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 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 :
.....

- Encodez-le ci-contre (chiffre des unités tout à droite, en remplaçant p de votre login par 1) : par exemple, pour un numéro $p1234567$, ie 11234567 vous grisez le 1 de la première colonne, le 1 de la deuxième, le 2 de la troisième...).

☐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();
```

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 Question de cours

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

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

Question 2 **(1 point)** Qu'est-ce que l'ordonnancement preemptif ? l'ordonnancement collaboratif ?

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

Question 3 **(1 point)** Donnez brièvement les différences entre threads et processus (au sens Unix du terme).

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

Question 4 **(1 point)** Quelles sont les différences entre attente active et attente passive ?

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

Sous-total : 4

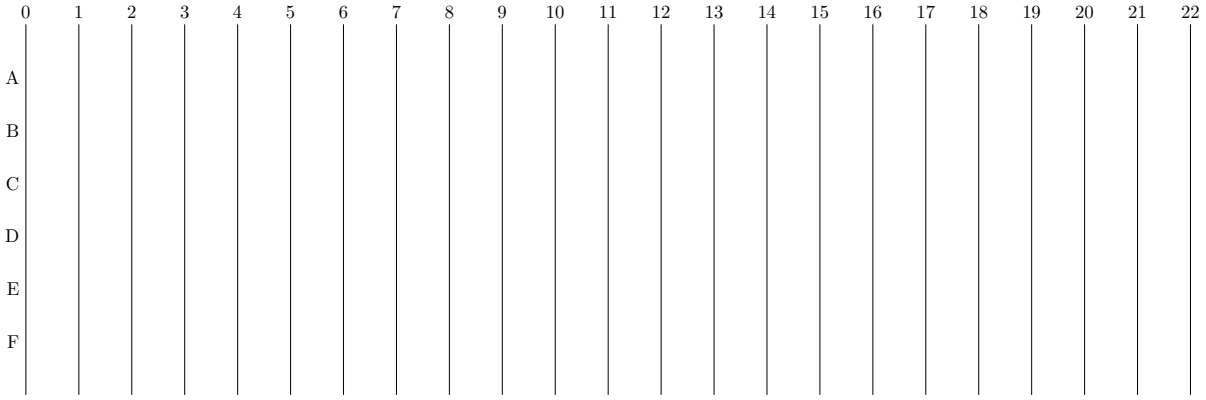
2 Ordonnancement

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 2 unités de temps. Les tâches partagent un mutex M.

Tâche	Date d'arrivée	Politique	Priorité	Durée	Remarque
A	0	SCHED_FIFO	0	3	
B	1	SCHED_FIFO	5	3	
C	2	SCHED_RR	10	3	
D	12	SCHED_RR	2	2	prend le mutex M durant toute son execution
E	13	SCHED_RR	4	4	prend le mutex M durant toute son execution
F	14	SCHED_RR	4	4	

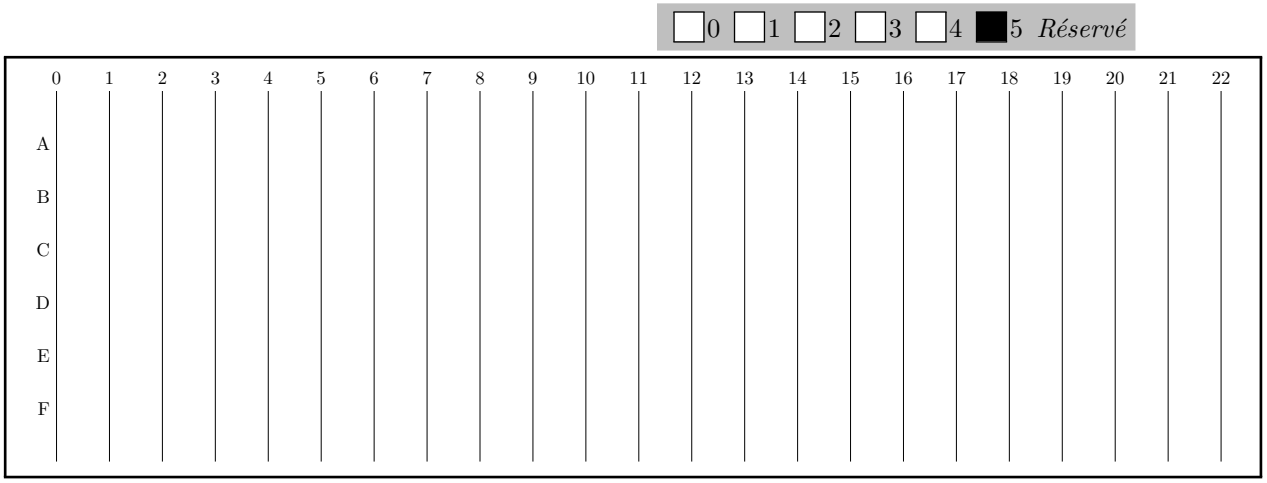
Faire l'ordonnancement de ces tâches. Vous pouvez utiliser le brouillon si besoin. Barrez la réponse incorrecte si vous répondez plusieurs fois. Si vous avez vu une autre présentation en TD (sur une seule ligne à la place d'une ligne par tâche), vous pouvez représenter votre ordonnancement de cette manière.

Brouillon :



Réponse finale :

Question 5 (3 points)



B passe devant A (plus prioritaire), puis C passe devant (plus prioritaire). C ne rend pas la main (meme si RR). D prend le mutex donc bloque E. puis E/F en RR, quantum 2

