INSA
CENTRE VAL DE LOIRE

POO Avancée : Java

Threads : Precessus légers

Salwa SOUAF

Outline

Histoire

La notion de processus est apparue lorsque les ordinateurs sont devenus assez puissants pour permettre l'exécution de plusieurs programmes de façon concurrente. En effet lorsque plusieurs programmes s'exécutent simultanément, il est important d'assurer que la défaillance d'un seul programme n'entraîne pas la défaillance de tout le système. Dans cette optique, la notion de processus a permis l'isolement de chaque programme.
Dans un même temps, les modèles de programmation évoluaient et très vite le désir d'exprimer de la concurrence au sein d'un même programme s'est fait ressentir. La première réponse à cette demande a été l'utilisation de plusieurs processus associés à différents moyens de communication (mémoire partagée, sémaphores, pipes, et puis socket). Cette solution présentait plusieurs inconvénients majeurs. Le premier est que l'ensemble des moyens de communication demande un passage en mode noyau (ce qui est coûteux en temps). Le second inconvénient est que la création d'un processus est relativement longue. Les autres problèmes étaient le nombre limité de processus au sein des systèmes d'exploitation et une utilisation mémoire non négligeable.
Il fallait définir une nouvelle entité. La solution proposée a été d'associer plusieurs exécutions concurrentes à un même processus. Pour cela on a donc défini les informations minimales nécessaires permettant d'avoir plusieurs fils d'exécution au sein d'un même processus, les threads (ou processus légers) étaient nés.

Outline

Processus lourds et légers

Processus lourds

Definition

Les processus (ou processus lourds) sont en général plus proches du système d'exploitation. Ils s'agit par exemple d'un processus unix. Ce type de processus obtient un espace mémoire dédié, une pile, un nouveau jeu de variable, toutes ces données restant inaccessibles d'un processus à l'autre. L'utilisation du fork() de POSIX est un exemple de processus lourd. Le seul cas de partage entre ces processus est le code et les accès aux descripteurs de fichiers. Par contre, chaque processus obtient une pile, l'état des registres du CPU et doit redéfinir le masquage pour l'interception des signaux. Cependant, utiliser des processus lourds communiquants est difficile à mettre en oeuvre, surtout à cause de l'accès à la mémoire partagée. Pour partager les données, on préfère utiliser les processus légers.

Il existe aussi la possibilité de créer des processus lourds, depuis la machine virtuelle. Cela revient à un fork() suivit d'un execXX en C. Les classes à utiliser sont les classes Runtime et Process, par exemple:

```
Runtime.getruntime().exec("cmd");
```

Definition

Un processus léger est un flot d'exécution partageant l'intégralité de son espace d'adressage avec d'autres processus légers. Il est en général géré à un plus haut niveau, par rapport au système d'exploitation. Les différents processus légers peuvent s'exécuter alors dans un même processus lourd, mais de manière concurrente.

Processus lourds et légers

Processus légers : Création en java

En Java, ces processus légers sont appelés **thread**. En pratique, on peut utiliser la classe Thread et Runnable, par exemple:

```
public class MyThread extends Thread {
public void run() { ... }

public class MyRunnable implements Runnable
public void run() { ... }
```

```
MyThread t = new MyThread();
t.start();

Thread t = new Thread(new MyRunnable);
t.start();
```

Un thread est actif lorsqu'il est en cours d'exécution de la méthode *run()*. Avant cet état, il est existant (ce qui correspond au new...). Il peut être éventuellement bloqué ou interrompu (*interrupt()*). Lorsque le thread atteint l'accolade fermante du *run()*, il a atteint sa fin d'exécution et peut être collecté par le garbage collector, à condition que l'utilisateur redonne la référance correspondante. En cours d'exécution, il peut volontairement laisser la main à d'autres processus en utilisant les méthodes *sleep()* ou *pause()*.

