intention[1] && tour == 1)
      ;
    /* section critique */
    intention[0] = false;
    /* section non-critique */
  }
}
```

```
void T1()
{
  while (true) {
    intention[1] = true;
    tour = 0;
    while (intention[0] && tour == 0)
      ;
    /* section critique */
    intention[1] = false;
    /* section non-critique */
  }
}
```

Mathématiquement les algorithmes de Peterson et Dekker sont corrects mais les processeurs multi-coeur les mettent à mal.

- Cohérence des mémoires caches
- Ordre des accès mémoires

Pour corriger cela il faut ajouter des barrières de syncheronisation, définit en C++ par

```
std::atomic_thread_fence(std::memory_order_acq_rel)
```

```
bool intention[2] = {false,false};
int tour = 0; // ou 1

void T0()
{
  while (true) {
    intention[0] = true;
    tour = 1;
    BARRIER;
    while (intention[1] && tour == 1)
      ;
    /* section critique */
    intention[0] = false;
    /* section non-critique */
  }
}
```

```
void T1()
{
  while (true) {
    intention[1] = true;
    tour = 0;
    BARRIER;
    while (intention[0] && tour == 0)
      ;
    /* section critique */
    intention[1] = false;
    /* section non-critique */
  }
}
```

3.2. Dekker

`std::atomic_thread_fence(std::memory_order_acq_rel)`

```
bool etat[2] = {false, false};
int tour = 0; // ou 1

void T0() {
    while (true) {
        etat[0] = true;
        BARRIER;

        while (etat[1]) {
            if (tour == 1) {
                etat[0] = false;
                while (tour == 1)
                  ;
                etat[0] = true;
                BARRIER;
            }
        }

        // Section critique
        tour = 1;
        etat[0] = false;
        // Section non-critique
    }
}
```

```
void T1() {
    while (true) {
        etat[1] = true;
        BARRIER;

        while (etat[0]) {
            if (tour == 0) {
                etat[1] = false;
                while (tour == 0)
                  ;
                etat[1] = true;
                BARRIER;
            }
        }

        // Section critique
        tour = 0;
        etat[1] = false;

        // Section non-critique
    }
}
```