```
A  1                          8  9
B     2              6  7
C        3  4  5
D                                    12 13
E                                       14 15          18 19
F                                             16 17          20 21
```

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

Question 6 (0.25 point)

Temps de réponse de A : 9 ..... ☐Faux ■OK *Réservé*

Question 7 (0.25 point)

Temps de réponse de B : 6 ..... ☐Faux ■OK *Réservé*

Question 8 (0.25 point)

Temps de réponse de C : 3 ..... ☐Faux ■OK *Réservé*

Question 9 (0.25 point)

Temps de réponse de D : 2 ..... ☐Faux ■OK *Réservé*

Question 10 (0.25 point)

Temps de réponse de E : 7 ..... ☐Faux ■OK *Réservé*

Question 11 (0.25 point)

Temps de réponse de F : 8 ..... ☐Faux ■OK *Réservé*

Question 12 (1 point) Qu'est-ce qu'une inversion de priorité ? Y en a-t-il une ici ? Si oui à quel moment ?

Non ce n'est pas une inversion ici

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

.....

Sous-total : 5.5

3 Implémentation de la fonction `poll`

On s'intéresse à l'implémentation d'un mécanisme similaire à la fonction `poll` d'Unix. L'objectif de cette fonction est d'attendre qu'un descripteur de fichier parmi un ensemble soit prêt pour effectuer des entrées-sorties. Dans le cadre d'un modèle client/serveur, cette fonction peut par exemple servir au serveur pour attendre qu'une des sockets associées aux divers clients ait des données à lire. Dans cette partie, nous allons modéliser l'implémentation de cette fonction dans ce contexte.

Dans un monde sans risque d'accès concurrent à une même ressource, le code suivant vous donne une idée des différentes étapes pour un client (`unClient`) et le serveur (`serveur`) :

```
int peut_faire_ES=0;

void serveur(){
  while(1){
    if (peut_faire_ES) {
      peut_faire_ES--;  // Pose problème en parallèle
                        // à remplacer par un mécanisme robuste
      d = lireDonnee(id);  // Il faudra ajouter un mécanisme pour récuperer id
      consommerDonnee(d);
    }
  }
}

void unClient(int id, int nb_writes){
  for (int i = 0; i < nb_writes; i++){
    produireDonnee(id);
    peut_faire_ES++;  // Pose problème en parallèle,
                      // à remplacer par un mécanisme robuste
  }
}
```

Un client manipule un descripteur de fichier associé à son identifiant (numéro de thread). Il produit une certaine quantité de données (`nb_writes` données en tout) sur ce descripteur via l'appel à la fonction `produireDonnee()`. De son côté, le serveur fait une boucle infinie. Dès qu'il y a une donnée à consommer sur un des descripteurs de fichiers, ce qui est indiqué par la variable `peut_faire_ES`, il fait les opérations suivantes : `lireDonnee()` pour lire la donnée et `consommerDonnee()` pour l'utiliser. Le mécanisme pour récupérer le bon descripteur de fichier (l'identifiant du thread qui a produit la donnée dans notre contexte) n'est volontairement pas donné, ce sera à vous de le gérer.

Les fonctions `lireDonnee()`, `consommerDonnee()` et `produireDonnee()` ne sont pas à implémenter dans ce sujet.

Le but ici est de reproduire en plus simple ce que fait un système d'exploitation. On se place dans un cas où on ne dispose pas d'opérations comme `read()` qui est bloquant quand les données ne sont pas disponibles. `lireDonnee(id);` provoquerait ici une erreur si on l'appelle avant le `produireDonnee(id);` correspondant. L'objectif de l'exercice est donc de garantir qu'on ne lit jamais de données qui ne sont pas encore disponibles.

C'est à vous de proposer un mécanisme pour stocker les descripteurs de fichier sur lesquels on peut faire des opérations. On supposera que les clients font uniquement des écritures et que le serveur veut faire des lectures.

Question 13 **(1 point)** Écrire le code du `main` qui instancie un thread exécutant `serveur()` et `NB_CLIENTS` threads qui exécutent `unClient(int id, int nb_writes)`. On supposera qu'il existe un tableau `nb_writes` de taille `NB_CLIENTS` tel que `nb_writes[i]` soit le nombre d'écritures réalisées par le thread d'identifiant `i`. Assurez-vous que ce code termine proprement.

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

.....

.....

.....

.....

La gestion de la concurrence va être réalisée par une classe externe, qui va mettre en place deux méthodes, `putData(id)` pour indiquer qu'il est possible de faire une lecture sur le descripteur de fichier "id" et `getData()` pour obtenir un descripteur de fichier sur lequel on peut faire une lecture. 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.

Question 14 **(0.5 point)** Champs de la classe : ☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

.....

.....

.....

.....

.....

Question 15 **(2 points)** Méthode `putData` (paramètres, type de retour et corps de la fonction) :

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

Question 16 **(2 points)** Méthode `getData` (paramètres, type de retour et corps de la fonction) :

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

Question 17 **(1 point)** Modifiez maintenant les fonctions `serveur()` et `unClient()` pour utiliser votre moniteur. Indiquez où et comment doit être instanciée votre classe.

☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

Question 18 **(1 point)** Dans l'implémentation actuelle, le serveur fait une boucle infinie (`while(1)`). Proposez un mécanisme pour qu'il puisse terminer proprement.

☐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_CLIENTS 5
#define NB_WRITES 10

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

  int getData()
  {
    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 sockets;

void serveur(){
  while(1){
    int data_fd = sockets.getData();
    if (data_fd == -1){
      break;
    }
    cout << "Je␣lis␣depuis␣" << data_fd << endl;
    // d = lireDonnee(data_fd);
    // consommerDonnee(d);
  }
}
```

```
void unClient(int id, int nb_writes){
  for (int i = 0; i < nb_writes; i++){
    // produireDonnee(id);
    sockets.putData(id);
  }
}

int main(void){
  int nb_writes[NB_CLIENTS];
  for (int i = 0; i < NB_CLIENTS; i++){
    nb_writes[i] = rand()%10;
  }

  thread serv = thread(serveur);

  vector<thread> v;
  for (int i = 0; i < NB_CLIENTS; i++){
    v.push_back(thread(unClient, i, nb_writes[i]));
  }

  for (thread &t : v){
    t.join();
  }
  sockets.putData(-1);
  serv.join();
}
```