Processus lourds et légers

Processus lourds Vs. Processus légers

Processus	Thread
Plus proches du système d'exploitation.	Exécuté dans l'environnement d'un process.
Mémoire dédiée (consommation de ressource).	Espace mémoire partagée entre threads ayant le même parent.
Isolation de d'autres processus	Partage inter-thread (Données, Ressources…)
Création/Arrêt plus lents	Prend moins du temps

Outline

Schedulers

Plusieurs processus lourds peuvent s'exécuter en concurrence sur un ou plusieurs processeurs. Le système d'exploitation donne la main alternativement à chaque processus lourd. On parle de scheduler pour exprimer le fait qu'on doit ordonnancer la prise du fil d'exécution par chaque processus lourd, alternativement. A l'intérieur d'un même processus lourd, le scheduler peut avoir des comportements différents pour la politique de prise du fil d'exécution des processus légers. Le choix de cette politique peut poser des problèmes de concurrence (dead lock, données corrompues, ...) puisque le scheduler peut par exemple décider arbitrairement d'arrêter l'exécution d'un thread pour donner la main à un autre. Les systèmes de gestion de threads se subdivisent en deux catégories :

- les schedulers coopératifs
- les schedulers préemptifs

Schedulers

Scheduler coopératif

Dans ces systèmes, les threads s'exécutent jusqu'à ce qu'ils décident de relâcher explicitement le processeur (instruction yield) pour laisser un autre thread s'exécuter. En d'autres termes, l'ordonnancement des threads doit être réalisé par le programmeur.

Avantages: Le principal avantage de ce modèle est la simplicité de la gestion des données partagées. En effet, puisque les threads rendent explicitement la main, le problème des données partagées pouvant se trouver dans un état incohérent ne se pose pas.

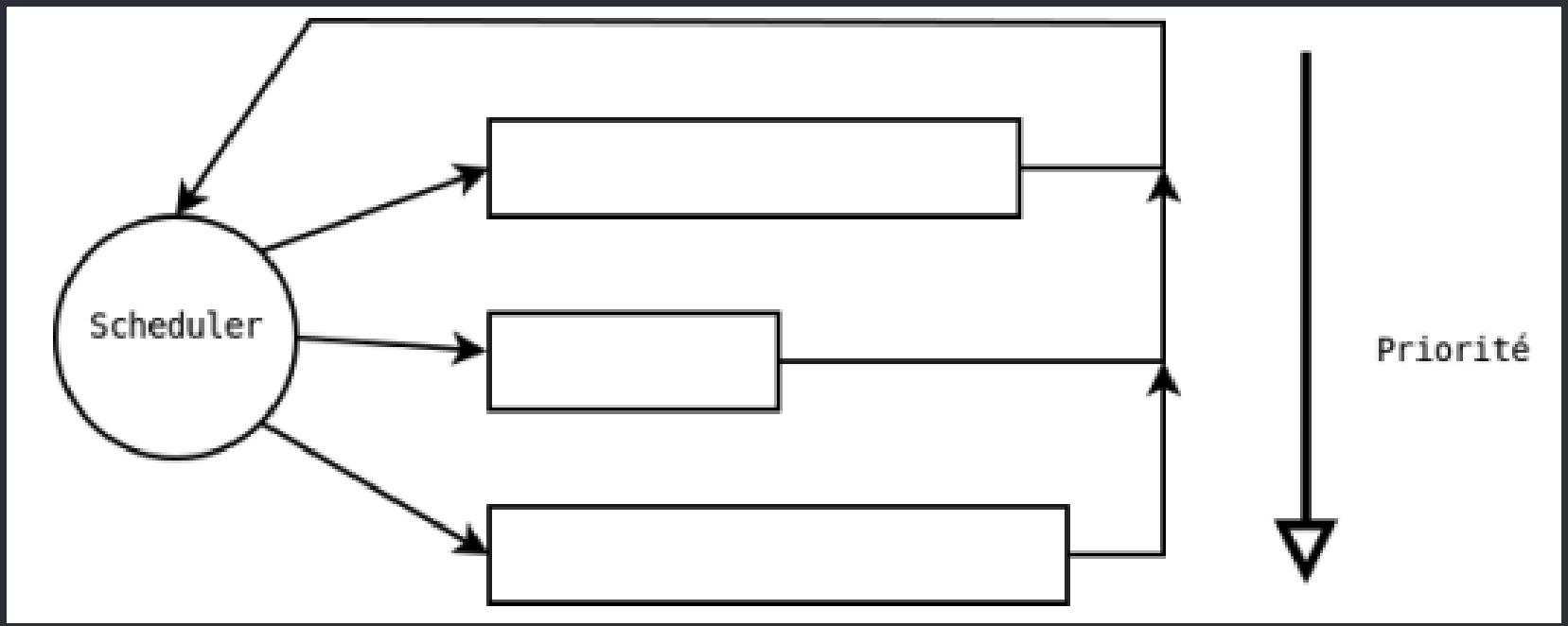
Inconvénients: Le principal désavantage de telles implémentations est l'énorme difficulté voire l'impossibilité de le faire fonctionner sur plusieurs processeurs. On notera que la gestion de l'ordonnancement n'apporte pas que des avantages et pose notamment le problème de choisir intelligemment les endroits où réaliser un yield. Enfin, de telles implémentations demandent impérativement l'utilisation d'entrées/sorties non bloquantes (pas toujours disponibles) pour éviter que l'ensemble du processus soit bloqué.

