Corrected

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 la dernière page (blanche).

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.

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)** Un mutex est :

- [x] Un mécanisme permettant de mettre en place une section critique.
- [ ] Une section de code exécutée par au maximum un seul thread
- [ ] Un compteur toujours positif
- [x] Un mécanisme permettant d'assurer qu'une section de code n'est exécutée qu'au plus par un thread en même temps.

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

- [x] Un compteur toujours positif
- [ ] Une section de code exécutée par au maximum un seul thread
- [ ] Un objet contenant des données, des fonctions/méthodes, un mutex et éventuellement des variables de conditions

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

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

Sous-total : 2

2 Ordonnancement

Question 4 ♣ (1 point) L'ordonnanceur d'un système d'exploitation s'occupe de :

- [x] décider si les processus peuvent être préemptés et si oui quand le faire
- [x] décider dans quel ordre exécuter les fils d'exécution (threads)
- [ ] placer les données en mémoire
- [ ] gérer les périphériques
- [x] décider dans quel ordre exécuter les processus

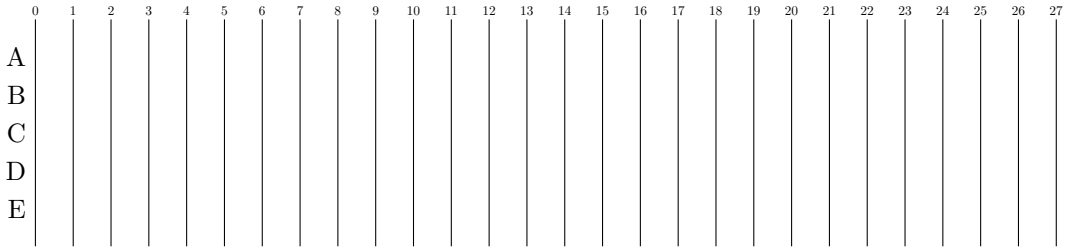
Question 5 ♣ (1 point) On considère trois processus sous Linux : `A` est en mode `SCHED_OTHER`, `B` en mode `SCHED_FIFO` avec priorité 10, et `C` en mode `SCHED_RR` en priorité 8. Cochez l'exécution correcte :

- [ ] B s'exécute pendant un quantum, puis rend la main à C, qui s'exécute pendant un quantum, ..., et enfin A s'exécute en dernier
- [ ] C s'exécute, puis B, puis A
- [x] B s'exécute, puis C, puis A
- [ ] A s'exécute, puis B, puis C
- [ ] C s'exécute pendant un quantum, puis rend la main à B, qui rend la main à C, et enfin A s'exécute en dernier

Nous utilisons un ordonnancement préemptif 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. Nous allons utiliser un jeu de tâches qui mélange des tâches périodiques et des tâches ponctuelles. De plus, les tâches peuvent être interrompues (en attente d'une lecture disque par exemple). Le mutex `M` est partagé entre les tâches.

