Concurrence

Sébastien Speierer

22 juin 2014

Table des matières

0.1 Un processus

La programmation concurrente permet d'exploiter la possibilité qu'ont le processeur et les périphérique de travailler en parallèle. Un **processus** peut être comparé à un programme séquentiel : il exécute des instructions, l'une aprs l'autre. Il existe différents modes de partage de processeur :
- **Pseudo-parallélisme** : à tout instant seul un processus est en exécution et la commutation de processus s'effectue **à l'insu** des processus. (Géré par le noyau)
- **quasi-parallélisme** : à tout instant seul un processus est en exécution mais la commutation de processus se fait **à la demande** du processus actif. On utilise souvent le terme de **coroutine** comme synonyme de processus quasi parallèle.

Un **noyau** est un allocateur de processeur : il s'occupe de partager le processeur entre les différents processus.

0.1.1 Structure d'un noyau

Un **descripteur de processus** : structure de donnée permettant de retrouver toute l'information propre à un processus (variable, état,contenu des registres). Ces informations constituent le **contexte d'exécution** d'un processus.

L'**horloge** produit des interruptions à intervalles réguliers qui provoque un commutation de processus. A chaque interruption de l'horloge, le processus actig perd le processeur au profit d'un autre processus.

0.2 Les Threads JAVA

Un processus est le terme utilisé pour désigner l'entité qu exécute le code. Dans le contexte d'un OS, le terme processus désigne l'entité constituée d'un espace mémoire, de ressources (fichiers, sockets,...), ainsi que d'une ou plusieurs entités exécutant du code. Ces dernières sont alors généralement appelées **threads**. Voilà différentes méthodes d'un thread JAVA :
- **run** : qui spécifie le code exécuté par le thread
- **getId()** et **currentThread()**
- **join()** : bloque le thread courant jusqu'à ce que le thread référencé se termine.
- **sleep(long millis)** : suspend l'exécution du thread pour la durée spécifiée.
- **yield()** : permet au thread courant de rendre le CPU, sans pour autant être bloqué.
- **setDaemon(boolean b)** : set le thread en daemon (ou non). Cette méthode doit être exécuté avant t.start(). De manière générale, un thread englobant d'autres threads ne se termine que lorsque tous les threads "internes" et "user" (pas daemon) se sont terminés.
- **start()** : débute l'exécution de la méthode run.
- **setPriority(int newPriority)** : attribue une priorité au thread.

Voici les 4 états d'un thread JAVA
- **Initial** : état depuis la création jusqu'à t.start()
- **Runnable** : le thread est susceptible d'obtenir le CPU
- **Blocked** : le thread est en attente d'un événement
- **Exiting** : état après la fin de l'exécution de sa méthode run.

Le mot clé **volatile** doit être utilisé pour toute variable partagée entre plusieurs threads.

1 Exclusion mutuelle

1.1 Section critique

Un **ressources critiques** est un objet qui doit être accédé en exclusion mutuelle. On utilise le terme **section critique** pour désigner la partie du code d'un processus dans lequel le processus accède à une ressource critique. Une exclusion mutuelle empêche les processus d'accéder à une section critique si un autre processus y est déjà "dedans". On exige :
– qu'un processus doit pouvoir entrer en section critique si aucun processus ne l'occupe.
– qu'un processus qui désire enter en section critique pourra le faire au bout d'un temps fini.

1.2 Masquage des interruptions

Masquer les interruptions permet de résoudre le problème de l'exclusion mutuelle : plus d'interruptin, donx plus de commutation de processus, donc plus de risque de violer l'exclusion mutuelle. Cette solution n'est acceptable seulement si la section critique est courte (car elle empêche les entrées-sorties).

