Programmation système (2° partie) : Programmation concurrente

Thierry Vaira

BTS SN-IR Avignon

v0.1 17 février 2016

Sommaire

Frères des Ecoles Chrétiennes
Saint-Jean-Baptiste de la Salle - Avignon

Processus lourds vs processus légers

- **Multitâche moins coûteux pour les *threads* (processus léger)** : puisqu'il n'y a pas de changement de mémoire virtuelle, la commutation de contexte (*context switch*) entre deux *threads* est moins coûteuse que la commutation de contexte obligatoire pour des processus lourds.
- **Communication entre *threads* plus rapide et plus efficace** : grâce au partage de certaines ressources entre *threads*, les IPC (*Inter Processus Communication*) sont inutiles pour les *threads*.
- **Programmation utilisant des *threads* est toutefois plus difficile** : obligation de mettre en place des mécanismes de synchronisation, risques élevés d'interblocage, de famine, d'endormissement.

Introduction

Dans la programmation concurrente, le terme de **synchronisation** se réfère à deux concepts distincts (mais liés) :

- La **synchronisation de processus ou tâche** : mécanisme qui vise à bloquer l'exécution des différents processus à des points précis de leur programme de manière à ce que tous les processus passent les étapes bloquantes au moment prévu par le programmeur.
- La **synchronisation de données** : mécanisme qui vise à conserver la cohérence entre différentes données dans un environnement multitâche.

Les problèmes liés à la synchronisation rendent toujours la programmation plus difficile.

Définitions

- **Section critique** : C'est une partie de code telle que deux *threads* ne peuvent s'y trouver au même instant.
- **Exclusion mutuelle** : Une ressource est en exclusion mutuelle si seul un *thread* peut utiliser la ressource à un instant donné.
- **Chien de garde (*watchdog*)** : Un chien de garde est une technique logicielle (compteur, *timeout*, signal, ...) utilisée pour s'assurer qu'un programme ne reste pas bloqué à une étape particulière du traitement qu'il effectue. C'est une protection destinée généralement à redémarrer le système, si une action définie n'est pas exécutée dans un délai imparti.

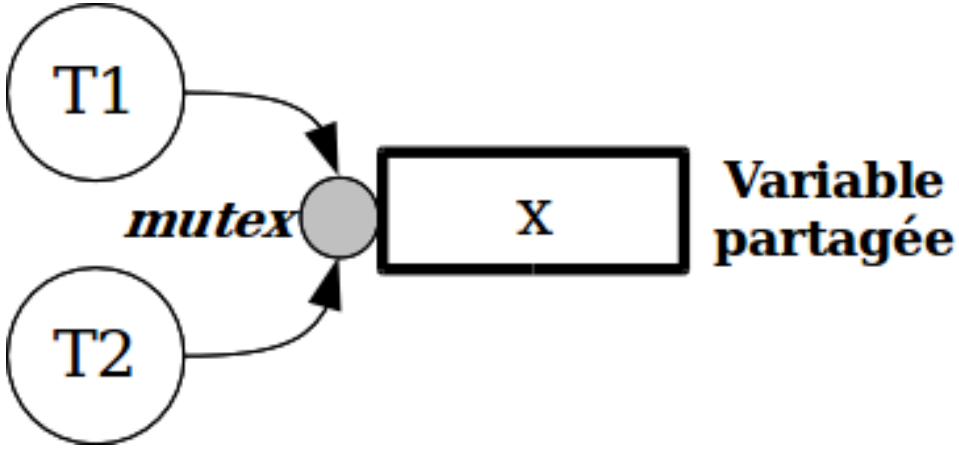
Frères des Ecoles Chrétiennes
Saint-Jean-Baptiste de la Salle - Avignon

Mécanisme de synchronisation : le mutex

Rappel : La cohérence des données ou des ressources partagées entre les processus légers est maintenue par des mécanismes de synchronisation.

Il existe deux principaux mécanismes de synchronisation de données pour les *threads* : le **mutex** et le sémaphore.