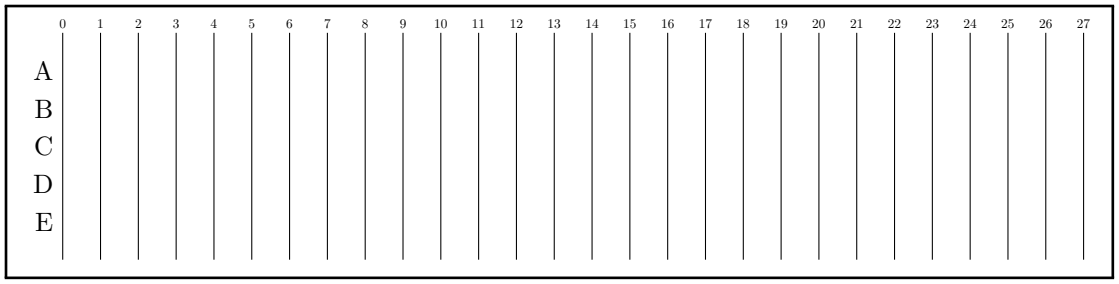
Tâche	Date(s) d'arrivée(s)	Priorité	Durée	Remarque
A	0, 6, 12	8	2	
B	0, 12	10	2	À chaque fois, après 1 unité de temps la tâche est interrompue pendant 2 unités de temps pour une lecture disque pendant laquelle le processeur est libéré, puis le processeur calcule encore pendant 1 unité de temps. À chaque exécution, il y a donc 2 unités de temps de calcul sur le processeur plus 2 unités de temps d'entrée/sortie.
C	9	6	2	Commence par prendre le mutex `M`, doit s'exécuter pendant presque 2 unités de temps et relâche le mutex `M`.
D	7	5	3	Après un peu moins d'1 unité de temps de calcul, elle prend un mutex `M`, doit s'exécuter pendant encore presque 1 unité de temps, relâche le mutex `M`, et calcule encore 1 unité de temps.
E	15	0	3	À chaque fois, après un peu moins d'1 unité de temps de calcul, elle prend un mutex `M`, doit s'exécuter pendant encore presque 1 unité de temps, relâche le mutex `M`, et calcule encore 1 unité de temps.

Question 6 **(2.5 points)** 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*




```
A  -  1  2              1  2                    -  1  2
B  1  -  -  2                                   1  -  -  2
C                                        -  m  m
D                          -  1  m  -  -  -  -  -  3
E                                                -  -  1  2  3
   ↑  .  .  .  .  ↑  .  .  .  .  ↑  .  .  .  .  ↑  .  .  .  .  ↑  .  .  .  .  ↑
   0              5              10             15             20             25
```

Question 7 **(2 points)** Qu'est-ce qu'une inversion de priorité ? Y en a-t-il une ici ? Si oui à quel moment ?

Cf. le cours. Non, il n'y en a pas ici.

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Question 8 Quel est le pire temps de réponse (ou latence) de chaque tâche sur l'intervalle demandé ?

Temps de réponse de A : (0.3 point) 3 ..... ☐Faux ■OK *Réservé*

Temps de réponse de B : (0.3 point) 4 ..... ☐Faux ■OK *Réservé*

Temps de réponse de C : (0.3 point) 3 ..... ☐Faux ■OK *Réservé*

Temps de réponse de D : (0.3 point) 10 ..... ☐Faux ■OK *Réservé*

Temps de réponse de E : (0.3 point) 5 ..... ☐Faux ■OK *Réservé*

Sous-total : 8

3 Gestion de la concurrence sur une entrée de parking

On considère un parking souterrain, comportant un nombre limité, `NB_PLACES` de places. Le parking est accessible via deux entrées, l'une à l'est et l'autre à l'ouest (on utilisera un type énuméré `enum cote EST, OUEST;`). À chaque entrée se trouve une barrière. On utilise des conventions similaires à celles utilisées pour le pont de Miralonde vu en TD :

— On modélise une voiture par un thread (il y a donc autant de threads que de voitures).

— Une voiture qui arrive à l'entrée `c` (avec `c == EST` ou `c == OUEST`) appelle la fonction `demande_entree(c)`. La voiture est autorisée à passer quand la fonction termine, et elle est bloquée tant que la fonction s'exécute (y compris si elle attend sur une variable de condition).

— Une voiture qui sort du parking appelle automatiquement la fonction `sortie()`. Il n'y a qu'une sortie au parking.

On suppose les constantes `NB_VOITURES`, `NB_PLACES`, `NB_ESSAIS` définies (entiers strictement positifs).

3.1 Empêcher les voitures d'entrer quand le parking est plein

Dans un premier temps, nous allons écrire une version permettant de garantir qu'une voiture est bloquée en attente au niveau de la barrière d'entrée tant que le parking est plein. Lorsqu'une voiture sort du parking, une des barrière (est ou ouest) choisie arbitrairement laisse entrer une voiture s'il y en a une en attente.