1.3 Les Verrous

Un verrou est l'outil le plus primitif permettant d'implémenter une section critique sans faire de l'attente active ni masquer les interruptions. Il contient un boolean qui défini sont état (ouvert/fermé) ainsi qu'une liste d'attente. Lorsqu'un processus tente de prendre le verrou, si celui-ci est fermé, le processus est placé en liste d'attente. Une déverrouillé, le verrou débloque un processus qui était en liste d'attente. (On implémente un masquage d'interruption sur ces méthode verrouiller et déverrouiller car elles sont courtes)

1.4 Les Sémaphores

Un sémaphore est un généralisation du verrou, puisque le champ *état* d'un verrou qui ne pouvait prendre que deux valeurs est remplacé ici par un champ n de type entier. Un sémaphore ne peut être manipulé que grâce à deux primitives **P** et **V** définit ainsi :

Algorithm 1: P

```
Data: var s : sémaphore
s.n := s.n-1 ;
if s.n < 0 then
    bloquer le processus en queue de la liste "s.en-attente" ;
end
```

Algorithm 2: V

```
Data: var s : sémaphore
s.n := s.n+1 ;
if s.n <= 0 then
    débloquer le processus en tête de la liste "s.en-attente" ;
end
```

On remarque que dans un sémaphore, la valeur absolue $|s.n|$ indique le nombre de processus bloqués dans la liste d'attente. L'initialisation du champ n d'un sémaphore va dépendre du type de problème à résoudre. L'identificateur **mutex** est un abréviation couramment choisi pour désigner un sémaphore permettant de réaliser l'exclusion mutuelle (init s.n à 1).
Un erreur facilement commise avec les sémaphores consiste à exécuter un appel de la primitive P à l'intérieur d'une section critique. Cette situation risque d'engendrer un **interblocage**.

1.5 Problèmes "Lecteur-Rédacteur"

Dans le problème des lecteurs et des rédacteurs, nous considérons deux classes de processus : les processus lecteurs et le processus rédacteurs. Les processus lecteurs désirent lire des données ; les processus rédacteurs désirent modifier ces données. Un solution correcte de ce problème devra assurer l'exclusion mutuelle entre les rédacteurs (pas d'écriture simultanée pour éviter un risque d'incohérence des données) et l'exclusion mutuelle entre les lecteurs et le rédacteurs (pas de lecture pendant une écriture, pour éviter de lire des données dans un état incohérent). Les lecteurs, en l'absence des rédacteurs, doivent toutefois pouvoir accéder simultanément aux données.
Il y a 3 cas à distinguer :
- **Priorité aux lecteurs** : lecture possible lorsque personne n'écrit. Écriture possible lorsque personne n'écrit, personne le lit et personne n'attend pour lire.
- **Priorité aux rédacteurs** : lecture possible lorsque personne n'écrit et personne n'attend pour écrire. Écriture possible lorsque personne n'écrit et personne ne lit.
- **Même priorité aux lecteurs et aux rédacteurs** : lecture possible lorsque personne n'écrit. Écriture possible lorsque personne ne lit et personne n'écrit.

1.6 Les Événements

La **synchronisation** consiste imposer un ordre sur l'exécution des instructions des processus. L'événement est l'outil le plus primitif permettant de résoudre le problème de synchronisation. C'est un variable qui contient un boolean "survenu" et une liste d'attente. Il peut avoir lieu ou ne pas avoir lieu. Il peut être manipulé grâce aux procédures *attendre*, *déclencher* et *réinitialiser*.

Il est aussi possible d'implémenter une **synchronisation à l'aide de sémaphore**. Il suffit de l'initialiser à 0 comme ça le processus qui tentera d'entrer dans le sémaphore sera mit en attente.

1.7 Les Moniteurs

Au contraire des autre outils présentés, le moniteur est une unité syntaxique ce qui signifie qu'il permet de regrouper des variables ainsi que les procédures agissant sur ces variables. Mais en plus, le moniteur assure l'exclusion mutuelle sur les procédures déclarées dans le moniteur. La synchronisation s'exprime à l'intérieur d'un moniteur grâce à des **signaux**. Un signal peut être manipulé par les procédures **wait** et **send**. L'exécution de s.wait bloque un processus et celui-ci sera alors en attente du signal s. Il sera débloqué par l'exécution de s.send. Plusieurs processus peuvent être en attente du même signal et l'exécution du s.send n'en débloque qu'un seul, à savoir le premier s'étant mis en attente du signal. Notons que lorsqu'un processus se bloque par l'exécution de s.wait, l'exclusion mutuelle sur le moniteur est levée.
Lorsqu'un processus P1 exécute s.send, il est temporairement suspendu et c'est le processus P2 (qui était dans la liste d'attente de s) qui devient le processus actif du moniteur. Ceci garantie que la condition transportée par le signal s est toujours vraie lorsque P2 reprend son exécution. P1 redevient le processus actif dès que P2 quitte le moniteur.