- Un **mutex** (**verrou d'exclusion mutuelle**) possède deux états : verrouillé ou non verrouillé.
- Trois opérations sont associées à un mutex : `lock` pour verrouiller le mutex, `unlock` pour le déverrouiller et `trylock` (équivalent à `lock`, mais qui en cas d'échec ne bloque pas le *thread*).



Mécanisme de synchronisation : le sémaphore

Il existe deux principaux mécanismes de synchronisation de données pour les *threads* : le mutex et le **sémaphore**.

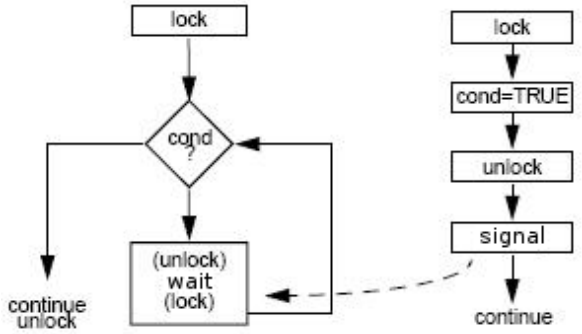
- Un **sémaphore** est un **mécanisme empêchant deux processus ou plus d'accéder simultanément à une ressource partagée**.
- Un **sémaphore général** peut avoir un très grand nombre d'états car il s'agit d'**un compteur** dont la valeur initiale peut être assimilée au nombre de ressources disponibles. **Un sémaphore ne peut jamais devenir négatif**.
- Un **sémaphore binaire**, comme un mutex, n'a que **deux états** : 0=verrouillé (ou occupé) ou 1=déverrouillé (ou libre).
- Un **sémaphore bloquant** est un sémpahore de synchronisation qui est initialisé à 0=verrouillé (ou occupé).
- Trois opérations sont associées à un sémaphore : `Init` pour initialiser la valeur du sémaphore, `P` pour l'acquisition (*Proberen*, tester ou P(uis-je)) et `V` pour la libération (*Verhogen*, incrémenter ou V(as-y)).

Mécanisme de synchronisation : la variable condition

La coordination de l'éxécution entre processus légers est maintenue par des mécanismes de synchronisation.

Il existe deux principaux mécanismes de synchronisation de processus pour les *threads* : la **variable condition** et la barrière.

- Les **variables conditions** permettent de suspendre un fil d'exécution (*thread*) tant que des données partagées n'ont pas atteint un certain état.
- Deux opérations sont disponibles : `wait`, qui bloque le processus léger tant que la condition est fausse et `signal` qui prévient les *threads* bloqués que la condition est vraie.

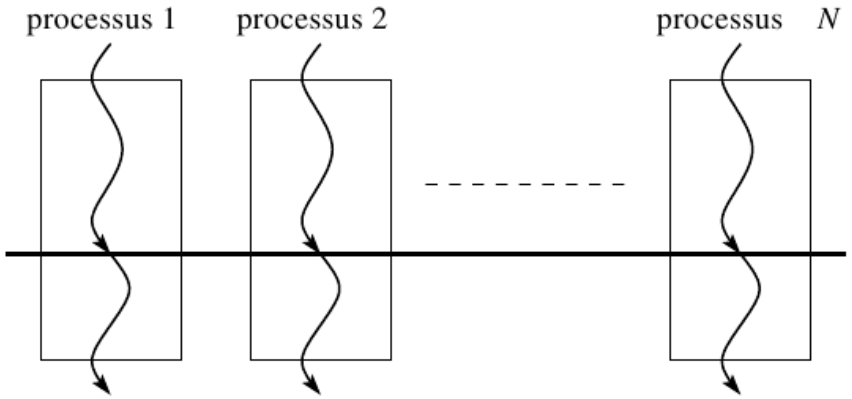


Mécanisme de synchronisation : la barrière

La coordination de l'éxécution entre processus légers est maintenue par des mécanismes de synchronisation.

Il existe deux principaux mécanismes de synchronisation de processus pour les *threads* : la variable condition et la **barrière**.

- Une **barrière de synchronisation** (ou mécanisme de rendez-vous) permet de **garantir qu'un certain nombre de tâches ait passé un point spécifique**. Ainsi, chaque tâche qui arrivera sur cette barrière devra attendre jusqu'à ce que le nombre spécifié de tâches soient arrivées à cette barrière.