Question 9 **(1 point)** Écrire une classe (ou une structure si vous n'êtes pas assez à l'aise avec C++) `Parking` qui agira comme un moniteur de Hoare, et déclarer ses champs (méthodes et constructeur viendront dans les questions suivantes). Vous aurez au minimum besoin d'instancier un ou plusieurs objets de la bibliothèque de threads C++11, et d'un entier pour compter le nombre de places libres.

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

Question 10 **(0.5 point)** Écrire un constructeur `Parking(unsigned nb_places)` qui initialise le système avec `nb_places` places disponibles.

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

Question 11 **(1 point)** Écrire la méthode `void demande_entree(cote c)` de `Parking` (ou une fonction si vous préférez). Si le parking est plein, alors la fonction attend qu'une voiture sorte.

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

Question 12 **(1 point)** Écrire la méthode `void sortie()`, appelée quand une voiture sort. Cette méthode peut débloquer une voiture à une des entrées.

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

Question 13 **(1 point)** Écrire une fonction `voiture(int n)` qui sera exécutée par les threads représentant les voitures. Cette fonction doit exécuter `NB_ESSAIS` fois de suite les opérations `void demande_entree(cote c)` puis `sortie()`. Le côté `c` sera choisi de telle sorte que la moitié des voitures entrent par l'est, et l'autre moitié par l'ouest.

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

.....

Question 14 **(1.5 points)** Écrire une fonction `main()` qui instancie `NB_VOITURES` voitures et un parking `Parking`. Faites en sorte que le programme termine proprement quand toutes les voitures ont terminé leur travail.

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

Question 15 **(1.5 points)** Qu'est-ce qu'un interblocage (*deadlock*) ? Est-ce possible d'en avoir un sur ce système ? Si oui, donnez un exemple. Si non, montrez-le.

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

Question 16 (1.5 points) Qu'est-ce qu'une famine (*starvation*) ? Est-ce possible d'en avoir une sur ce système ? Si oui, donnez un exemple. Si non, montrez-le.

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

Sous-total : 9

3.2 Équité entre les entrées

Le code ci-dessus choisit arbitrairement quelle voiture laisser entrer lorsqu'une voiture sort et qu'il y avait des voitures en attente. Pour simplifier, on suppose que pour chaque entrée du parking, les voitures en attente sont débloquées dans leur ordre d'arrivée (ce qui n'est pas modélisé par notre code), mais nous allons faire en sorte que le système choisisse l'entrée sur laquelle une voiture doit être débloquée de manière équitable. Par exemple, si un voiture arrive et attend d'abord à l'est, puis une autre à l'ouest, et qu'une voiture sort, on souhaite que le système débloque forcément l'entrée à l'est, vu que c'est l'entrée où une voiture est arrivée la première. Une solution pour cela est d'utiliser une FIFO de côtés (`std::queue<cote>`) pour mémoriser dans l'ordre les côtés depuis lesquels les demandes d'entrée sont faites.

Pour rappel, voici un exemple d'utilisation de `std::queue` :

```
std::queue<int> f;
f.push(42); f.push(12);
cout << f.front(); // 42
cout << f.front(); // toujours 42
f.pop(); // Retire la première valeur
cout << f.front(); // 12
```

Proposez une nouvelle classe `ParkingFIFO` garantissant que les voitures sont acceptées dans l'ordre des requêtes (nouveaux champs, puis changements des fonctions dans les questions suivantes) :

Question 17 **(0.5 point)** Les nouveaux champs de la classe (ou structure) :

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

Question 18 **(1.5 points)** La fonction `demande_entree(cote c)` :

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Question 19 **(1.5 points)** La fonction `sortie()` :

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

Sous-total : 3.5
Total : 22.5