+1/1/60+

Programmation Concurrente

Contrôle Continu Intermédiaire

Durée totale : 1h30

Toute communication (orale, téléphonique, par messagerie, etc.) avec les autres étudiants est interdite. Aucun document autorisé.

Vous rendrez le sujet complet agraphé. Vous reporterez votre **NUMÉRO D'ÉTUDIANT** sur la première page (ci-dessous).

— Pour la partie QCM, plusieurs réponses peuvent être valides à chaque question, on souhaite avoir **toutes** les réponses valides. Chaque question admet au moins une réponse valide et au moins une réponse incorrecte. Les réponses incorrectes peuvent entraîner des points négatifs (il n'y a pas de point négatif pour une question si vous n'avez coché aucune case). Le barème est indicatif.

— Pour les parties rédigées, vous répondrez obligatoirement dans les parties prévues pour, et seulement en cas d'extrême nécessité sur les blancs en base de pages (dernière page par exemple).

Consignes :

- Utilisez un **stylo à bille noir ou bleu foncé**.
- **Noircir ou bleuir** la/les cases, sans dépasser sur les autres cases !
- Pour corriger (dernier recours) : effacez proprement la case.
- Ne pas oublier de noter votre **numéro d'étudiant**.

Numéro d'étudiant :

- Numéro d'étudiant :

  .....
- Encodez-le ci-contre.

☐0	☐0	☐0	☐0	☐0	☐0	☐0	☐0
☐1	☐1	☐1	☐1	☐1	☐1	☐1	☐1
☐2	☐2	☐2	☐2	☐2	☐2	☐2	☐2
☐3	☐3	☐3	☐3	☐3	☐3	☐3	☐3
☐4	☐4	☐4	☐4	☐4	☐4	☐4	☐4
☐5	☐5	☐5	☐5	☐5	☐5	☐5	☐5
☐6	☐6	☐6	☐6	☐6	☐6	☐6	☐6
☐7	☐7	☐7	☐7	☐7	☐7	☐7	☐7
☐8	☐8	☐8	☐8	☐8	☐8	☐8	☐8
☐9	☐9	☐9	☐9	☐9	☐9	☐9	☐9