Quelques problèmes connus

Les mécanismes de synchronisation peuvent conduire aux problèmes suivants :

- **Interblocage** (*deadlocks*) : Le phénomène d'interblocage est le problème le plus courant. L'interblocage se produit lorsque deux *threads* concurrents s'attendent mutuellement. Les *threads* bloqués dans cet état le sont définitivement.

```
TâcheA :
  Obtenir M1
  Obtenir M2
  Action nécessitant les deux verrous
  Rendre M2
  Rendre M1
```

```
TâcheB :
  Obtenir M2
  Obtenir M1
  Action nécessitant les deux verrous
  Rendre M1
  Rendre M2
```

- **Famine** (*starvation*) : Un processus léger ne pouvant jamais accéder à un verrou se trouve dans une situation de famine. Cette situation se produit, lorsqu'un processus léger, prêt à être exécuté, est toujours devancé par un autre processus léger plus prioritaire.
- **Endormissement** (*dormancy*) : cas d'un processus léger suspendu qui n'est jamais réveillé.

Synthèse

- Section critique : section de code où jamais plus d'une tâche ne peut être active
- Exclusion mutuelle : éviter que des ressources partagées ne soient utilisées en même temps par plusieurs tâches
- Chien de garde (*watchdog*) : tâche de fond assurant la protection contre le blocage de tâches
- Mutex : technique permettant de gérer un accès exclusif à des ressources partagées
- Sémaphore : variable compteur permettant de restreindre l'accès à des ressources partagées

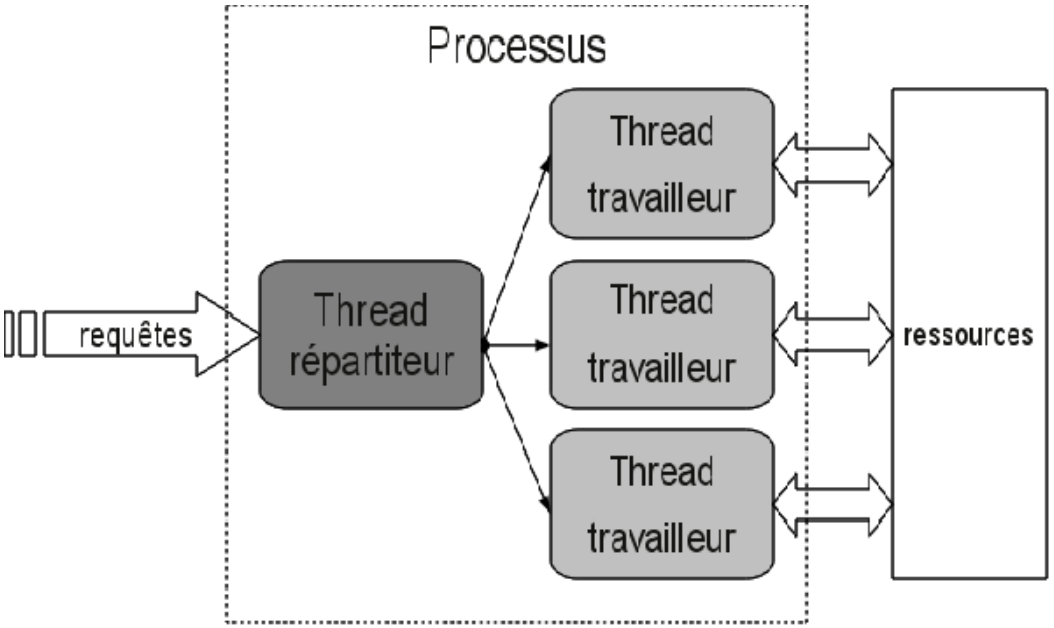
Constat

- Chaque programme comportant des processus légers est différent. Cependant, certains modèles communs sont apparus.
- Ces **modèles** permettent de **définir comment une application attribue une activité à chaque processus léger et comment ces processus légers communiquent entre eux**.
- On distingue généralement 3 modèles :
  - **Modèle répartiteur/travailleurs ou maître/esclaves**
  - **Modèle en groupe**
  - **Modèle en pipeline**

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

Modèle répartiteur/travailleurs

Un processus léger, appelé le répartiteur (ou le maître), reçoit des requêtes pour tout le programme. En fonction de la requête reçue, le répartiteur attribue l'activité à un ou plusieurs processus légers travailleurs (ou esclaves).



