HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Prof. Yann Thoma
Prof. Alberto Dassatti
Florian Vaussard

# Exercices de programmation concurrente (PCO)

semestre automne 2025 - 2026

## Exclusion mutuelle, synchronisation, coordination

---

## **Question 1:** Exclusion mutuelle

Problème 1 : Soit la séquence d'instructions suivante :

```
void fonction() {
    int nombre=0;
    for(int i=0;i<10;i++)
        nombre++;
    printf("Nombre: %d\n",nombre);
}
```

Si deux threads sont lancés concurremment avec cette fonction à exécuter, quelles sont les valeurs affichées possibles en fin d'exécution ?
Problème 2 : Même question, mais avec le code suivant :

```
void fonction() {
    static int nombre=0;
    for(int i=0;i<10;i++)
        nombre++;
    printf("Nombre: %d\n",nombre);
}
```

## **Question 2:** Exclusion mutuelle

```
static int nombre;

void threadA() {
    nombre=0;
    nombre++;
    printf("Nombre: %d\n",nombre);
}

void threadB() {
    nombre=1;
    nombre*=2;
    printf("Nombre: %d\n",nombre);
}
```

Si ces fonctions sont exécutées de manière concurrente, quelles sont les valeurs possibles de `nombre` en fin d'exécution ?

**Question 3:** Exclusion mutuelle

Soit les deux tâches $T_0$ et $T_1$ données sur le code suivant et devant réaliser l'exclusion mutuelle dans leur section critique.

```
1  static bool intention[2] = {false,false};
2  static int tour = 0; // ou 1
3
4  void T0()
5  {
6      while (true) {
7          intention[0] = true;
8          while (tour != 0) {
9              while (intention[1])
10                 ;
11             tour = 0;
12         }
13         /* section critique */
14         intention[0] = false;
15         /* section non-critique */
16     }
17 }
18
19 void T1()
20 {
21     while (true) {
22         intention[1] = true;
23         while (tour != 1) {
24             while (intention[0])
25                 ;
26             tour = 1;
27         }
28         /* section critique */
29         intention[1] = false;
30         /* section non-critique */
31     }
32 }
```

L'exclusion mutuelle est-elle garantie par les parties prélude et postlude des tâches ? Justifiez votre réponse.

Le code peut être récupéré ici : 

## **Question 4:** Verrous

Commentez la différence entre les deux codes suivants :

```
void taskA() {
    PcoMutex mutex;
    mutex.lock();
    std::cout << "Section critique" << std::endl;
    mutex.unlock();
}
```

```
void taskB() {
    static PcoMutex mutex;
    mutex.lock();
    std::cout << "Section critique" << std::endl;
    mutex.unlock();
}
```

## **Question 5:** Synchronisation

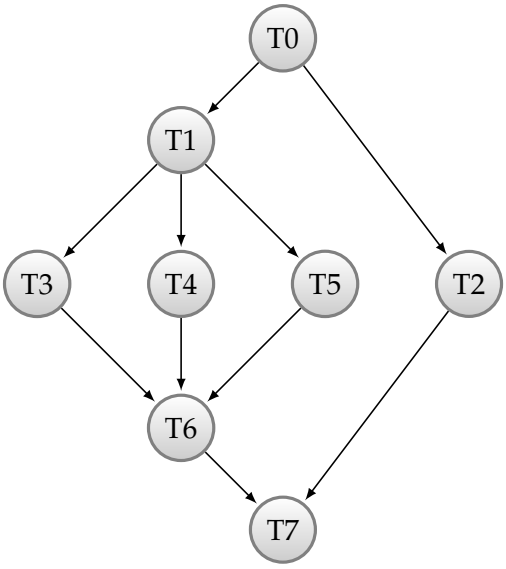
Nous désirons réaliser une application possédant 2 tâches. Le programme principal est en charge de lancer les deux tâches. Une fois démarrées, les tâches doivent attendre un signal du programme principal pour s'exécuter, ceci afin de garantir que toutes les tâches sont créées avant qu'elles ne commencent effectivement leur traitement.
Comment résoudre le problème

1. à l'aide de sémaphores ?
2. à l'aide de mutex ?