Le mode coopératif exige de rendre explicitement le fil d'exécution du scheduler. Le scheduler est donc passif: il attend qu'un processus rende la main pour ensuite choisir le prochain processus.

Schedulers

Scheduler coopératif : FCFS

L'algorithme FCFS (First-Come-First-Served) (premier arrivé, premier servi) est le plus simple des algorithmes d'ordonnancement. Le premier processus qui entre dans la file est le premier à bénéficier du processeur. Ce système est par contre très peu rentable. Les processus longs sont favorisés par rapport aux processus courts. Le type de gestion est non-préemptif, car rien n'est prévu par le système d'exploitation pour déloger le processus du cpu, ou favoriser un processus dans la file. Sur la figure ci-dessous, le scheduler choisit tout d'abord par priorité décroissante le processus à exécuter. Puis, à priorité égal, il donne la main séquentiellement à chaque processus.



Schedulers

Scheduler préemptif

Dans ces systèmes, il existe un thread particulier ou bien une partie du système d'exploitation (que l'on nomme scheduler ou ordonnanceur) qui est chargé de décider quel thread doit s'exécuter. Son rôle est de répartir au mieux (en fonction des priorités par exemple) les ressources du système. De ce fait le programmeur n'a pas à s'occuper de l'ordonnancement.

