CC - ASR7 Programmation concurrente

2 heures

29 mars 2018

Afin d'obtenir tous les points il vous est demandé de justifier vos résultats. Le barème proposé est susceptible d'être modifié lors de la correction. Il n'est présent que pour vous donner une idée du poids relatif des différentes questions.

Pour toutes les questions, « Implémentez la fonction ... » signifie donner le prototype (types des arguments et de la valeur de retour) et le corps de la fonction. Les détails syntaxiques (point-virgules, arguments exacts des templates, `#include`...) ne sont pas pris en compte dans l'évaluation, mais essayez d'utiliser une syntaxe aussi proche que possible du C++.

I Ordonnancement

Nous allons nous intéresser à un jeu de tâches périodiques (tâches A, B, C) et une tâche D non périodique. De plus, deux de ces tâches (A et C) partageront un mutex.

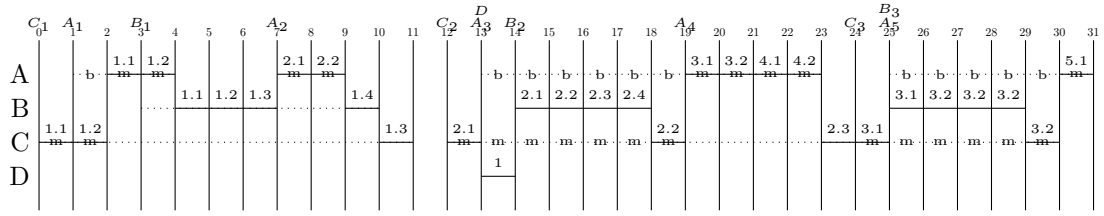
Vous devez utiliser un ordonnancement préemptif (la priorité des tâches est donnée dans le tableau suivant, sachant que plus la valeur de priorité est haute, plus la tâche est prioritaire).

Tâche	Période	Date(s) d'arrivee(s)	Priorité	Durée	Remarque
A	6	1, 7, 13, 19, 25, ...	10	2	Périodique, à chaque fois la tâche utilise le mutex du début à la fin
B	11	3, 14, 25, ...	8	4	Périodique
C	12	0, 12, 24	1	3	À chaque fois, la tâche a besoin du mutex au début de l'exécution (temps 1 et 2)
D	-	13	6	1	Ponctuelle

Q.I.1) - Faire l'ordonnancement de ces tâches sur 30 unités de temps. Vous pouvez écrire votre réponse sur la feuille de réponse ci-dessous, à rendre avec votre copie (n'oubliez pas d'y inscrire votre numéro d'anonymat).

Q.I.2) - Quel est le temps de réponse de chaque tâche sur l'intervalle demandé (c'est-à-dire le maximum des durées constatées pour chaque tâche) ?

Solution:



— A : 3 ; 2 ; 8 ; 4 ; 6 ⇒ 8
— B : 7 ; 4 ; 4 ⇒ 4
— C : 11 ; 12 ; (9) ⇒ 12
— D : 1

II Problème : le parisien

Vous avez entendu parler du problème du lecteur rédacteur. Le but est de protéger l'accès à une structure de données, par exemple une base. Les threads sont décomposés en 2 classes :

— Les écrivains (ou rédacteurs) qui modifient les données. Pour éviter les problèmes de cohérence, un écrivain doit être seul à éffectuer une modification et ne peut faire de modification pendant une lecture.

— Les lecteurs qui ne font que lire. Un lecteur ne peut pas lire pendant une modification mais plusieurs lecteurs peuvent lire en même temps.

C'est un problème très proche de celui du pont à une seule voie (Miralonde).

Ce problème est connu pour désavantager les écrivains. En effet, ces derniers risquent la famine dans le cas où il y a beaucoup de lecteurs qui arrivent régulièrement. Dans ce cas, comme les lecteurs empêchent les écrivains de rentrer mais pas leur semblables, si le nombre d'écrivains est important, il y aura toujours un lecteur en train d'utiliser la base. Il suffit pour cela qu'un lecteur arrive avant la sortie du dernier lecteur présent.

