Corrected

Programmation Concurrente

Contrôle Continu Intermédiaire

Durée totale : 1h30

Toute communication (orale, téléphonique, par messagerie, etc.) avec les autres étudiants est interdite. 1 feuille A4 recto-verso manuscrite autorisée.

Vous rendrez le sujet complet agrafé. 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 bas 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)** Un sémaphore 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
- ■ Un compteur toujours positif ou nul
- ☐ Un objet contenant des données, des fonctions/méthodes, un mutex et éventuellement des variables de conditions

Question 2 ♣ **(0.5 point)** Cochez les réponses qui sont des traductions ou synonymes de thread :

- ■ Lightweight process
- ☐ Processus lourd
- ■ Processus léger
- ■ Fil d'exécution

Question 3 ♣ **(0.5 point)** Ce programme :

```
int v = 0;
mutex m1, m2;

void incr(int &v) {
  for (int i = 0; i < N; ++i) {
        m1.lock();
        m2.lock();
        ++v;
        m2.unlock();
        m1.unlock();
  }
}

void decr(int &v) {
  for (int i = 0; i < N; ++i) {
        m1.lock();
        m2.lock();
        --v;
        m2.unlock();
        m1.unlock();
  }
}

int main() {
  thread t1(incr, ref(v));
  thread t2(decr, ref(v));
  t1.join();
  t2.join();
}
```

- ☐ Peut produire un interblocage (deadlock) à l'exécution
- ■ Termine correctement
- ■ Utilise deux mutex inutilement, un seul suffirait
- ☐ Fait une boucle infinie

Question 4 ♣ **(0.5 point)** Cochez les réponses en rapport avec l'ordonnancement collaboratif :

- ■ Les tâches rendent la main lorsqu'elles sont finies ou lors de certains appels systèmes particuliers
- ■ Les tâches ne sont pas interruptibles
- ☐ Le système peut interrompre une tâche à tout moment
- ☐ Permet d'éviter le problème de famine

Question 5 (1 point) Qu'est-ce qu'un moniteur de Hoare ?

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Sous-total : 3

2 Ordonnancement

Question 6 (1 point) Présentez brièvement l'ordonnancement SJF (shortest job first), et donnez au moins un avantage et un inconvénient

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

bon temps de reponse moyen mais il faut connaitre la duree des taches. Risque de famine

Question 7 **(1 point)** Présentez brièvement l'ordonnancement FIFO, et donnez au moins un avantage et un inconvénient

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

peu couteux mais mauvais temps de reponse moyen. Pas de risque de famine

Question 8 **(0.5 point)** L'ordonnancement FIFO (sans priorité) peut-il être preemptif ? Argumentez brièvement

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

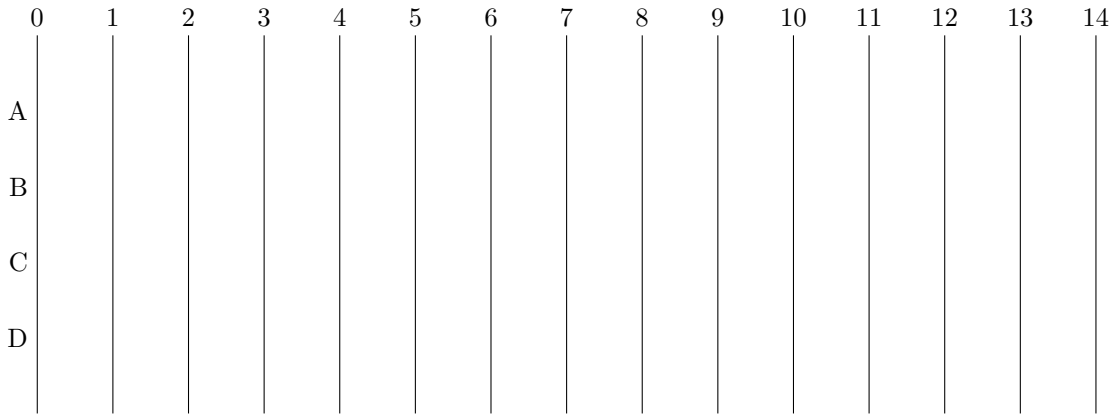
non, le thread elu serait celui qui s'executait deja

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. Le quantum de temps est de 2 unités de temps. Les tâches partagent un mutex M.