Un squelette de code est présent ici : 

## **Question 6:** Séquentialité

Soit le graphe d'exécution des tâches suivant :



Les tâches doivent exécuter leur traitement dans l'ordre indiqué par les flèches. En l'occurence la tâche $T0$ doit être terminée avant que $T1$ et $T2$ ne commencent.

1. Ecrire un programme permettant de garantir ce fonctionnement, en utilisant la fonction `PcoThread::join()`.
2. Faire de même, en utilisant des sémaphores pour la synchronisation.

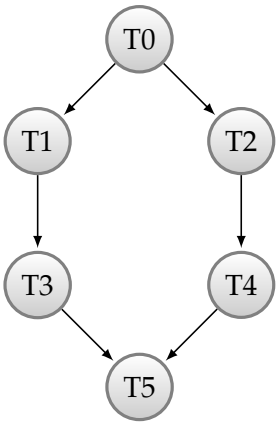
Un squelette de code est présent ici : 

---

**Question 7:** Séquentialité

---

Soit le graphe d'exécution des tâches suivant :




Les tâches doivent exécuter leur traitement dans l'ordre indiqué par les flèches.

1. Ecrire un programme permettant de garantir ce fonctionnement, en utilisant la fonction `PcoThread::join()`.
2. Faire de même, en utilisant des sémaphores pour la synchronisation.

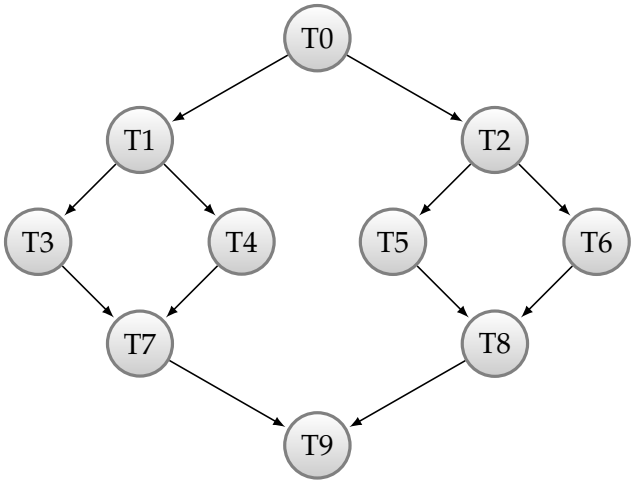
Un squelette de code est présent ici : 

---

**Question 8:** Séquentialité

---

Soit le graphe d'exécution des tâches suivant :




Les tâches doivent exécuter leur traitement dans l'ordre indiqué par les flèches.

1. Ecrire un programme permettant de garantir ce fonctionnement, en utilisant la fonction `PcoThread::join()`.
2. Faire de même, en utilisant des sémaphores pour la synchronisation.

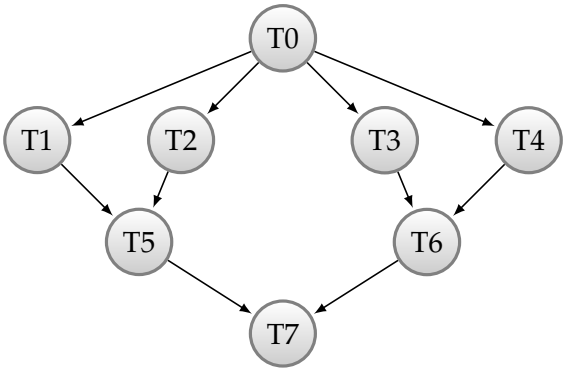
Un squelette de code est présent ici :

---

**Question 9:** Séquentialité

---

Soit le graphe d'exécution des tâches suivant :




Les tâches doivent exécuter leur traitement dans l'ordre indiqué par les flèches.