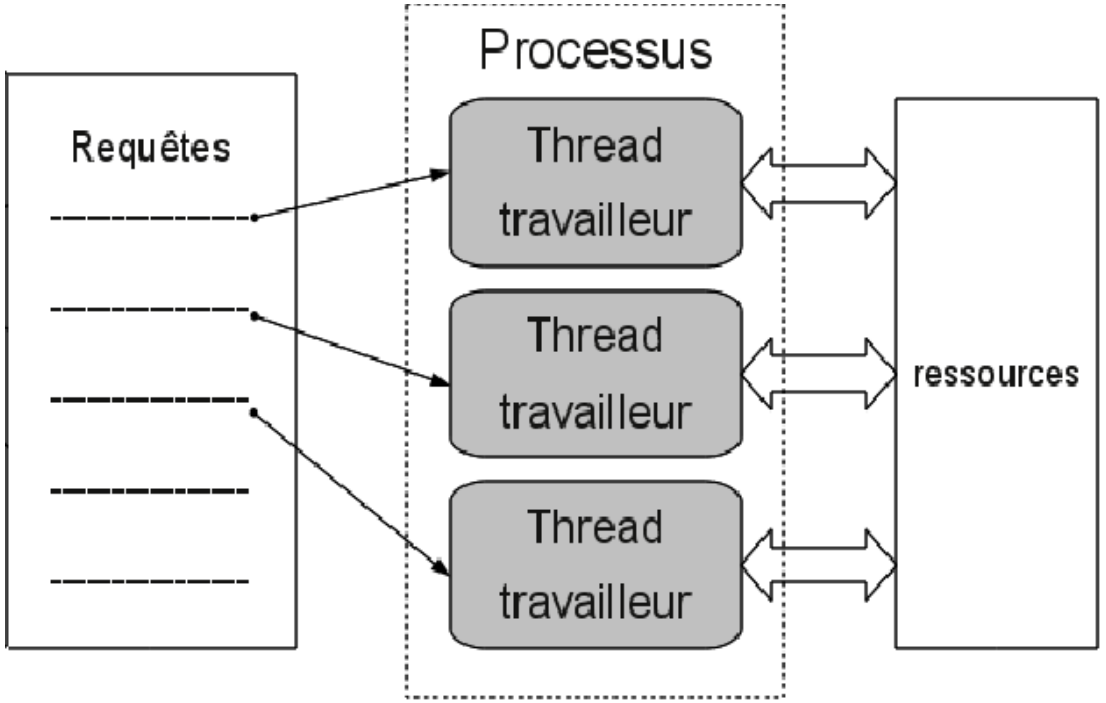
Le modèle répartiteur/travailleurs convient aux serveurs de bases de données, serveurs de fichiers, gestionnaires de fenêtres (*window managers*) et équivalents ...

Modèle répartiteur/travailleurs (suite)

- Les processus légers travailleurs peuvent être créés dynamiquement lors de l'arrivée d'une requête.
- Une autre variante est la création anticipée (au départ) de tous les processus travailleurs (un processus léger par type de requête).
- La création d'un processus intermédiaire (*thread pool*) s'occupant de la gestion des travailleurs peut être aussi envisagé. Ainsi, le répartiteur s'occupe principalement de la réception des requêtes et signale au processus intermédiaire le traitement à réaliser.

