#(Nom)
Note: 13.2/20 (score total : 13.21/20)

+58/1/36+

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 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 étudiant à coder (8 chiffres, il est sur votre carte étudiant !)

- Notez-le ici :

Numéro d'étudiant :

[redacted] .....

- Encodez-le ci-contre (chiffre des unités tout à droite, en remplaçant p de votre login par 1) : par exemple, pour un numéro *p*1234567, ie 11234567 vous grisez le 1 de la première colonne, le 1 de la deuxième, le 2 de la troisième...).

[redacted]

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();
```

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
```

1 Questions de cours

Question 1 **(1 point)** Voici une partie d'un programme où chaque thread effectue une partie d'un calcul :

```
#include <thread>
#include <mutex>
#include <unistd.h>
using namespace std;

double sum = 0.0;
#define NB_STEPS 1000

/* Fonction de calcul, définie ailleurs et qui
   ne nous intéresse pas ici. */
double calcul_local(int id, int i);

void local_sum(int id_thread){
  for (int i = 0; i < NB_STEPS; i++){
    sum += calcul_local(id_thread, i);
  }
}
```

La fonction `main`, qui n'est pas donnée dans ce listing, existe, est correctement programmée, crée plusieurs threads puis attend que tous les threads ont terminé avant de terminer l'exécution du programme.
Quel est le problème ici ? Proposez deux solutions pour le résoudre et analysez leurs différences (avantages et inconvénients des deux solutions).

☐Faux ☐Partiel ☐OK *Réservé* 1/1

Il y a Problème de concurrence, accès à la variable sum partagé
1) Utiliser des mutex, faire m.lock() et m.unlock() avant et après l'affectation de la variable
2) Utiliser un tableau pour stocker les résultats de la somme. Inconvénient : coûteux en mémoire. Avantage : pas de problème de concurrence, pas besoin d'utiliser des mutex

Question 2 **(0.5 point)** Quelle est la politique d'ordonnancement par défaut pour les threads ?

☐Faux ☐Partiel ☐OK *Réservé* 0/0.5

FIFO

Question 3 **(1 point)** Quelle est la différence entre un *thread* (fil d'exécution) et un processus (au sens « processus Unix » par exemple) ? ☐Faux ☐Partiel ☐OK *Réservé* 1/1

Un processus permet de partager un même materiel entre plusieurs utilisateurs, c'est tout ce qui est nécessaire à l'execution. Un processeur utilise des threads (un ou plusieurs). Chaque thread a son propre compteur et son propre pile. Le contexte d'un processeur est lourd, alors que celui d'un thread est rapide.

Question 4 **(1 point)** Quelles sont les différences entre attente active et attente passive ? ☐Faux ☐Partiel ☐OK *Réservé* 1/1

PASSIVE :
Supposons qu'une zone critique est bloquée par un mutex. Pendant ce temps le thread est bloqué, il ne s'utilise pas le processeur, il sera reveillé lors de la liberation du mutex.
ACTIVE : Supposons qu'on applique try-lock dans une zone critique. Alors le thread n'est pas bloqué et il peut faire d'autre chose, puis retourner d'accéder à la zone critique.

Question 5 **(0.5 point)** Qu'est ce que la préemption ? ☐Faux ☐Partiel ☐OK *Réservé* 0.5/0.5

Capacité d'interrompre une tâche en train de s'executer.

Question 6 **(0.5 point)** Est ce que FIFO peut être préemptif ? Pourquoi ? ☐Faux ☐Partiel ☐OK *Réservé* 0.5/0.5

Non, car il y a toujours le même thread qui reste s'executer.

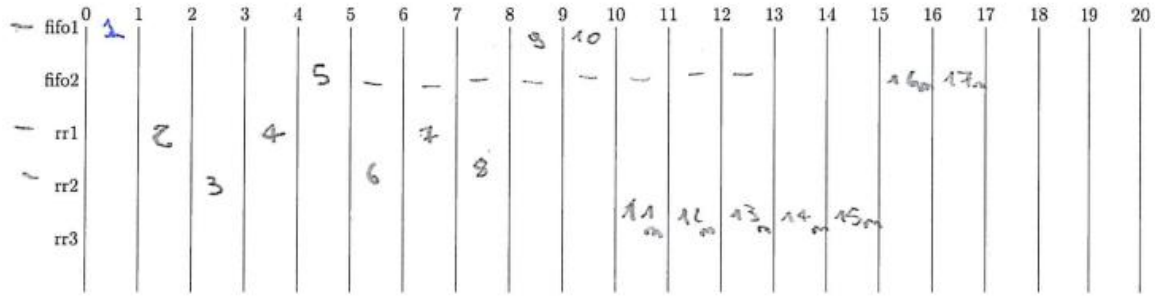
2 Ordonnancement

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 **1 unité de temps**. Les tâches partagent un mutex M.