1.8 Problèmes "Producteur-Consommateur"

Le problème du producteur et du consommateur est un classique des problèmes de synchronisation. Le producteur génère une information qui doit être transmise au consommateur. La communication se passe selon le schéma suivant :
- le producteur produit des messages et les dépose dans une zone de mémoire commune aux deux processus.
- le consommateur prélève les message et le utilise
- les messages, dans la zone de mémoire commune, sont gérés en queue.

Le tampon (zone mémoire commune) permet d'absorber les différences temporaires de débit.

La résolution du problème par le moniteur implique d'implémenter un moniteur avec deux signaux :

– le signal nonplein, qui transporte la relation nb-message $<$ n
– e signal nonvide, qui transporte la relation nb-message $>$ 0
Si le producteur veut déposer un message alors que le tampon est plein, il se met en attente du signal nonplein. Après avoir déposé le message, le producteur envoie le signal nonvide puisqu'à ce moment là, la condition nb-message $>$ 0 est vrai. De même, si le consommateur veut prélever un message alors que le tampon est vide, il se met en attente du signal nonvide. Après avoir consommé un message, le consommateur envoie le signal nonplein, puisqu'à ce moment nb-message $<$ n.

2 Synchronisation en JAVA

2.1 Les Moniteurs JAVA

Java offre une forme simplifiée de moniteur. Les principales différences sont les suivantes : un seul signal (anonyme) par moniteur Java et sémantique des moniteurs de Java plus rudimentaire.
L'exclusion mutuelle d'un moniteur s'obtient en déclarant la méthode d'une classe *synchronized*. A l'intérieur d'une méthodes synchronized de l'objet O, la synchronisation s'exprime grâce aux méthodes suivantes :
– **wait()** : bloque le thread appelant et lève l'exclusion mutuelle sur O
– **wait(long timeout)** : identique à wait(), mais si le signal anonyme associé à O n'a pas été envoyé dans le délai spécifié par timeout, le thread est débloqué (mais celui-ci doit encore obtenir l'accès au moniteur O).
– **notify()** : débloque l'un des threads en attente du signal tout en conservant l'accès exclusif au moniteur. (Donc le thread débloqué doit encore obtenir l'accès au moniteur).
– **notifyAll()** : débloque tous le threads en attente du signal. (Ils doivent encore obtenir l'accès au moniteur pour être exécuté).
La sémantique en Java ne donne pas la garantie que la *condition* est encore vraie lorsque le thread débloqué reprend le moniteur. Pour cette raison, il faut toujours utiliser while : *while(!condition)wait() ;*
En Java, à chaque objet O est associé un verrou. L'entrée dans un moniteur est implémentée par l'exécution de *lock(verrou-o)* et la sortie d'un moniteur est implémentée par l'exécution de *unlock(verrrou-o)*.
Attention au appel imbriqués de moniteur : considérons maintenant deux objets O1 et O2 avec les deux méthodes synchronisée m1() et m2(). Considérons un thread T qui depuis O1.m1() appelle O2.m2() ; dans m2(), le thread T exécute wait(). Cela conduit T à relâcher verrou-o2 et à attendre le signal de O2. Par contre T détient toujours verrou-o1 ! Cela peut représenter un risque d'interblocage.
On dit qu'une class C est **thread-safe**, si elle est correcte même dans le cas où ses méthodes sont appelées simultanément par plusieurs threads.