Tâche	Date d'arrivée	Politique	Priorité	Durée	Remarque
A	0	SCHED_RR	5	3	Verrouille le mutex M après 1 unité de temps, et le déverrouille 2 unités de temps plus tard (avant de terminer).
B	2	SCHED_RR	10	2	Verrouille le mutex M pendant toute son exécution.
C	1	SCHED_RR	5	5	
D	7	SCHED_RR	10	5	Après 1 unité de temps, la tâche est interrompue pendant 2 unités de temps pour une lecture disque, puis la tâche peut reprendre (une fois la lecture terminée)

Faire l'ordonnancement de ces tâches. Vous pouvez utiliser le brouillon si besoin. Barrez la réponse incorrecte si vous répondez plusieurs fois. Si vous avez vu une autre présentation en TD (sur une seule ligne à la place d'une ligne par tâche), vous pouvez représenter votre ordonnancement de cette manière.

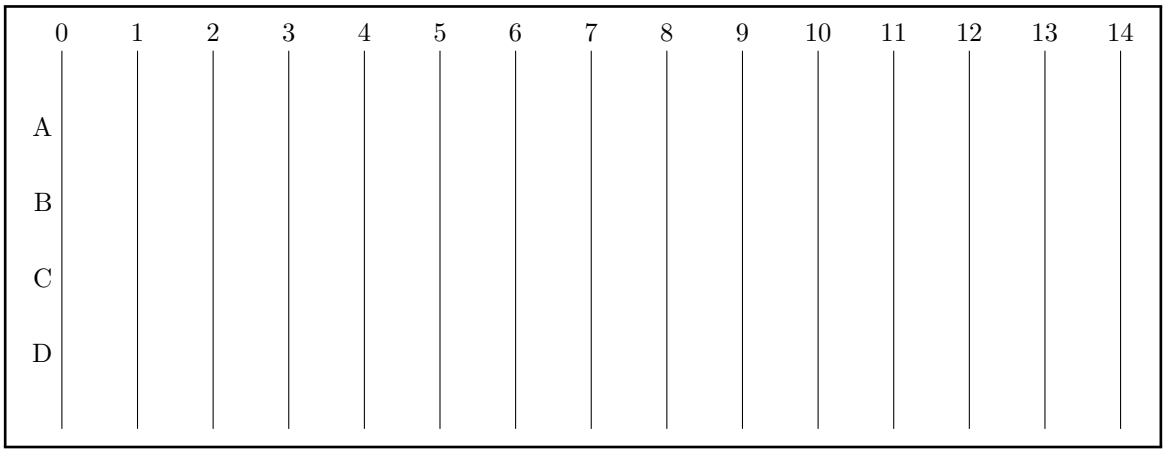
Brouillon :

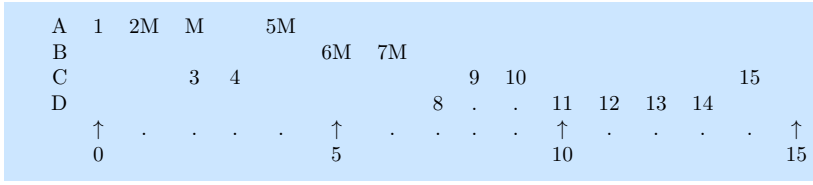


Réponse finale :

Question 9 (3 points)

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*





Question 10 Quel est le temps de réponse (ou latence) de chaque tâche sur l'intervalle demandé ?

Temps de réponse de A : (0.5 point) 5 ..... ☐Faux ■OK *Réservé*

Question 11

Temps de réponse de B : (0.5 point) 5 ..... ☐Faux ■OK *Réservé*

Question 12

Temps de réponse de C : (0.5 point) 14 ..... ☐Faux ■OK *Réservé*

Question 13

Temps de réponse de D : (0.5 point) 7 ..... ☐Faux ■OK *Réservé*

Question 14 **(1 point)** Qu'est-ce qu'une inversion de priorité ? Y en a-t-il une ici ? Si oui à quel moment ?

Oui, C passe devant B car A détient le mutex

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

.....

Sous-total : 8.5

3 Gestion de la concurrence pour acheter des livres dans une librairie

Afin de se cultiver et de passer le temps, des lecteurs veulent acheter chacun un livre. La librairie se charger de commander les livres (ce qui requiert un temps indéfini plus ou moins long) et chacun des lecteurs achète un livre quand c'est possible.