Pour trouver une solution, il est possible de changer légèrement le comportement des écrivains. Ces derniers acceptent d'attendre un certain temps, mais si ce temps est dépassé, ils changent de comportement et interdisent à de nouveaux lecteurs d'entrer. Bien sûr pour éviter la corruption des données, cet écrivain au comportement spécial devra toujours attendre que les lecteurs présents en section critique soient partis, mais puisque les nouveaux lecteurs seront bloqués avant d'entrer en section critique, les écrivains sont sûrs de pouvoir effectuer leur tâche.

Le nouvel algorithme des écrivains est donc :

Données : T : temps d'attente polie
si *Il y a des lecteurs ou des rédacteurs* **alors**
  J'attends, mais moins que T secondes
si *J'ai trop attendu* **alors**
  Je bloque les nouveaux lecteurs et j'attends la sortie des présents car j'ai du travail !
Je rentre en section critique.

Algorithme 1 : Écrivain parisien

Vous devez proposer un moniteur ParisienGenevois qui implémente cela. Le moniteur que vous proposez doit comporter 4 méthodes :

— DebutEcritureP, FinEcritureP qui sera appelée au début et à la fin de l'écriture.

— DebutLectureG, FinLectureG qui sera appelée au début et à la fin des lectures.

Il est bien évident que le comportement du lecteur doit se plier aux besoins des écrivains [1]. De plus, le moniteur que vous proposez doit veiller en priorité à ne pas enfreindre les règles de présence sur la structure de données :

— Au plus un thread doit être entre les deux méthodes DebutEcritureP et FinEcritureP et s'il y en a un, il ne peut y avoir de thread entre les méthodes DebutLectureG et FinLectureG.

— Plusieurs thread peuvent être entre les deux méthodes DebutLectureG et FinLectureG. Mais s'il y en a un (ou plus), il ne peut y avoir de thread entre les méthodes DebuteEcritureG et FinEcritureG.

Enfin, votre moniteur ne doit pas risquer de provoquer un *deadlock*. Ne cherchez pas la perfection, on considèrera comme acceptables les solutions pour lesquelles l'attente « polie » peut durer moins que prévu dans des cas patologiques.

Pour ce travail, il est nécessaire que vous utilisiez une nouvelle méthode des classes std::condition_variable et std::condition_variable_any :

```
std::cv_status wait_for(std::unique_lock<std::mutex>& lock,
                        const std::chrono::duration& att_max);
```

Cette fonction permet (comme wait) d'attendre une notification en libérant un mutex. Mais contrairement au wait simple, l'attente est bornée pas le temps : att_max. Si ce temps est dépassé, le thread est débloqué. Le retour de la fonction permet de faire la différence entre un déblocage normal (issu d'une notification) ou celui issu de l'expiration du délai. En effet, ce résultat est égal à std::cv_status::no_timeout dans le premier cas et std::cv_status::timeout dans le second. Par exemple :

1. Cette exercice est une œuvre de fiction. L'université ne prend pas la responsabilité des interprétations qui pourraient en être faites. De toute façon nous avons changé les noms !

```
using namespace std;
...
condition_variable_any cond;
mutex mut;
...
// attente de moins d'une seconde
cv_status cvs = cond.wait_for(mut, chrono::seconds(1))
if (cvs==cv_status::timeout) {
  // sortie à la fin de la période maximum
  ...
} else {
  // sortie anticipée
  ...
}
```

Q.II.1) - Écrire la structure de données du moniteur sans ses fonctions

Q.II.2) - Implémenter le constructeur

Q.II.3) - Implémenter les fonctions

Q.II.4) - Écrire le code de threads qui utilisent ces fonctions :

— `void parisien(int num, int nbtours)` qui effectue nbtours fois l'action suivante :

```
reflechie()
DebutEcritureP()
ecritDiatribe()
FinEcritureP()
```

— `void genevois(int num, int nbtours)` qui effectue nbtours fois l'action suivante :

```
mangeDuChocolat()
DebutLectureG()
litManuelDeSavoirVivre()
FinLectureG()
```

Q.II.5) - Écrire une fonction main qui lance N threads parisiens et M threads genevois et ne se termine que lorsque tous ont fini leur tâche.