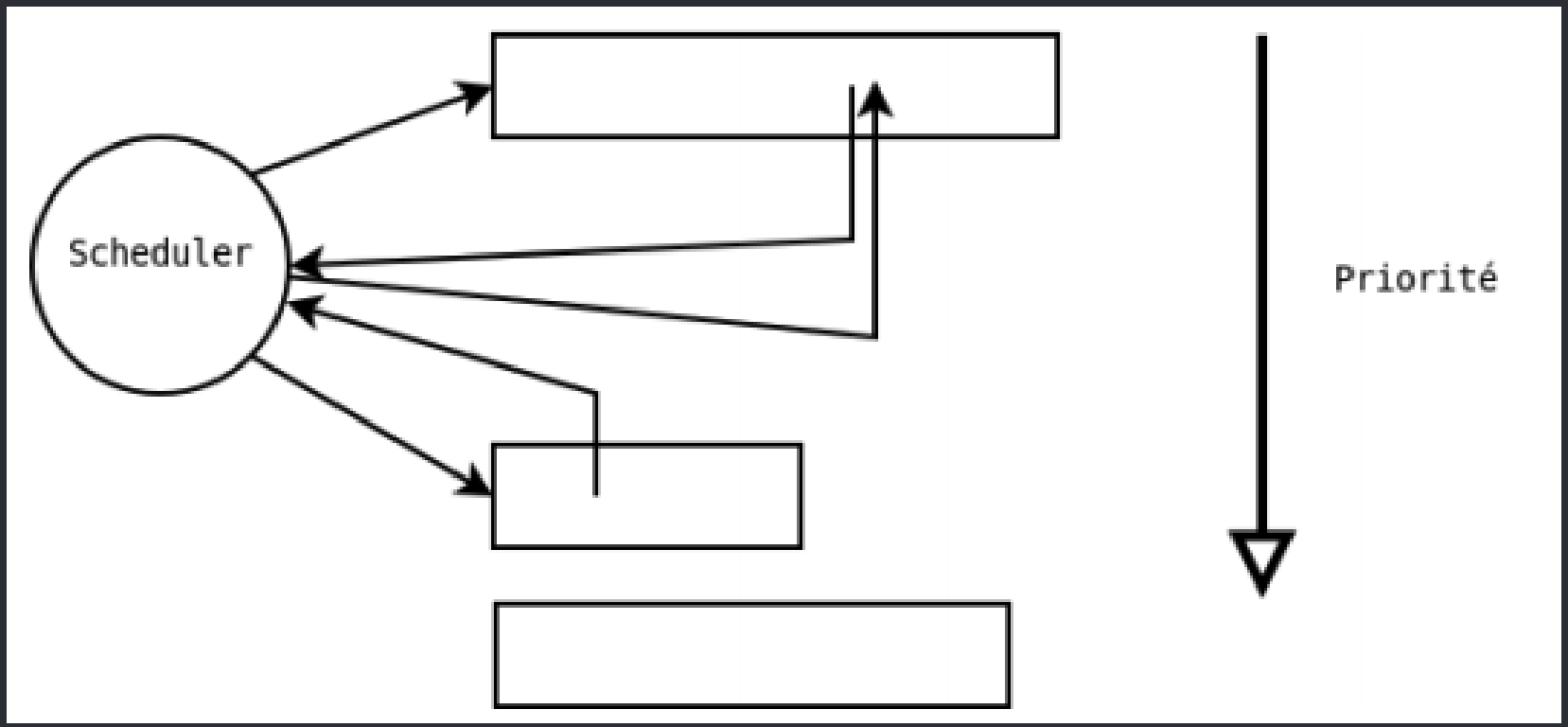
Avantages: Les avantages de ce modèle sont multiples. La charge de définir un ordonnancement étant assurée par une entité externe la rend dynamique. Ceci a pour conséquence de la rendre beaucoup plus souple. De plus, l'utilisation d'entrées/sorties bloquantes ne pose plus aucun problème.

Inconvénients: Le principal désavantage de ce modèle est la difficulté que pose la gestion des données partagées. En effet, puisque l'ordonnancement est externalisé aucune supposition ne peut être faite sur l'ordre d'exécution des threads et sur les interruptions du scheduler qui peuvent survenir à tout moment.

Schedulers

Scheduler préemptif

Pour un scheduler préemptif, le scheduler peut prendre la main pendant l'exécution d'un thread et donner le fil d'exécution à un thread en cours d'exécution. Sur la figure ci-dessous on montre le changement du fil d'exécution en plein milieu des processus.



Schedulers

Scheduler préemptif : round-robin

L'algorithme round-robin est spécialement adapté aux systèmes en temps partagé. On définit un quantum de temps (time quantum) d'utilisation de l'unité centrale. La file d'attente des processus éligibles est vue comme une queue circulaire (fifo circulaire). Tout nouveau processus est placé à la fin de la liste.
Deux choses l'une, soit le processus actif rend le fil d'exécution avant la fin de sa tranche de temps (pour cause d'entrée/sortie ou volontairement) soit il est préempté, et dans les deux cas placé en fin de liste. Un processus obtiendra le processeur au bout de (n -1)T secondes au plus n nombre de processus et T longueur du quantum de temps): la famine est donc assurément évitée.
Remarquons que si le quantum de temps est trop grand, round-robin devient équivalent à FCFS. De l'autre coté si le quantum de temps est très court, nous avons théoriquement un processeur n fois moins rapide pour chaque processus (n nombre de processus). Malheureusement si le quantum de temps est court, le nombre de changements de contexte dûs à la préemption grandit, d'où une diminution du taux utile, d'où un processeur virtuel très lent. Une règle empirique est d'utiliser un quantum de temps tel que 80% des processus interrompent naturellement leur utilisation de l'unité centrale avant l'expiration du quantum de temps.