Sous-total : 7.5

3.1 Autre approche : un serveur multithread

Dans la partie précédente, nous avons vu comment l'implémentation de la fonction `poll` pouvait permettre à un serveur d'attendre sur plusieurs entrées/sorties. Une autre solution, largement utilisée, serait d'implémenter un serveur multi-thread. Chaque thread du serveur s'occupe d'un unique client et inversement : on a donc autant de threads serveur que de threads client. Le mécanisme de `poll` n'est plus utile : chaque thread serveur va attendre qu'il soit possible de faire une entrée/sortie sur le descripteur de fichier qui lui est associé.

Question 19 **(0.5 point)** Quelles sont les modifications que vous devez faire dans le `main` ? On suppose désormais que la fonction `serveur` prend en paramètre un indice pour indiquer de quelle communication s'occupe le thread (on rappelle qu'on a désormais un thread serveur par thread client). ☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Question 20 **(1 point)** Quelles modifications devez-vous faire dans votre moniteur (`putData()`, `getData()`, etc.) ? Indiquer quelles lignes de votre code doivent être modifiées et les nouvelles structures à utiliser. ☐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_CLIENTS 5
#define NB_WRITES 10

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

  int getData()
  {
    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 sockets[NB_CLIENTS];

void serveur(int id){
  while(1){
    int data_fd = sockets[id].getData();
    if (data_fd == -1){
      break;
    }
    cout << "Je␣suis␣" << id << ",␣je␣lis␣depuis␣" << data_fd << endl;
    // d = lireDonnee(data_fd);
    // consommerDonnee(d);
  }
}

void unClient(int id, int nb_writes){
  for (int i = 0; i < nb_writes; i++){
    // produireDonnee(id);
    sockets[id].putData(id);
  }
}

int main(void){
  int nb_writes[NB_CLIENTS];
  for (int i = 0; i < NB_CLIENTS; i++){
    nb_writes[i] = rand()%10;
  }

  vector<thread> v_client;
  vector<thread> v_serveur;
  for (int i = 0; i < NB_CLIENTS; i++){
    v_client.push_back(thread(unClient, i, nb_writes[i]));
    v_serveur.push_back(thread(serveur, i));
  }

  for (int i = 0; i < NB_CLIENTS; i++){
    v_client[i].join();
    sockets[i].putData(-1);
    v_serveur[i].join();
  }
}
```

Gérez maintenant la concurrence entre les clients et le serveur multi-threadé en utilisant des sémaphores à la place du moniteur de Hoare des questions précédentes. On a toujours un thread serveur pour chaque thread client. On vous demande ici d'utiliser le pseudo code vu en CM et en TD pour l'utilisation des sémaphores.

Question 21 **(1.5 points)** Donnez les structures de données nécessaires à la gestion de la concurrence ainsi que le code des fonctions `unClient(int id, int nb_writes)` et `serveur(int id)`. ☐0 ☐1 ☐2 ☐3 ☐4 ■5 *Réservé*

On utilise un tableau de sémaphores, initialisés à 0, une par connexion. Un client fait un V() pour produire, un serveur un P() pour consommer

Sous-total : 3 Total : 20