Rappels sur C++11 et les threads

Pour vous aider, voici un rappel de la syntaxe C++11 pour les threads :

```
// Création et attente de terminaison d'un thread :
int main () {
        // ...
        std::thread t(f, 42, std::ref(x));
        // ...
        t.join();
}

// Déclaration d'un mutex
std::mutex m;

// Verrouillage/déverrouillage d'un mutex :
m.lock();
// ...
m.unlock();

// Instantiation d'un verrou :
{
        std::unique_lock<std::mutex> l(m);
        // ...
}
```

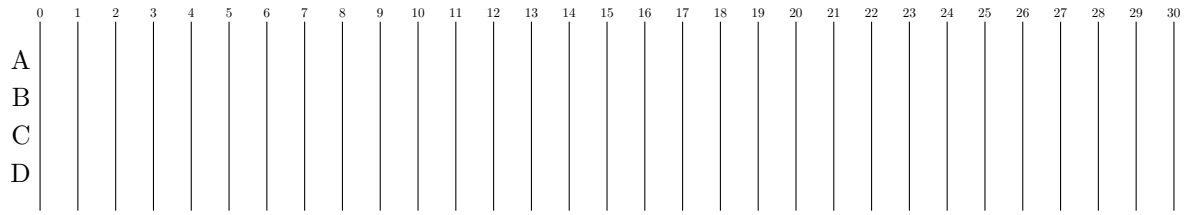
```
// Opérations sur une variable de condition :
std::condition_variable c;
c.wait(l); // l de type verrou (std::unique_lock par exemple)
c.notify_one();
c.notify_all();

// Opérations sur une variable de condition :
std::condition_variable_any c;
c.wait(m); // m de type mutex
c.notify_one();
c.notify_all();
```

III Feuille de réponse

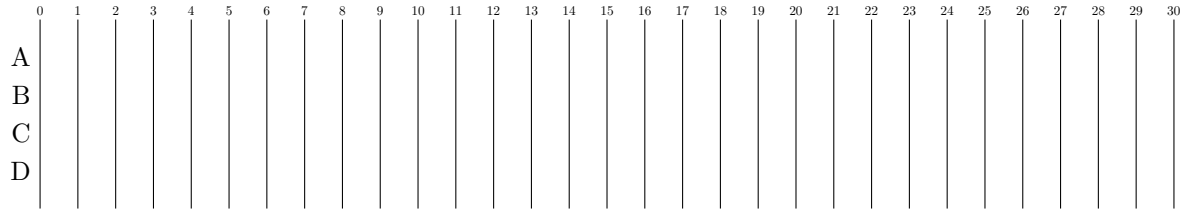
Numéro d'anonymat : _ _ _ _ _ _

Un seul des tableaux doit être rempli, les autres peuvent servir de brouillons ou de secours en cas d'erreur. **Un seul sera corrigé**, barrez distinctement les brouillons.

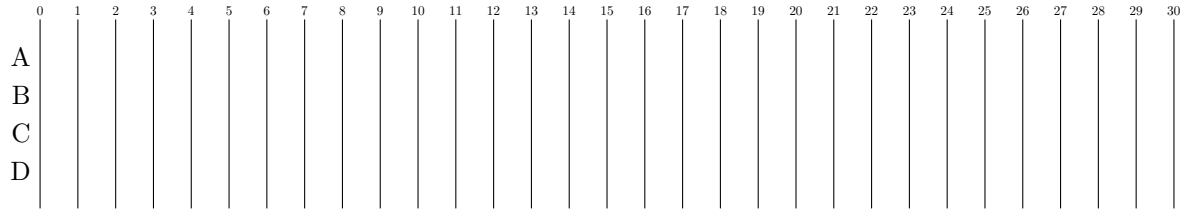
Brouillon :



Brouillon :



Question I.2 :



Temps de réponse :

A :	B :	C :	D :