Dans un monde sans risque d'accès concurrent à une même ressource, le code suivant vous donne les différentes étapes réalisées par la librairie (`libraire`) et les lecteurs (`lecteur`). Le libraire se charge d'acheter autant de livres que possible, et chaque lecteur essaie d'acheter un et un seul livre.

```
int livre_actuel=0;
int livres_commandes=0;

void libraire(){
  while(1){
    commanderLivre(); // Prend un temps indéfini
    etiqueterLivre(livres_commandes);
    livres_commandes++;
  }
}

void lecteur(){
  while ( livre_actuel >= livres_commandes ){
    // On attend qu'il y ait un livre disponible
  }
  acheterLivre(livre_actuel); // La personne achete le i-eme livre
  livre_actuel++;
}
```

Chaque livre est identifié par un numéro unique (0, puis 1, puis 2, ...), que le libraire inscrit sur chaque livre avec la fonction `etiqueterLivre(numero)`. La fonction `acheterLivre(numero)` permet d'acheter le livre correspondant au numéro courant. Il faut pour cela que ce livre ait été commandé et étiqueté par le libraire. Le libraire stocke le prochain numéro de livre dans la variable `livre_commandes`, et les lecteurs partagent une variable `livre_actuel` qui correspond au numéro du prochain livre à acheter.

On suppose que `commanderLivre()`, `etiqueterLivre()` et `acheterLivre()` sont fournies.

L'objectif de cet exercice est de mettre en place un mécanisme qui assure que l'accès aux livres se fait de manière exclusive (le même livre ne peut pas être acheté par deux personnes).

Dans toute cette partie, le code attendu doit être écrit en C++.

3.1 Première version

La gestion de la concurrence va être réalisée par une classe externe, qui va mettre en place deux méthodes, `ajouterLivre` pour ajouter un livre à une file et `obtenirNumero` pour obtenir un numéro de livre à acheter. Les numéros de livres sont simplement stockés dans des variables de type `int`. Donnez une implémentation de cette classe, 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.

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
```

Question 15 **(1 point)** Écrire le code du `main` qui instancie un thread exécutant `libraire()` et `nb_lecteurs` threads qui exécutent `lecteur()`. Assurez-vous que ce code termine proprement.

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

.....

.....

Question 16 **(0.5 point)** Champs de la classe : ☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

Question 17 **(1 point)** Méthode `ajouterLivre` (paramètres, type de retour et corps de la fonction) : ☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

Question 18 **(1 point)** Méthode `obtenirNumero` (paramètres, type de retour et corps de la fonction) : ☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

Question 19 **(0.5 point)** Modifiez maintenant les fonctions `libraire()` et `lecteur()`. Indiquez où et comment doit être instanciée votre classe. ☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

```
#include <thread>
#include <mutex>
#include <iostream>
#include <vector>
#include <condition_variable>
#include <queue>
#include <unistd.h>
using namespace std;

#define NB_LECTEURS 5

int livre_actuel=0;
int livres_commandes=0;

class ProdCons
{
public:
  void ajouterLivre(int livres_commandes)
  {
    m.lock();
    data.push(livres_commandes);
    c.notify_all();
    m.unlock();
  }

  int obtenirNumero()
  {
    m.lock();
```

```
    while (data.size() == 0) {
      c.wait(m);
    }
    int result = data.front();
    data.pop();

    m.unlock();
    return result;
  }
private:
  mutex m;
  queue<int> data;
  condition_variable_any c;
};

ProdCons les_livres;

void libraire(){
  while(1){
    sleep(1); //commanderLivre(); // Prend un temps indéfini
    //etiqueterLivre(livres_commandes);
    les_livres.ajouterLivre(livres_commandes);
    livres_commandes++;
  }
}

void lecteur(){
  int mon_livre = les_livres.obtenirNumero();
  std::cout << "j'ai␣eu␣le␣livre␣" << mon_livre << std::endl;
}

int main(void){
  thread lib = thread(libraire);

  vector<thread> v;
  for (int i = 0; i < NB_LECTEURS; i++){
    v.push_back(thread(lecteur));
  }