Modèle en groupe

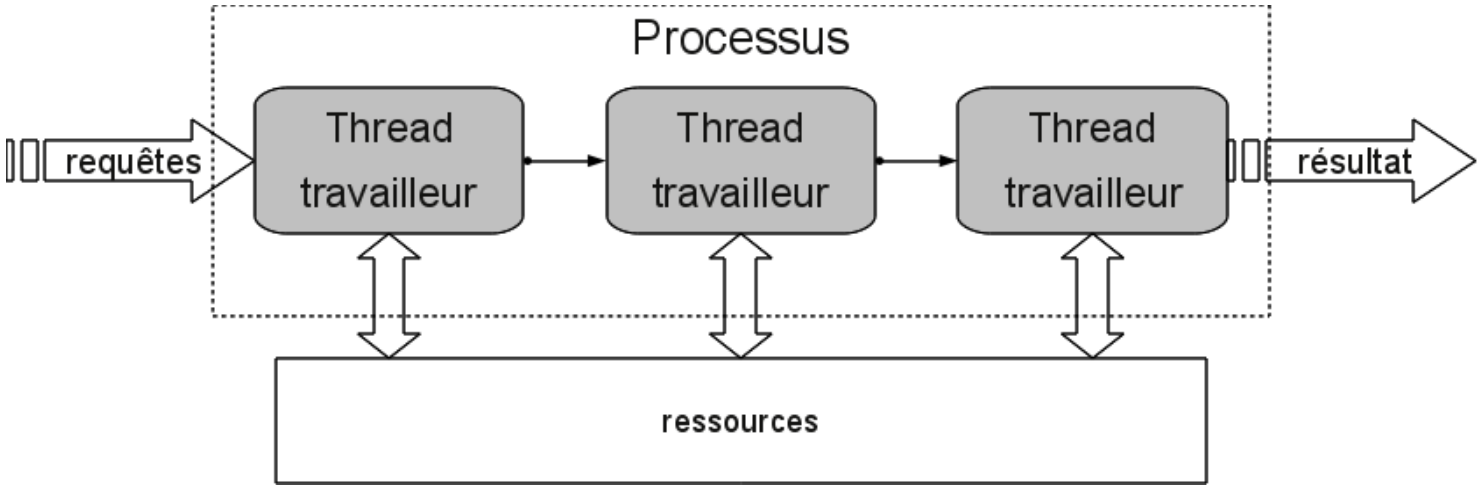
Chaque processus léger réalise les traitements concurremment sans être piloté par un répartiteur (les traitements à effectuer sont déjà connus). Chaque processus léger traite ses propres requêtes (mais il peut être spécialisé pour un certain type de travail).



Le modèle en groupe convient aux applications ayant des traitements définis à réaliser. Par exemple un moteur de recherche dans une base de données.

Modèle en pipeline

L'exécution d'une requête est réalisée par plusieurs processus légers exécutant une partie de la requête en série. Les traitements sont effectués par étape du premier processus léger au dernier. Le premier processus engendre des données qu'il passe au suivant.



Le modèle en pipeline est utilisé pour les traitements d'images, traitements de textes, mais aussi pour toutes les applications pouvant être décomposées en étapes.

La réentrance

- Une **fonction est réentrante** dès lors qu'**elle peut être appelée avec des paramètres différents simultanément par plusieurs *threads***.

- Par extension, une **classe est dite réentrante** si **ses fonctions membres (méthodes) peuvent être appelées de manière sûre par plusieurs *threads* (ou par le même *thread*) simultanément sur différentes instances**.

La réentrance (suite)

- Les classes C++ sont souvent réentrantes, simplement parce qu'elles accèdent uniquement à leurs propres données membres (attributs). Tout *thread* peut appeler une méthode sur une instance d'une classe réentrante, tant qu'aucun autre *thread* n'appelle une méthode de la même instance de la classe en même temps.

- Une **classe réentrante** est une **classe dont toutes les fonctions ne s'appuient que sur l'état de l'objet et des paramètres fournis**. Elles ne dépendent dans leur déroulement ou dans leur retour ni de variables statiques de la classe, ni de variables statiques locales, ni de variables globales et ne font appels qu'à des classes réentrantes ou des fonctions libres réentrantes.