Schedulers

Choix d'ordonnancement dans Java

Le choix a été de ne pas faire de choix. En effet, contraint par l'objectif d'être multi-plateformes, le langage Java a décidé de ne pas faire de spécifications trop contraignantes sur l'implémentation et le fonctionnement des threads. Java ne définit que quelques règles sur la nature des threads et la façon de les ordonnancer. Toutefois, La spécification Java des threads permet une implémentation préemptive, ce qui permet d'effectuer un mapping des threads sur les threads du système d'exploitation hôte. Dans ce cas le scheduler peut-être par exemple du type round-robin.

La conséquence de ceci est que si l'on désire être réellement portable au sens de la spécification Java, il faut que les programmes puissent fonctionner (c'est à dire avoir un fonctionnement identique) quelles que soient les spécificités de l'implémentation. Ceci demande donc de faire des appels à la méthode yield, dans le cas d'une implémentation coopérative, sans pour autant profiter des avantages de cette implémentation (une meilleure maîtrise des données partagées).

Toutefois, la majorité des machines virtuelle que l'on trouve aujourd'hui sur les ordinateurs de bureau utilisent des threads préemptifs, à la différence de ce qui existe dans les Cards et dans certains systèmes embarqués. On peut donc raisonnablement partir du principe que, pour la programmation d'applications destinées au desktop, on considère que l'implémentation est préemptive.

Schedulers

Priorité et interruptions

Il est possible de changer la priorité d'un thread afin qu'il ait une priorité particulière pour accéder au processeur. Le thread de plus forte priorité accède plus souvent au processeur. Par défaut, un thread a la priorité de son père. Pour changer la priorité d'un thread on utilise la méthode suivante: setPriority (int prio).
Seule la possibilité d'interrompre un processus a été gardé dans Java 1.5. Les méthodes permettant de stopper ou de suspendre un thread sont dépréciées, pour éviter de laisser des objets en cours de modification dans un état "non-cohérent".
La méthode interrupt() positionne un statut d'interruption et peut lever des exceptions si le thread était en pause ou en cours d'utilisation d'entrées/sorties.

Schedulers

Fil d'exécution et atomicité - 1

Les instructions des différentes threads peuvent être exécutées à tour de rôle, dans un ordre qui dépend du choix du scheduler. Le scheduler donne le fil d'exécution du processeur à un thread pendant un certain temps, puis reprend la main, pour réévaluer le prochain thread qui prendra de nouveau le fil d'exécution. Dans le cas où plusieurs processeurs sont disponibles, le scheduler s'occupe de répartir n threads sur m processeurs.
On parle d'atomicité d'une instruction lorsque le scheduler ne peut pas prendre la main au cours de l'exécution de cette instruction. En Java, l'atomicité n'est garantie que sur les affectations (sauf pour les double et long). Sinon, le scheduler peut interrompre l'exécution d'une instruction et lui redonner la main à posteriori pour terminer son accomplissement. Les instructions atomiques classiques sont par exemple l'ouverture d'un bloc synchronizé, la prise d'un jeton d'un sémaphore, ou l'acquisition d'un lock.

Une erreur classique en programmation est de croire que le **if** est atomique ce qui peut conduire à des bugs liés à la concurrences de threads:

```
boolean array_access = false;
if (!array_access)
{
   array_access = true;
   ... // do the job
}
array_access = false;
```

Schedulers

Rendez-vous et pauses

Un thread peut attendre qu'un autre thread se termine. On utilise la méthode *join()* pour attendre la terminaison d'un autre fil d'exécution concurrent avant de continuer son propre fil d'exécution.

```
MyThread m1 = new MyThread();
MyThread m2 = new MyThread();
m1.start();
m2.start();
m1.join();
m2.join();
```