  for (thread &t : v){
    t.join();
  }
  lib.join();
}
```

3.2 Généralisation

On ajoute maintenant quatre imprimeurs, qui impriment et étiquettent les livres avec un numéro unique avant de les fournir au libraire. Le lecteur doit toujours connaître le numéro du livre qu'il a acheté. Pour assurer l'unicité des numéros de livres, chaque imprimeur aura un compteur local `local_cpt` (0, puis 1, puis 2, ...) et un identifiant d'imprimeur `int id_imprimeur`, et chaque livre sera étiqueté avec la valeur `local_cpt + id_imprimeur * 1000000`. Le libraire n'aura plus besoin d'appeler `etiqueterLivre()`. Utilisez le mécanisme mis en place précédemment afin de gérer l'accès aux livres entre le libraire et les imprimeurs. Vous commencerez par créer 4 threads `imprimeur()` qui s'occupent d'imprimer les livres (appel à la fonction `imprimerLivre()`, que l'on suppose disponible) avant de les fournir au libraire. Écrivez le code de la méthode `imprimeur()` et modifiez celui de la méthode `libraire()`.

Attention, une rupture de stock chez un imprimeur ne doit pas gêner l'approvisionnement par

les autres imprimeurs.

Question 20 **(1.5 points)** ☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

```
#include <thread>
#include <mutex>
#include <iostream>
#include <vector>
#include <condition_variable>
#include <queue>
#include <unistd.h>
```

```
using namespace std;

#define NB_LECTEURS 500
#define NB_IMPRIMEURS 4
#define MAX_LIVRES 100000

int livre_actuel=0;
int livres_commandes=0;

class ProdCons
{
public:
  void ajouterLivre(int livres_commandes)
  {
    m.lock();
    data.push(livres_commandes);
    c.notify_all();
    m.unlock();
  }

  int obtenirNumero()
  {
    m.lock();

    while (data.size() == 0) {
      c.wait(m);
    }
    int result = data.front();
    data.pop();

    m.unlock();
    return result;
  }
private:
  mutex m;
  queue<int> data;
  condition_variable_any c;
};

ProdCons les_livres;
ProdCons production_les_livres;

void imprimeur(int id_imprimeur){
  int local_cpt = 0;

  while(1){
    usleep(200); //produireLivre(); // Prend un temps indéfini
    int etiquette = local_cpt + MAX_LIVRES * id_imprimeur;
    production_les_livres.ajouterLivre(etiquette);
    local_cpt++;
  }
}

void libraire(){
  while(1){
    int livre_imprime = production_les_livres.obtenirNumero();
    les_livres.ajouterLivre(livre_imprime);
```

```
  }
}

void lecteur(){
  int mon_livre = les_livres.obtenirNumero();
  std::cout << "j'ai␣eu␣le␣livre␣" << mon_livre << std::endl;
}

int main(void){
  thread lib = thread(libraire);

  vector<thread> v;
  for (int i = 0; i < NB_IMPRIMEURS; i++){
    v.push_back(thread(imprimeur, i));
  }
  for (int i = 0; i < NB_LECTEURS; i++){
    v.push_back(thread(lecteur));
  }

  for (thread &t : v){
    t.join();
  }
  lib.join();
}
```

Afin de limiter le transport de livres entre la librairie et les imprimeurs, les imprimeurs fournissent maintenant les livres au libraire par lots de 1000. Généralisez les méthodes `ajouterLivre()` et `obtenirNumero()` pour pouvoir gérer un tableau d'entiers à la place d'un unique entier.

Question 21 (1.5 points) ☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

on modifie ProdCons pour gerer des tableaux de 1000 entiers à la place d'un entier. Côté libraire, un seul appel a obetnirLivre() (avec 1000 livres dont) puis 1000 appels à ajouterLivre() pour les rendre disponibles à la vente

3.3 Acheter un livre en particulier

On considère maintenant une variante du problème ci-dessus, avec seulement le libraire et le lecteur. Cette fois-ci la solution ne fera plus appel à un moniteur (vous n'utiliserez pas la classe précédemment écrite).

Les lecteurs souhaitent maintenant choisir le livre qu'ils achètent au lieu de prendre le premier

livre disponible. Nous supposons maintenant que le libraire dispose de NB_REFERENCES livres, numérotés de 0 à NB_REFERENCES. Le libraire achète autant de livres que possible et doit se souvenir de la quantité de livres disponibles pour chaque référence. Le lecteur prend désormais en paramètre la référence du livre qu'il souhaite acheter. Gérez la concurrence entre les lecteurs et le libraire en utilisant des sémaphores. Donnez les structures de données nécessaires et le code des fonctions `libraire()` et `lecteur(int reference)`.

Question 22 **(1.5 points)** ☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

tableau de sémaphores de taille NB_REFERENCES, chacune initialisée à 0. le libraire fait V(tab[i])pour ajouter la référence i, le lecteur fait P(tab[i]) pour récupérer la référence i

Sous-total : 8.5
Total : 20