Tâche	Date d'arrivée	Politique	Priorité	Durée	Remarque
fifo1	0	SCHED_FIFO	2	3	
fifo2	4	SCHED_FIFO	10	3	Après 1 unité de temps, la tâche est interrompue pendant 8 unités de temps pour une lecture disque, puis la tâche peut reprendre (une fois la lecture terminée). Verrouille le mutex M une fois la lecture disque terminée et jusqu'à la fin de son exécution
rr1	1	SCHED_RR	5	3	
rr2	1	SCHED_RR	5	3	
rr3	6	SCHED_RR	1	5	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. 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 :

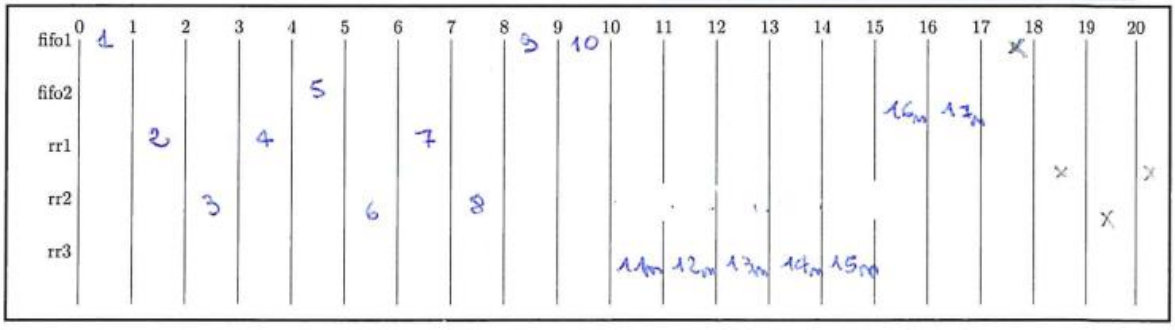


Réponse finale :

Question 7 (3 points)

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 *Réservé*

3/3



Question 8 **(1 point)** Quel est le temps de réponse (ou latence) de chaque tâche sur l'intervalle demandé ? ☐Faux ☐Partiel ☐OK *Réservé* 0.5/1

fifo 1 : 10

fifo 2 : 13

rr1 : 6 rr3 : 5

rr2 : 6

Temps de réponse de fifo1 :

Temps de réponse de fifo2 :

Temps de réponse de rr1 :

Temps de réponse de rr2 :

Temps de réponse de rr3 :

Question 9 **(1.5 points)** Qu'est ce qu'une inversion de priorité ? Est-ce qu'il y a une inversion de priorité ici ? Justifiez. ☐0 ☐1 ☐2 ☐3 *Réservé* 1/1.5

Supposons 2 processus A et B, A plus prioritaire que B, qui partage un même mutex. A un instant donné, B requiert le mutex. Si une tâche de priorité moyenne C arrive à ce moment, le calcul C risque B car plus prioritaire et A ne peut toujours pas s'exécuter car B détient le mutex. C s'exécute alors avant A, même si A a une priorité plus importante. On a une inversion de priorité quand une tache de priorité moins importante s'execute avant une tache plus importante. Entre 13 et 15 unité de temps, rr3 s'execute avant fifo 2 mais priorité fifo 2 > priorité rr3

Question 10 **(1 point)** Quel mécanisme peut être mis en place afin de contourner le problème des inversions de priorité ? ☐Faux ☐Partiel ☐OK *Réservé* 0/1

Dans ce cas, on peut attribuer un priorité = 3 à rr3, de façon à amorcer son execution après 3 unités de temps, laisser executer fifo 2 jusqu'à la fin et finir par rr3 en 2 dernière unité de temps à rr3

3 Gestion de la concurrence pour construire un lotissement de maisons

Un promoteur immobilier a fait l'achat de `nb_parcelles` parcelles sur lesquelles il souhaite construire des maisons. La construction de chaque maison nécessite `nb_etapes` étapes : le terrassement, la construction des murs, la toiture, le carrelage, la peinture... Dans un premier temps, nous supposerons que chaque artisan est spécialisé : il ne peut réaliser qu'une étape de la construction. Par ailleurs, les tâches ont un ordre défini. Par exemple, l'artisan chargé du terrassement (qui sera le premier à travailler) pourra s'occuper de la parcelle suivante une fois la première parcelle terminée. Il avertira le maçon pour qu'il puisse commencer la construction des murs, qui lui même avertira le troisième artisan et ainsi de suite.

3.1 Première version

On suppose qu'il existe une fonction `work(int p, int i)` qui réalise la tâche de l'artisan chargé de l'étape `i` pour la parcelle `p`. On vous donne ici le pseudo code de chaque artisan, qui correspondra à un thread :

```
void travail_chaine(int etape) {
  for (int p=0; p<nb_parcelles; p++) {
    // Attendre que l'artisan qui s'occupe de l'étape
    // précédente ait terminé
    // À FAIRE

    work(p, etape); // Réaliser le travail de l'étape courante.

    // Si on est en charge d'une étape autre que la dernière ...
    if (etape != (nb_etapes-1)) {
      // ... alors prévenir l'artisan qui s'occupe de l'étape
      // suivante qu'il peut commencer
      // À FAIRE
    }
    // Si on est en charge de la dernière étape
    else {
      fprintf(stdout, "maison␣%d␣terminée\n", p);
    }
  }
}
```

Question 11 **(1 point)** Écrire le code du `main` qui instancie les `nb_etapes` threads. Assurez-vous que ce code termine proprement.