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();
```

1 Questions de cours

Question 1 ♣ **(0.5 point)** Une section critique est :

- ☐ Une situation où deux tâches ne partageant pas de mutex et où la tâche la moins prioritaire s'exécute alors que la tâche la plus prioritaire l'attend
- ☐ Une section de code exécutée par au maximum un seul thread
- ☐ Une section de code protégée par un mutex, ou autre mécanisme équivalent
- ☐ Un compteur toujours positif

Question 2 ♣ **(0.5 point)** Un sémaphore est :

- ☐ Un compteur toujours positif
- ☐ Un objet contenant des données, des fonctions/méthodes, un mutex et éventuellement des variables de conditions
- ☐ Une situation où deux tâches ne partageant pas de mutex et où la tâche la moins prioritaire s'exécute alors que la tâche la plus prioritaire l'attend
- ☐ Une section de code exécutée par au maximum un seul thread

Question 3 **(1 point)** Qu'est-ce qu'un moniteur de Hoare ?

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*

.....

.....

.....

.....

.....

.....

Question 4 **(1 point)** Quelle est la différence entre un *thread* (fil d'exécution) et un processus (au sens « processus Unix » par exemple) ?

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*

.....

.....

.....

.....

.....

.....

2 Ordonnancement

Question 5 ♣ (2 points) On considère trois processus sous Linux : `A` est en mode `SCHED_FIFO`, `B` et `C` en mode `SCHED_RR`, toutes les tâches ont une priorité 5, toutes arrivent à l'instant 0 (donc l'ordonnanceur a le choix de l'ordre d'exécution), ont une durée de 5 unités de temps. Le quantum est de 3 unités de temps. Cochez les débuts d'exécutions possibles :

- ☐ A s'exécute pendant 3 unités de temps, puis C en entier, puis B pendant 3 unités de temps, ...
- ☐ B s'exécute pendant 3 unités de temps, puis C pendant 3 unités de temps, puis A en entier, ...
- ☐ B s'exécute pendant 3 unités de temps, puis A en entier, puis C pendant 3 unités de temps, ...
- ☐ B s'exécute en entier, puis C en entier, puis A en entier, ...
- ☐ A s'exécute pendant 3 unités de temps, puis B en entier, puis C pendant 3 unités de temps, ...
- ☐ A s'exécute en entier, puis B pendant 3 unités de temps, puis C pendant 3 unités de temps, ...
- ☐ A s'exécute pendant 3 unités de temps, puis B pendant 3 unités de temps, puis C pendant 3 unités de temps, ...

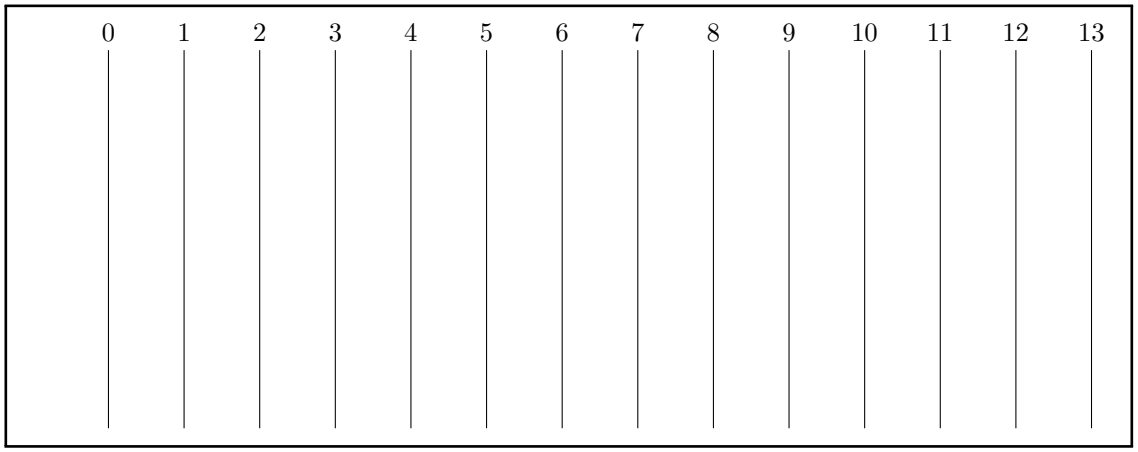
Question 6 (1 point) Donner un exemple d'ensemble de tâches où un ordonnancement « Shortest Job First » (plus court d'abord) préemptif donne un temps de réponse moyen plus court que l'ordonnancement « FIFO » sans priorité. Donnez une solution la plus simple possible (moins de 4 tâches, vous n'aurez pas tous les points si vous utilisez plus de tâches que nécessaire).

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*

Tâche	Date d'arrivée	Durée

Question 7 **(1 point)** Donnez l'ordonnancement SJF préemptif du jeu de tâches que vous avez proposé ci-dessus.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*