2.2 Paquetages pour la programmation concurrente

– **ReentrantLock** : un verrou avec les méthode *lock()* et *unlock()*. Il est conseillé d'utiliser un try/catch pour la section critique et de mettre le unlock() dans le finally ce qui assure que unlock() sera toujours exécuté.
– **Condition** : l'interface condition permet d'avoir plusieurs signaux par moniteurs. Une condition a les méthodes suivantes :
  – **await()** : comme wait()
  – **signal()** : comme notify()
  – **signalAll()** : comme notifyAll()
  Une condition est créée à partir d'un objet de la class ReentrantLock (*mutex.newCondition()*). Les méthodes si dessus ne peuvent que être exécuté par un thread qui a le verrou associé. Si on appelle *await()*, on va alors lacher le verrou associé et tenter de le reprendre un fois que le thread est "notify".
– **Semaphore** : un sémaphore avec les méthode *acquire()* qui se comporte comme la méthode P() et la méthode *release()* qui se comporte comme la méthode V().
– **CyclicBarrier** : une barripre de synchronisation, pour un ensemble de threads, assure la synchronisation suivante. Tous les threads sont bloqués par la barrière jusqu'à ce que le nombre spécifié de

threads aient atteint la barrière. On utilise alors la méthode *barr.await()* pour se mettre en attente sur la barrière.
- **ArrayBlockingQueue<E>** : un classe pour résoudre le problème des producteurs-consommateurs. Elle implémente le tampon dans un tableau de taille fixe. Le méthodes pour ajouter et retirer un élément du tampon sont appelées *put(E e) et take()*.
- **ExecutorService** : un interface permettant de créer de pool de threads. Un pool de thread permet de contrôler la création de threads, par exemple pour assure que le nombre de threads ne dépasse jamais une valeur maximale. Un pool de thread permet également de recycler les threads pour éviter la répétition des opérations coûteuses de création et de destruction de threads. Il existe deux méthodes (de la classe Executors) pour créer ces pools :
  - *newSingleThreadExecutor()*
  - **newFixedThreadPool(int nbThreads)**

  Un **tâche** est une séquence d'instruction, alors qu'un thread est l'entité qui exécute une tâche. Un tâche implémente l'interface **Runnable**. On utilise la méthode *execute(Runnable r)* pour faire exécuter un tâche à un pool de thread.
- **ScheduledExecutorService** : un interface pour l'exécution différée et l'exécution périodique de tâches. C'est un pool de thread qui a la capacité d'exécuter des tâches avec un délai et de manière répété en utilisant la méthode *schedule()*.

3 Multiprocesseurs

Dans le cas d'un multiprocesseur à mémoire partagée, le masquage des interruptions ne permet pas d'implémenter un section critique. L'exclusion mutuelle de base requiert une instruction spéciale. Il s'agit généralement soit de l'instruction *testAndSet*, soit de l'instruction *compareAndSwap*.

3.1 testAndSet

L'instruction testAndset permet de lire et écrire un emplacement en mémoire de manière atomique, c-à-d en exclusion mutuelle (implémentation hardware). L'instruction a deux opérandes : une adresse mémoire et un entier. L'instruction permet de manière atomique de lire le contenu de adr et d'y écrire val.

Algorithm 3: testAndSet(adr,val)

```
ancVal = memory[adr];
memory[adr] = val;
return (ancVal == val);
```

Puisque cette opération est atomique, aucun autre testAndSet ne peut être appelé au milieu de l'exécution d'un autre testAndSet. Avec testAndSet, une section critique est facile à implémenter en utilisant un variable partagée x initialisée à 0 (x==0 indique que la section critique est libre, x==1 indique que la section critique est occupée).
- Le code *while(testAndSet(adr,1))* est exécuté à l'entrée de chaque procédure du noyau.
- Le code *x=0* est exécuté à la sortie de chaque procédure du noyau.

3.2 compareAndSwap

L'instruction compareAndSwap est un alternative à testAndSet avec un spectre d'utilisation plus large. L'instruction a trois opérande : une adresse mémoire et deux valeurs (valAttendue et nouvVal). L'instruction permet de manière atomique de remplacer conditionnellement un emplacement mémoire avec une nouvelle valeur et de retourner l'ancienne valeur.

Algorithm 4: compareAndSwap(adr,valAttendue,nouvVal)

```
ancVal = memory[adr];
if ancVal == valAttendue then
    memory[adr]=nouvVal;
end
return ancVal;
```

L'implémentation d'une section critique avec compareAndSwap requiert également l'utilisation d'une variable partagée x aussi initialisé à 0==libre.
- Le code *while(compareAndSwap(adr,0,1)==1)* est exécuté pour entrer dans la section critique.
- Le code *x=0* est exécuté à la sortie de la section critique.