☐ Faux ☐ Partiel ■ OK *Réservé* 1/1

```
int main(void){
vector<thread> thr;
for(int i=0; i<nb_etapes; i++)
{thr.push_back(thread(travail_chaine, i));}
for(int i=0; i<nb_etapes; i++)
{thr[i].join();}}
```

Question 12 **(2.5 points)** Mettez maintenant en place le mécanisme utilisant un moniteur de Hoare pour attendre *efficacement* la fin du travail de l'artisan précédent et prévenir l'artisan suivant. On attend ici les champs utilisés, les différentes méthodes, l'adaptation/modification de la fonction `travail_chaine()` (entête et corps) et l'instanciation des différents objets.

☐0 ☐1 ☐2 ☐3 ☒4 ☐5 *Réservé*

2/2.5

```
class Prod_Cons {
private:
int nb_etapes;
vector<int> maisons;
mutex m;
vector<condition_variable_any> c;
public:
Prod_Cons(int NETAPES, int MAISONS)
void attend_travail(int etape);
void transmet_travail(int etape); }
void travail_recoit(Prod_Cons p){
for(int i=0; i<p.maisons; i++)
{ p.attend_travail(i);
p.transmet_travail(i); }}
```

```
void attend_travail(int etape){
m.lock();
while(maisons[etape]==0)
c[etape].wait(m); }
maisons[etape]--;
m.unlock(); }
```

```
void transmet_travail(int etape){
m.lock()
maisons[etape]++;
c[etape].notify_one();
m.unlock(); }
```

Question 13 **(1 point)** Justifiez pourquoi la solution que vous avez mise en place est la plus efficace possible.

☒ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4 *Réservé* 0/1

On utilise le moniteur de Hoare. On n'a pas de problème de deadlock ni de ~~famine~~ concurrence car le while fait exécuter un thread à la fois. On n'a pas de problème de famine, toutes les mains parviennent à ~~un~~ par toutes les étapes. Dès qu'on finit une étape, elle transmise à la suivante (y compris la dernière).

Question 14 **(0.5 point)** Que se passe t-il si on a plusieurs artisans qui peuvent réaliser une même tâche (par exemple plusieurs plombiers) ? Votre code doit-il être modifié ?

☒ Faux ☐ Partiel ☐ OK *Réservé* 0/0.5

Programme de concurrence. On devrait surveiller la tâche en question et appliquer à nouveau le moniteur de Hoare.

3.2 Avec des sémaphores

On veut maintenant implémenter le même mécanisme avec des sémaphores. L'idée est d'utiliser un tableau de sémaphores pour pouvoir attendre la fin de la tâche précédente et prévenir que la tâche suivante peut être réalisée.

Question 15 **(2 points)** Donnez le code correspondant, en précisant les structures de données utilisées, leur initialisation, *etc.*

☐0 ☐1 ☒2 ☐3 *Réservé*

1.333/2

On pourrait créer nb_etapes sémaphores et les initialiser à 0. Une fois la tâche transmise on incrémente le compteur du sémaphore. Quand on veut savoir si la tâche en amont est terminée (système analogue des mutex).

3.3 Avec plus de parallélisme

On suppose maintenant que les étapes 2 et 3 peuvent être réalisées en parallèle. Par exemple, les tâches de carrelage (dans la cuisine) et de peinture (dans les chambres) pourront être réalisées simultanément. Comment pourriez vous modifier le code du travail à la chaîne pour vous adapter à cette nouvelle contrainte ? Expliquez comment généraliser l'approche.

Question 16 **(1 point)** Donnez le code correspondant, en étant le plus précis possible.

☐0 ☐1 ☐2 ☐3 *Réservé* 0/1

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

3.4 En utilisant une liste de tâches

Pour finir, on considère une nouvelle variante du problème où tous les artisans peuvent réaliser chacune des tâches, ils ne sont plus spécialisés. Afin d'optimiser le parallélisme de l'approche, nous utiliserons une liste de tâches, et les artisans (pool de threads) piocheront dans cette liste la prochaine tâche a effectuer. Vous pouvez utiliser les classes définies en TP ou en TD sans réécrire le code correspondant.

Question 17 **(1.5 points)** Donnez le code correspondant, en précisant les structures de données et les nouvelles méthodes. Quel est l'avantage de l'approche ?

☐0 ☒1 ☐2 ☐3 ☐4 *Réservé* 0.375/1.5

Ici... peut... utiliser... une... file... de... tâches... L'avantage... est... que... tous... les... artisans... sont... toujours... en... travail... d'exécuter... une... tâche,... ce... qui... réduit... la... diminue... le... temps... d'exécution... et... construction... de... toutes... les... maisons