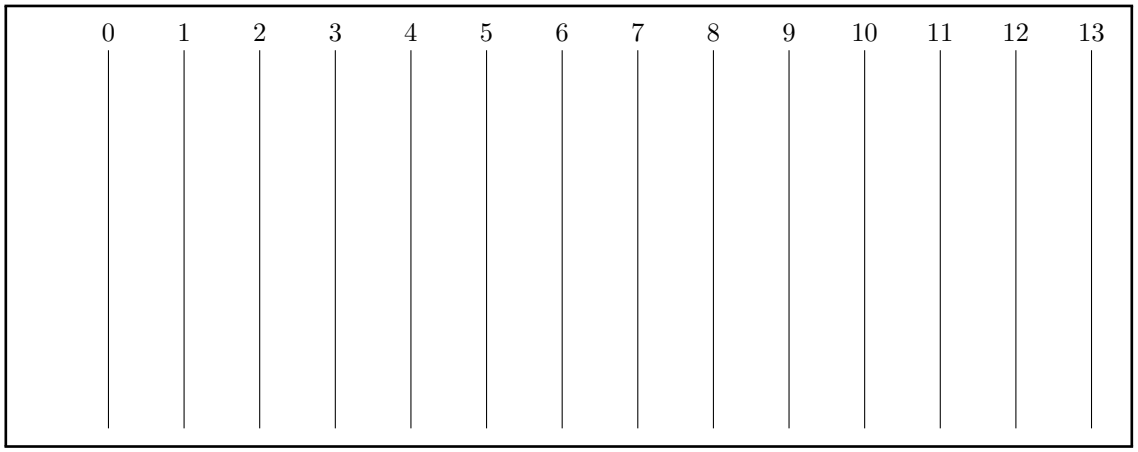
Question 8 **(0.5 point)** Quel est le temps de réponse moyen sur cet ordonnancement (donnez la formule de calcul et le résultat) ?

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*

.....

Question 9 **(1 point)** Donnez l'ordonnancement FIFO sans priorité du jeu de tâches que vous avez proposé ci-dessus.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*



Question 10 **(0.5 point)** Quel est le temps de réponse moyen sur cet ordonnancement (donnez la formule de calcul et le résultat) ?

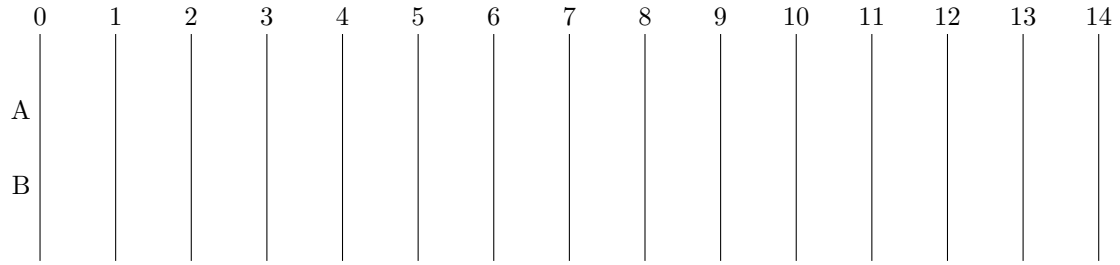
☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*

.....

Question 11 **(2 points)** Nous utilisons l'ordonnanceur préemptif de Linux avec priorité (Plus la valeur de priorité est importante, plus la tâche est prioritaire) qui se tient à chaque unité de temps sur un système monoprocesseur. Les tâches partagent un mutex M.

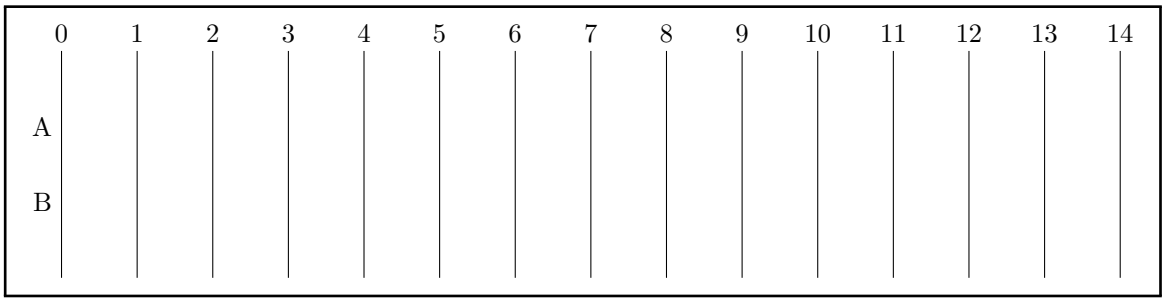
Tâche	Date d'arrivée	Politique	Priorité	Durée	Remarque
A	0	SCHED_RR	5	10	Verrouille le mutex M après 3 unités de temps, et le déverrouille 5 unités de temps plus tard.
B	4	SCHED_RR	5	2	Verrouille le mutex M pendant toute son exécution.

Faire l'ordonnancement de ces tâches. Vous pouvez utiliser le brouillon si besoin. Barrez la réponse incorrecte si vous répondez plusieurs fois.
Brouillon :



Réponse finale :

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*



Question 12 **(1.5 points)** Qu'est-ce qu'une inversion de priorité ? Y en a-t-il une ici ? Si oui à quel moment ? ☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*

.....

.....

.....

.....

.....

.....

Question 13 Quel est le temps de réponse (ou latence) de chaque tâche sur l'intervalle demandé ?

Temps de réponse de A : (0.5 point) ..... ☐Faux ☐OK *Réservé*

Temps de réponse de B : (0.5 point) ..... ☐Faux ☐OK *Réservé*

3 Communication d'un thread à l'autre

Dans le cours, nous avons vu le principe du producteur-consommateur qui permet à un thread d'envoyer des données à un autre thread.

Le but de cet exercice est d'expérimenter avec des variantes autour de ce principe. On considère deux threads, `PROD` (producteur) et `CONS` (consommateur), et on souhaite un mécanisme pour que `PROD` envoie des données à `CONS`.

3.1 Envoyer une valeur : la promesse

Un mécanisme très utilisé dans les langages modernes est celui de la promesse. Une promesse est un objet exposant deux méthodes : `set` (parfois appelée *resolve*) qui permet à `PROD` d'envoyer la valeur, et `get` qui permet à `CONS` de récupérer la valeur. `set` stocke la donnée dans l'objet, et `get` renvoie cette donnée si elle est disponible. Dans le cas où `get` est appelée avant `set`, la méthode `get` attend que `set` soit appelée avant de renvoyer la valeur. Ainsi, on est certain que `get` ne renvoie jamais une valeur non-initialisée.

Pour simplifier, nous considérons ici que la valeur à transmettre est toujours de type `float`.

Une utilisation possible est la suivante :