3.3 Solutions non-bloquantes

Une solution bloquante a la caractéristique qu'un processus peut empêcher le progrès des autres processus. Cette solution présente des risques d'interblocage ainsi que des problèmes si le processus en section critique ne quitte jamais la section critique.
On a alors développé des solutions non-bloquante et on le classe en deux types :
- **solutions sans verrouillage** (lock-free) : une telle solution, dans le cas d'un ensemble P de processus exécutant une opération concurrente, garantit qu'un des processus p de P terminera l'opération. Une fois p ayant terminé l'opération, si aucun nouveau processus ne vient agrandir P, la terminaison d'un autre processus de P-p est garantie, et ainsi de suite.
- **les solutions sans attente** (wait-free) : une telle solution garantit que tout processus qui exécute une opération l'exécutera en un nombre borné d'instructions, la borne étant indépendante du nombre d'autre processus exécutant l'opération.

3.4 Les Rendez-vous

Le mécanisme de rendez-vous est un mécanisme de synchronisation fondamental. Il est aussi important dans le contexte de la modélisation des systèmes concurrents. C'est un mécanisme de synchronisation par les données : l'échange d'information et la synchronisation entre deux processu sont réalisées conjointement. Cette synchronisation s'exprime à l'aide des deux commandes suivantes, où P est un nom de processus :
- **P !e** : commande d'envoi. Envoie la valeur de l'expression e au processus P. Le processus exécutant P !e est bloqué jusqu'à la réception par P de la valeur envoyée.
- **P ?v** : commande de réception. Stocke dans la variable v la valeur reçue du processu P. Le processus exécutant P ?v est bloqué jusqu'à la réception de la valeur envoyée par P.

Alors que le blocage du processus récepteur est habituelle, la particularité du rendez-vous est le blocage du processus émetteur. Le transfert de l'information a lieur à l'instant où les deux commandes ont été exécutées. Cet instant est appelé rendez-vous. Contrairement aux solutions (producteur-consommateur) vues précédemment, la communication et la synchronisation sont ici exprimées à l'aide du même mécanisme. En utilisant les rendez-vous pour ce problème (p-c),les deux processus s'exécutent de manière fortement synchronisée car cette solution n'utilise pas de tampon entre le producteur et le consommateur.
On peut utiliser la notion de canal pour implémenter une désignation "indirecte" : On envoie sur un canal et on attend la réception sur un canal. Plusieurs processus peuvent envoyer/recevoir simultanément sur un même canal. On peut aussi utiliser le rendez-vous avec désignation indirecte sans transfert de donnée pour implémenter une section critique.

3.4.1 UPPAAL

Le rendez-vous est le mécanisme utilisé par UPPAAL pour synchroniser les automates, ce qui permet de modéliser les verrous, les sémaphores,...

3.5 Thread POSIX

POSIX est l'acronyme de *Portable Operating System Interface*. Un thread POSIX est identifié par un variable de type *pthread_t* et il est créé dynamiquement grâce à la fonction :

```
int pthread_create(thread, attibuts, code, argument)
```

qui a quatre arguments :
– thread : adresse de la variable qui contiendra l'identificateur du thread créé.
– attributs : attributs du thread (NULL par défaut)
– code : adresse de la fonction qui correspondra au code du thread.
– argument : argument passé à la fonction "code".
Le thread est créé dans le processus courant et peut s'exécuter immédiatement (pas d'appel start nécessaire).
Un thread POSIX se termine en appelant la fonction

```
void pthread_exit(void *retval)
```

La fonction `int pthread_join(pthread_t thread, void **value_ptr)` permet comme la méthode join() de JAVA à un thread de se bloquer dans l'attente de la terminaison d'un thread T. (2e paramètre permet de récupérer l'éventuel paramètre de `pthread_exit` spécifié par T)
La fonction `pthread_t pthread_self()` retourne l'identificateur du thread courant.
POSIX offre trois mécanisme de synchronisation, les verrous (appelés mutex), les sémaphores et les variable de condition. Voir polycop.