En cas d'attente, le thread ne consomme pas (ou peu) de temps CPU. On peut soi-même demander explicitement l'endormissement d'un thread pendant un temps donné en utilisant *Thread.sleep(int time)*, ce qui endort le thread courant (donc soi-même puisque c'est notre "moi-même" qui exécute cet appel). On peut aussi rendre explicitement la main au scheduler avec *yield()*, ce qui peut-être particulièrement utile lorsqu'on programme par exemple des interfaces graphiques.

```
// Affichage graphique
...
// Fin affichage graphique
Thread.yield();
```

Outline

Gestion de taches

Certaines applications nécessitent de traiter des ensembles de taches sur différents threads, souvent à des fins de performances. Il peut aussi s'agir de découper un calcul en plusieurs sous calculs, notamment si l'on projette ensuite de réaliser ce calcul de manière distribuée. Un certain nombre de classes de haut niveau sont dédiées à la gestion de ces taches et épargnent ce travail de gestion à l'aide des primitives de base de la concurrence.
L'interface *Future* permet d'attendre ou de manipuler une tache encore en cours d'exécution et dont on attend le résultat. Les principales méthodes utilisées sont *get()*, *cancel()*, *isCancelled()*, *isDone()*. La méthode *get()* retourne le résultat de la tache et reste bloquante si le résultat n'est pas encore calculé. Les autres méthodes permettent de contrôler la tache et éventuellement d'annuler son exécution.

Gestion de taches

Pool de threads

Pour éviter de devoir créer un thread à chaque fois qu'une nouvelle tache doit être exécutée, par exemple dans le cas classique d'un serveur web. On utilise dans ce cas un pool de thread, qui n'est rien d'autre qu'une collection de threads qui se nourrit de taches disponibles dans une queue. L'interface *Executor* permet de spécifier les méthodes classiques d'une tache et l'on peut ensuite décider de différentes polique d'exécutions des taches en agissant sur des objets implémentant *Executor*.

Gestion de taches

Pool de threads : politiques

Les politiques proposées dans Java 1.5 sont les suivantes:
Excecutors.newCachedThreadPool(): (unbounded thread pool, with automatic thread reclamation) un pool de thread de taille non limité, réutilisant les threads déjà créés et en créant de nouveau au besoin. Après 60 secondes d'inactivité, le thread est détruit.
Excecutors.newFixedThreadPool(int n): (fixed size thread pool) un pool de thread de taille n. Si un thread se termine prématurément, il est automatiquement remplacé.
Excecutors.newSingleThreadExecutor(): (single background thread) crée un seul thread ce qui garantit la séquentialité des taches mises dans la queue de taille non bornée.
Si l'on utilise des pools de threads de taille bornée, on peut se demander quel est la taille optimale de l'ensemble. La loi d'Amdahl donne une bonne idée du dimensionnement optimal pour une utilisation maximale d'un système multiprocesseur. Si WT est le temps moyen d'attente d'une tache et ST son temps moyen d'exécution, avec N processeurs la loi d'Amdahl propose un ensemble de threads de taille N/(1+WT/ST) Ceci ne tient évidemment pas compte des aléas temporel dus par exemple aux entrées/sorties.

Rendez-vous de threads

Lorsque l'on souhaite synchroniser plusieurs threads entre eux, c'est à dire faire en sorte que les threads s'attendent les uns les autres (par exemple après un calcul distribué), on peut utiliser la classe CyclicBarrier qui bloquent les threads à une barrière (l'instruction CyclicBarrier.await() et les débloquent quand tous les threads y sont rendus. Cette barrière est cyclique car elle peut-être réutilisée pour une prochaine utilisation (elle est réinitialisée).
La classe CountdownLatch est très similaire à CyclicBarrier mais elle est plus adaptée à la coordination d'un groupe de threads ayant à traiter un problème divisé en sous problèmes. Chaque thread décrémente un compteur en appellant countDown() après avoir traité une des taches, puis est bloquée à l'appel de CyclicBarrier.await(). Le débloquage des threads ne se fait que lorsque le compteur atteint 0.
Enfin, la classe Exchanger permet de réaliser des échanges de données entre deux threads coopératifs. Il s'agit d'une barrière cyclique de 2 éléments avec la possibilité supplémentaire de s'échanger des données lorsqu'ils atteignent la barrière.