```
void producer(Promise & p) {
        float pi = compute_pi();
        p.set(pi);
}

void consumer(Promise & p) {
        std::cout << "The␣producer␣sent␣me␣" << p.get() << std::endl;
}
```

Contrairement au producteur-consommateur vu en cours, on ne peut faire qu'un appel à `set`. On peut faire plusieurs appels à `get`, dans ce cas tous les appels renvoient la même valeur. On n'a donc jamais besoin de stocker plusieurs valeurs différentes.

Question 14 **(1 point)** Écrire une fonction `main` qui instancie deux threads exécutant respectivement `producer` et `consumer`, et termine proprement. ☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*

Donnez une implémentation de la classe `Promise`, en utilisant le principe du moniteur de Hoare. Donnez chaque partie dans le cadre correspondant ci-dessous. Pour toutes les méthodes, donnez la signature (type de retour et arguments) et le corps de la méthode.

Question 15 **(1 point)** Champs de la classe :

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*

Question 16 **(1 point)** Méthode `set` : ☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

Question 17 **(1 point)** Méthode `get` : ☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

Question 18 **(2 points)** Proposez une autre implémentation en utilisant un sémaphore, sans utiliser de mutex. Nous n'avons pas vu la syntaxe C++ pour les sémaphores, vous pouvez utiliser du pseudo-code (`P`, `V`, ...)

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

3.2 Communication de tableau

Dans le cas où le thread `PROD` souhaite envoyer un tableau au thread `CONS`, on peut utiliser le même principe, en stockant le tableau dans une promesse. Mais avec une telle solution, le thread `PROD` ne peut commencer à utiliser les éléments du tableau qu'une fois que l'ensemble du tableau est passé à `set`. On souhaite une solution plus flexible où les éléments sont produits un par un (dans l'ordre, en commençant par la première case du tableau), et peuvent être consommés dès qu'ils sont prêts.

Nous allons donc écrire une classe `PromisePlus` qui permet d'échanger des tableaux d'entiers suivant ce principe. Une utilisation possible est la suivante :

```
void producer(PromisePlus & p) {
        int fact_n = 1;
        for (int i = 0; i < SIZE; ++i, fact_n = fact_n * i) {
                p.set(i, fact_n);
        }
}

void consumer(PromisePlus & p) {
        for (int i = 0; i < SIZE; ++i) {
                std::cout << "fact(" << i << ") = "
                                << p.get(i)
                                << std::endl;
        }
        int sum = 0;
        for (int i = 0; i < SIZE; ++i) {
                sum += p.get(i);
```

```
        }
        std::cout << "La somme est : " << sum << std::endl;
}
```

Donnez une implémentation de la classe `PromisePlus`, en utilisant le principe du moniteur de Hoare. Donnez chaque partie dans le cadre correspondant ci-dessous.

Question 19 **(1 point)** Champs de la classe :

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*

.....

.....

.....

.....

.....

Question 20 **(1 point)** Méthode `set` :

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*

.....

.....

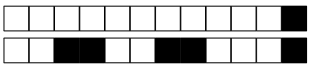
.....

.....

.....

.....

.....



Question 21 **(1 point)** Méthode `get` : ☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*

.....

.....

.....

.....

.....

.....

.....