Frères des Ecoles Chrétiennes
Saint-Jean-Baptiste de la Salle - Avignon

Thread-safe

- Une fonction est *thread-safe* si elle peut être appelée correctement par plusieurs *threads* distincts en même temps avec les mêmes paramètres. Grossièrement, une fonction *thread-safe* est une fonction qui se protège, si nécessaire, des problèmes synchronisation pour les accès concurrents (*race-condition*).
- Par conséquent, une fonction *thread-safe* est toujours réentrante, mais une fonction réentrante n'est pas toujours *thread-safe*.
- Par extension, une classe est *thread-safe* si toutes ses fonctions membres (méthodes) peuvent être appelées par plusieurs *threads* distincts simultanément sur la même instance.
- Une **classe est *thread-safe*** dès lors qu'elle **utilise des moyens de synchronisation (verrous) pour l'accès aux ressources partagées**.

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

Les threads en C++

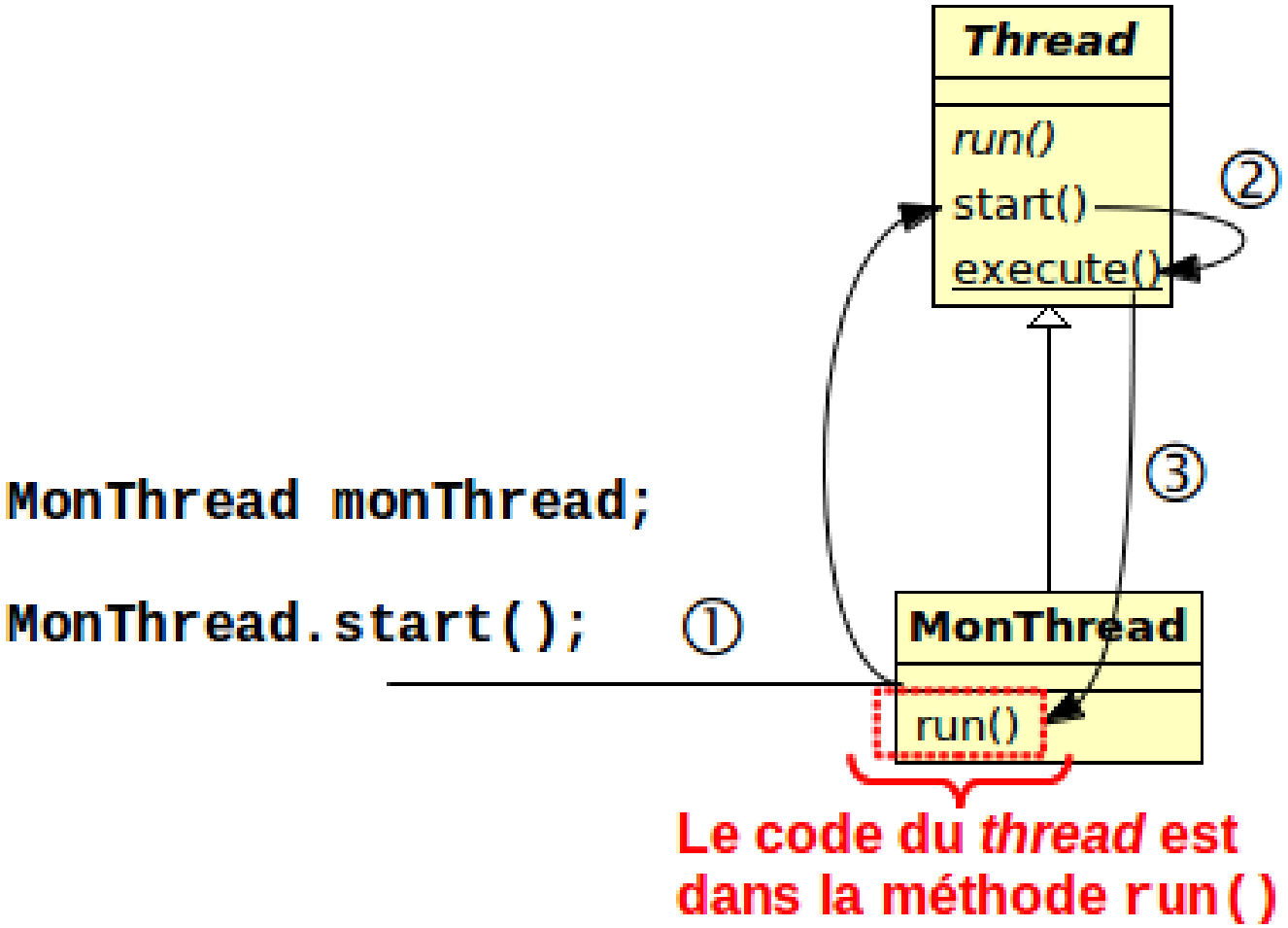
- Objectif : disposer d'une classe `Thread`.
- Rappel : un fil d'exécution (*thread*) est une fonction.
- Conclusion : une classe `Thread` doit donc posséder une méthode qui sera le fil d'exécution (*thread*).
- Problème :
  - Les APIs de base sont écrites en C et fournissent une fonction de création de *thread* (`pthread_create` et `CreateThread`) qui attend l'**adresse d'une fonction**.
  - Les **méthodes d'une classe** n'ont pas d'adresse connue au moment de la compilation.