1. Ecrire un programme permettant de garantir ce fonctionnement, en utilisant la fonction `PcoThread::join()`.
2. Faire de même, en utilisant des sémaphores pour la synchronisation.

Un squelette de code est présent ici :

---

**Question 10:** Verrou-sémaphore

---

Réaliser un verrou à base de sémaphores. La sémantique du verrou doit évidemment être préservée. Si un verrou est déjà déverouillé et que la fonction **`unlock`**`()` est appelée, alors cet appel ne doit pas avoir d'effet. Afin de simplifier le problème, il n'est pas nécessaire de garantir un ordre de réveil des threads. Le verrou aura l'interface suivante :

```
class MutexFromSem
{
public:
    MutexFromSem();
    ~MutexFromSem();

    void lock();
    void unlock();
    bool trylock(); //! Returns true if the mutex was
        successfully acquired, otherwise returns false without
        waiting if it is already locked
};
```

Un squelette de code est présent ici :

## **Question 11:** Barrière de synchronisation

Nous sommes intéressés à la réalisation d'une barrière de synchronisation. Une telle barrière fonctionne de la manière suivante :
Pour une barrière initialisée à $N$, les $N-1$ threads faisant appel à la méthode `arrive()` seront bloqués. Lorsque le $N^{\text{ème}}$ thread appel `arrive()`, tous les threads sont relâchés et peuvent poursuivre leur exécution.
La barrière aura l'interface suivante :

```
class PcoBarrier
{
public:
    PcoBarrier(unsigned int nbToWait);
    ~PcoBarrier();

    void arrive();
};
```

Trois types de solution peuvent être proposées, exploitant :

1. Des sémaphores
2. Des variables de condition
3. Un moniteur de Hoare

Un squelette de code est présent ici : 

## **Question 12:** Coordination

Nous désirons contrôler l'accès à un pont suspendu. Ce pont est capable de supporter un poids de `maxWeight` tonnes. Une voiture pèse 1 tonne, et un camion 10 tonnes. En considérant que les voitures et camions arrivant devant le pont sont modélisés par des threads, et que le pont est une ressource critique, écrire le code permettant la gestion correcte de l'accès au pont.
L'interface du gestionnaire de pont doit être la suivante. Le nombre `maxWeight` de tonnes supportées par le pont est passé en paramètre du constructeur.

```
class BridgeManager
{
public:
    BridgeManager(unsigned int maxWeight);
    ~BridgeManager();

    void carAccess();
    void truckAccess();
    void carLeave();
    void truckLeave();
}
```

Trois types de solution peuvent être proposées, exploitant :

1. Des sémaphores
2. Des variables condition
3. Un moniteur de Hoare

Un squelette de code est présent ici : 

---

**Question 13:** Coordination

---

Nous désirons contrôler l'accès à un pont suspendu. Ce pont est capable de supporter un poids de `maxWeight` tonnes. Les véhicules arrivent avec leur poids propre, qui est représenté en virgule flottante. En considérant que les véhicules arrivant devant le pont sont modélisés par des threads, et que le pont est une ressource critique, écrire le code permettant la gestion correcte de l'accès au pont.
L'interface du gestionnaire de pont doit être la suivante. Le nombre `maxWeight` de tonnes supportées par le pont est passé en paramètre du constructeur.

```
class BridgeManager
{
public:
    BridgeManager(float maxWeight);
    ~BridgeManager();

    void access(float weight);
    void leave(float weight);
}
```

Trois types de solution peuvent être proposées, exploitant :

1. Des sémaphores
2. Des variables de condition
3. Un moniteur de Hoare

Un squelette de code est présent ici : 

---

**Question 14:** Coordination

---

Nous désirons contrôler l'accès à un pont suspendu. Ce pont est capable de supporter un poids de `maxWeight` tonnes. Les véhicules arrivent avec leur poids propre, qui est représenté en virgule flottante. En considérant que les véhicules arrivant devant le pont sont modélisés par des threads, et que le pont est une ressource critique, écrire le code permettant la gestion correcte de l'accès au pont. Le manager doit appeler les méthodes **`start`**`()` et `stop()` pour arrêter et faire redémarrer les véhicules si nécessaire.
L'interface du gestionnaire de pont doit être la suivante. Le nombre `maxWeight` de tonnes supportées par le pont est passé en paramètre du constructeur.

```
class Vehicle
{
public:
    Vehicle(float weight);
    float getWeight() const;
    void stop();
    void start();
}

class BridgeManager
{
public:
    BridgeManager(float maxWeight);
    ~BridgeManager();

    void access(Vehicle *vehicle);
    void leave(Vehicle *vehicle);
}
```

Trois types de solution peuvent être proposées, exploitant :

1. Des sémaphores
2. Des variables de condition
3. Un moniteur de Hoare

Un squelette de code est présent ici : 

---

**Question 15:** Producteurs-consommateurs

---

Nous désirons réaliser un tampon simple dont le fonctionnement implique que chaque donnée qui y est placée doit être consommée par 2 consommateurs. L'interface de ce buffer sera la suivante :

```
template<typename T> class Buffer2Conso : public
    AbstractBuffer<T> {
public:
    Buffer2Conso();

    virtual ~Buffer2Conso();

    virtual void put(T item);
    virtual T get(void);
};
```

Trois types de solution peuvent être proposées, exploitant :

1. Des sémaphores
2. Des variables de condition
3. Un moniteur de Hoare

Un squelette de code est présent ici : 

---

**Question 16:** Producteurs-consommateurs

---

Nous désirons réaliser un tampon multiple dont le fonctionnement implique que chaque donnée qui y est placée doit être consommée par 2 consommateurs. L'interface de ce buffer sera la suivante :

```
template<typename T> class Buffer2Conso : public
    AbstractBuffer<T> {
public:
    Buffer2Conso(unsigned int nbElements);

    virtual ~Buffer2Conso();

    virtual void put(T item);
    virtual T get(void);
}
```

Trois types de solution peuvent être proposées, exploitant :

1. Des sémaphores
2. Des variables de condition
3. Un moniteur de Hoare

Un squelette de code est présent ici : 

---

**Question 17:** Sémaphore par moniteur

---

Réaliser un sémaphore faible grâce à :

1. Des variables de conditions
2. Un moniteur de Hoare

Un sémaphore faible n'a pas besoin de respecter un ordre FIFO pour sa file d'attente. Les threads en attentes peuvent donc être relâchés dans un ordre quelconque (ceci simplifie quelque peu la solution).
Le sémaphore aura l'interface suivante :

```
class SemaphoreFromMonitor
{
public:
    SemaphoreFromMonitor(unsigned int initValue);
    ~SemaphoreFromMonitor();

    void wait();
    void post();
    bool trywait(); //! Same as wait, but won't block. Returns
        true if the semaphore value was strictly positive,
        false else
};
```

Un squelette de code est présent ici : 

---

**Question 18:** Préchargement

---

Dans le cadre d'un programme devant récupérer des informations d'un grand nombre de fichiers lors d'une action spécifique de l'utilisateur, nous sommes intéressés à précharger ces fichiers afin d'offrir une meilleure ergonomie à l'application. La classe gérant le chargement des fichiers est `SimpleLoader`. Elle peut être utilisée telle quelle, mais nous sommes intéressés par une nouvelle implémentation permettant de faire le chargement en tâche de fond.
Pour ce faire nous devons implémenter la classe `ThreadedLoader`.

```
class Data;

class ILoader
{
public:
    Data* getData() = 0;
};

class SimpleLoader : public ILoader
{
public:
    Data* getData();
};

class ThreadedLoader : public ILoader
{
public:
    ThreadedLoader();
    ~ThreadedLoader;

    void startPreloading();
    Data* getData();
};
```

La fonction `getData()` de `ThreadedLoader` est exploitée tout au long du programme pour accéder les données chargées. Cette fonction est potentiellement bloquante si les données ne sont pas encore chargées. La fonction `startPreloading()` peut être appelée en début de programme, et son rôle est de lancer un thread qui sera responsable de charger l'ensemble des fichiers (non bloquante).
Le chargement réel des données est fait via l'appel de la méthode `getData()` de la classe `SimpleLoader` depuis les méthodes de `ThreadedLoader`.
Attention, l'usage de `startPreloading()` n'est pas obligatoire, et dès lors `getData()` doit gérer le chargement si celui-ci n'a pas été commencé. Le pointeur sur `DataDescriptor` retourné par `getData()` sera censé contenir les informations récupérées (non géré dans l'exercice).
Trois types de solution peuvent être proposées, exploitant :

1. Des sémaphores
2. Des variables de condition
3. Un moniteur de Hoare

Un squelette de code est présent ici : 

---

**Question 19:** Lecteurs-rédacteurs

---

Lors du dernier festival Baleinev, un souci technique a impliqué que les toilettes hommes se sont retrouvées fermées. Ne pouvant pas tolérer que les hommes et les femmes partagent les même toilettes, il a fallut gérer l'accès à ce précieux local en garantissant qu'il n'y avait pas possibilité d'avoir des hommes et des femmes en même temps dans celui-ci.
Ce ne fut pas une mince affaire.
Afin de choisir la meilleure manière de gérer un tel scénario l'année prochaine, l'équipe de Baleinev vous demande de modéliser différentes politiques de gestion des accès.

a. Les femmes sont prioritaires sur les hommes s'il y a déjà des femmes aux toilettes
b. Les femmes ont la priorité sur les hommes
c. Les personnes sont gérées selon leur ordre d'arrivée
d. Après un maximum de N femmes, c'est aux hommes d'y accéder, et après un maximum de N hommes, c'est aux femmes

Les toilettes disposent d'un nombre de places limitées, vous vous en doutez. Pour votre solution, vous allez implémenter une sous-classe de `AbstractToilet`. Le nombre de places disponibles aux toilettes est directement passé au constructeur.

```
class AbstractToilet {
public:
    AbstractToilet(int nbSeats) : nbSeats(nbSeats) {};
    virtual void manAccessing() = 0;
    virtual void manLeaving() = 0;
    virtual void womanAccessing() = 0;
    virtual void womanLeaving() = 0;

protected:
    int nbSeats;
}
```

Trois types de solution peuvent être proposées, exploitant :

1. Des sémaphores
2. Des variables de condition
3. Un moniteur de Hoare

Un squelette de code est présent ici : 

---

**Question 20:** Lecteurs-rédacteurs

---

Reprendre l'algorithme général avec priorité aux lecteurs, et l'adapter pour la priorité aux rédacteurs.
Après une version exploitant les sémaphores, transformer celle-ci pour en faire une version avec variables conditions, puis moniteur de Hoare.
Les trois types de solution pouvant être proposées sont donc :

1. Des sémaphores
2. Des variables de condition
3. Un moniteur de Hoare

Un squelette de code est présent ici : 

---

**Question 21:** Lecteurs-rédacteurs

---

Une application est composée de threads de deux classes, A et B. Une ressource est partagée entre tous les threads, selon les contraintes suivantes :

1. Les threads de classe A peuvent accéder concurrement à la ressource.
2. Les threads de classe B peuvent accéder concurrement à la ressource.
3. Les threads de différente classe ne peuvent accéder à la ressource au même instant.

Proposer un algorithme permettant de gérer l'accès à la ressource, en s'inspirant des solutions du chapitre. Considérez une solution où la coalition est possible entre threads d'une même classe.
Trois types de solution peuvent être proposées, exploitant :

1. Des sémaphores
2. Des variables de condition
3. Un moniteur de Hoare

Un squelette de code est présent ici :