Les threads en C++ (suite)

En C++, le principe est de créer une classe **abstraite** `Thread` qui contiendra :

- une méthode virtuelle pure `run()` dans laquelle on placera le code du *thread* une fois qu'on aura dérivé la classe mère. Cette méthode pourra accéder aux attributs et méthodes de sa classe.
- une méthode statique `execute()` qui exécutera `run()`
- une méthode `start()` qui créera et lancera le *thread* (appel à `pthread_create()`) en lui passant l'adresse de la méthode statique `execute()`

Les threads en C++ (fin)

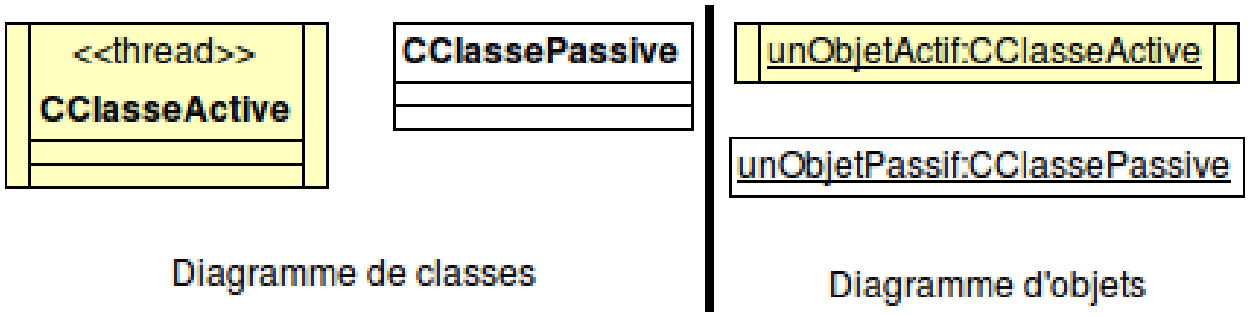


Notion de vue

- En UML, l'architecture logicielle d'une application est composée de **5 vues** : **cas d'utilisations, conception, processus, implémentation et déploiement**. Une vue est une description simplifiée d'un système observé d'un point de vue particulier.
- La **vue des processus** précisera les *threads* et les processus qui forment les mécanismes de concurrence et de synchronisation du système.
- On mettra l'accent sur les **classes actives** qui représentent les *threads* et les processus.
- La **vue des processus** montrera : la **décomposition du système en terme de processus (tâches), les interactions entre les processus (leur communication) et la synchronisation et la communication des activités parallèles (*threads*)**.

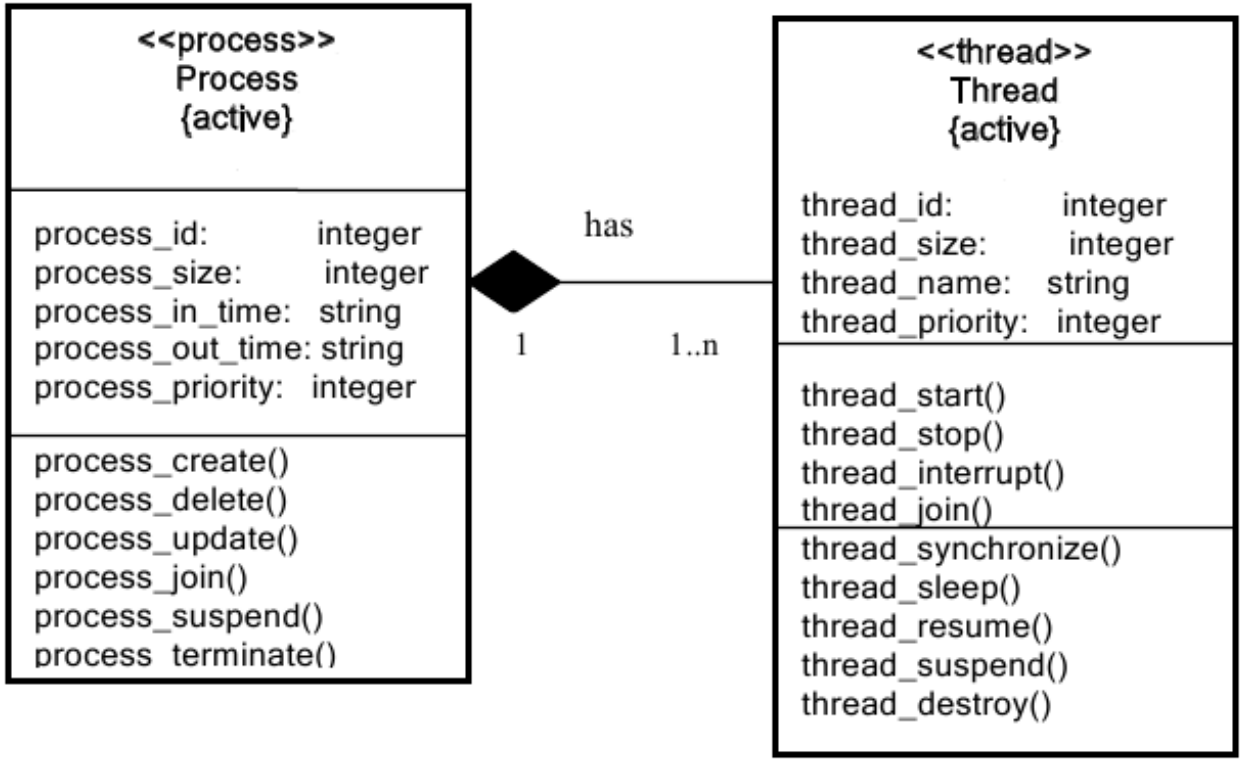
Classe active et objet actif

- UML fournit un repère visuel (bord en trait épais ou double trait) qui permet de distinguer les éléments actifs (processus ou *thread*) des éléments passifs.
- Une **instance d'une classe active** sera nommée **object actif**.
- Chaque processus ou *thread* au sein d'un système définit alors un **flot de contrôle distinct** (c'est le fil d'exécution).



Stéréotypes

- UML définit deux stéréotypes standards qui s'appliquent aux classes actives :
  - « `process` » : spécifie un flot « lourd » qui peut s'exécuter en concurrence avec d'autres processus.
  - « `thread` » : spécifie un flot « léger » qui peut s'exécuter en concurrence avec d'autres *threads* à l'intérieur d'un même processus.



Diagrammes

- Le découpage de l'application en *threads* (ou processus) apparaît lorsqu'on établit :
  - les diagrammes d'activités : activités concurrentes et/ou
  - les diagrammes d'états : états concurrents et/ou
  - les diagrammes de séquences : exécutions concurrentes de plusieurs objets

Besoins

- Le besoin est toujours la **parallélisation** :
  - le système réalise plusieurs activités en même temps et/ou
  - un objet prend plusieurs états en même temps et/ou
  - une utilisation du système réalise plusieurs exécutions en même temps

- Le besoin de *threads* apparaît clairement pendant la **phase d'analyse** lorsqu'**on distingue ce qui se produit de manière séquentielle de ce qui se produit de manière parallèle**.

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon