Programmation concurrente sans douleur (séance de révision)

Michel.riveill@univ-cotedazur.fr

http://www.i3s.unice.fr/~riveill

Race condition

- Une « condition de compétition » survient dans les logiciels lorsqu'un programme informatique, pour fonctionner correctement, dépend de la vitesse d'exécution relative des processus ou des threads impliqués.
- Il devient un bogue lorsqu'un ou plusieurs des comportements possibles sont indésirables.
- Pour avoir une « condition de compétition », il faut à minima :
  - Plusieurs processus ou thread souhaitant modifier une même donnée partagée
- Les effets d'une « condition de compétition » sont généralement difficile à reproduire car le résultat final est non déterministe et dépend de la vitesse d'exécution relative des processus ou des threads impliqués.
  - Les problèmes peuvent donc disparaître lorsqu'on fonctionne en mode de débogage, en ajoutant une journalisation supplémentaire ou en attachant un débogueur.

Approche théorique

- Pour modéliser le comportement d'un programme, il est possible d'utiliser les systèmes de transitions étiquetés (Labelled Transition System) ou leur notation algébrique (FSP)
- Un processus / thread est modélisé sous la forme d'un processus
- Un objet partagé est lui aussi modélisé sous la forme d'un processus
- Une preuve de correction est décrite sous la forme d'un processus décrivant comportement correct et incorrect
  - property permet de construire aisément les comportements correct et incorrect à partir du comportement correct
- Si la composition de l'application et de la preuve contient encore :
  - Un état puit = interblocage
  - Un état d'erreur = la preuve n'est pas satisfaite
  - Un cycle sur un sous ensemble d'action = famine des actions n'appartement pas au cycle

Programme correctement synchronisé

- Un programme est correctement synchronisé si et seulement si toutes les exécutions séquentielles cohérentes sont exemptes de « condition de compétition ».
- On utilise généralement 2 modèles de base
  - La section critique
    - Un seul processus peut exécuter un ensemble d'action
  - Le modèle lecteur-rédacteur
    - Un groupe de processus peut exécuter un même temps un ensemble d'actions (les lecteurs)
    - Un autre groupe de processus peut exécuter de manière exclusive un autre ensemble d'actions (les rédacteurs)
- Principe général
  - Prologue : vérifie que les conditions sont remplies pour accéder à la section critique exclusive ou partagée
  - Epilogue : libère la section critique et réveille si nécessaire le/les processus en attente

Section critique exclusive

Les règles de construction sont les suivantes :

- Exclusion mutuelle : il n'y a pas deux processus qui se trouvent en même temps dans leurs sections critiques.
- Absence d'interblocage : si un processus tente d'entrer dans sa section critique, alors un autre processus, pas nécessairement le même, finit par entrer dans sa section critique.
- Absence de famine : si un processus essaie d'entrer dans sa section critique, alors ce processus doit finalement entrer dans sa section critique.
- Aucune hypothèse sur les vitesses d'exécution des processus ne peut être faite
- Tous les processus exécutent un algorithme équivalent

Les maux de la programmation concurrente

- Propriété de sûreté (safety)
  - Quelque chose de mauvais ne doit pas arriver
  - Un événement indésirable ne s'est pas produit pendant l'exécution (lié au Passé)
  - Exemple :
    - l'accès à une ressource partagée comme l'imprimante nécessite que le processus utilisateur a un accès exclusif à la ressource (exclusion mutuelle).
    - L'interblocage se produit lorsque des processus concurrents s'attendent mutuellement. Un processus peut aussi s'attendre lui-même.
- Propriété de vivacité (liveness)
  - Quelque chose de bon peut toujours arriver
  - Un événement souhaité arrivera nécessairement (lié au futur)
  - Exemple : tout processus qui souhaite utiliser l'imprimante doit éventuellement avoir accès à l'imprimante.

Interblocage

- 4 conditions nécessaires et suffisantes
  - Ressources réutilisables : les processus concernés partagent des ressources qu'ils utilisent en vertu d'une exclusion mutuelle.
  - Acquisition progressive : les processus conservent les ressources qui leur ont déjà été allouées en attendant d'acquérir des ressources supplémentaires.
  - Pas de préemption : une fois acquises par un processus, les ressources ne peuvent être préemptées (retirées de force) mais sont seulement libérées volontairement.
  - Cycle d'attente : un cycle de processus existe de telle sorte que chaque processus détient une ressource que son successeur dans le cycle attend d'acquérir.
- Les solutions
  - L'autruche
  - La prévention : allocation ordonnée, allocation globale, algorithme du banquier
  - La détection/guérison : on construit le graphe des attentes, quand on détecte un cycle, on choisi un processus, réquisitionne ses ressources, défait ses actions et le recommence
  - L'évitement :
    - un processus peut attendre sur un processus plus ancien, sinon on réquisitionne ses ressources, défait ses actions et le recommence avec le même âge
    - Time out : à la fin du time out, on réquisitionne les ressources du processus, défait ses actions et le recommence avec le même âge

Les outils à notre disposition : approche par blocage

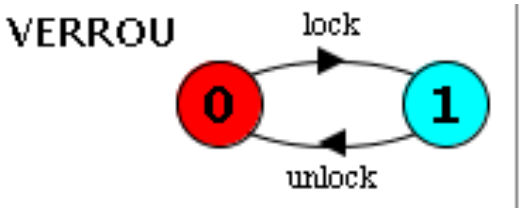
- Les verrous
  - Normaux
  - Ré-entrant
  - Read-Write
- Les sémaphores
  - Un compteur initialisé à une valeur donnée
  - Deux opérations atomiques
    - Down : -1 sur le compteur et si négatif la thread est bloquée
    - Up : si négatif, reveille une thread bloquée et +1 sur le compteur
- Les moniteurs
  - Un mutex pour construire une section critique
  - Des variables conditions qui possèdent deux opérations
    - Wait(condition, mutex) : sort de la section critique et bloque la thread sur la condition
    - Signal(condition) ou Broadcast(condition) : réveille une thread ou toutes les thread en attente sur la condition

Les outils à notre disposition : approche par blocage

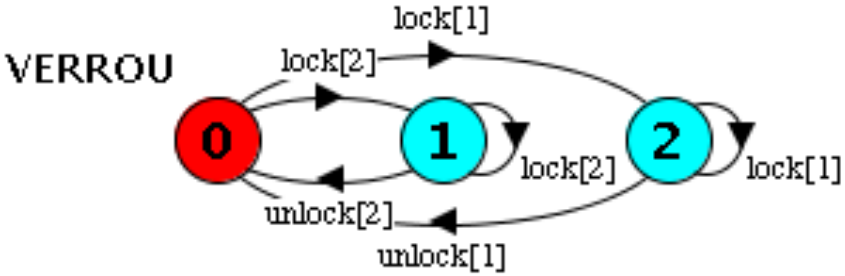
- Java
  - Verrous
  - Sémaphore
  - Moniteur en Java
- IPC System V
  - Sémaphore
  - Mémoire partagée
- Posix
  - Moniteur à l'aide de mutex, variables conditions
- Python
  - C'est vous qui l'avez étudié dans le cadre du projet

Verrou

- Le normal
  - VERROU = (lock -> unlock -> VERROU).



- Le ré-entrant
  - VERROU_RE = (lock[i:T] -> LOCK[i]),
    LOCK[i:T] = (unlock[i] -> VERROU_RE
    | lock[i] -> LOCK[i]).



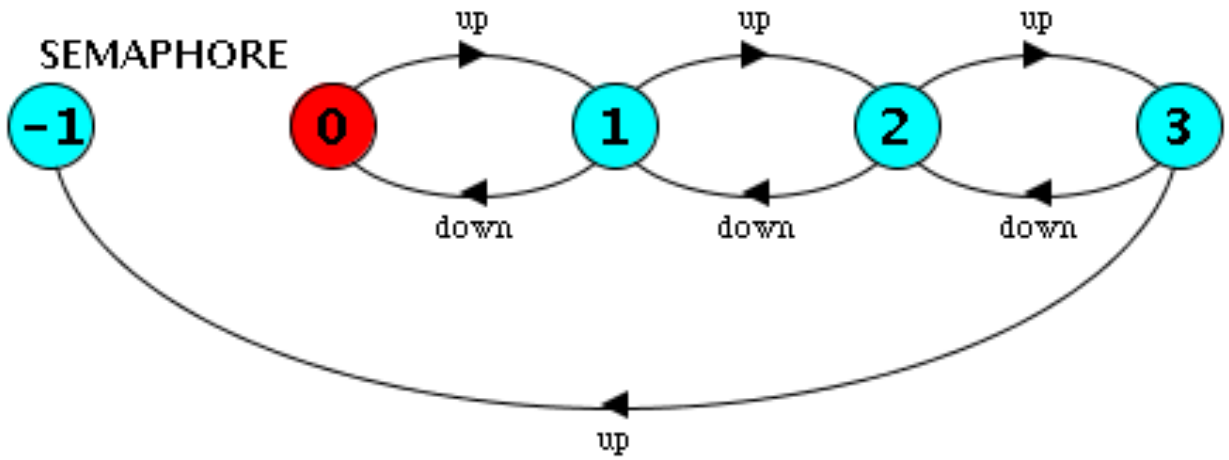
- Utilisation :
  - Contrôle de section critique

Utilisation de LTSA pour prouver qu'un verrou contrôle correctement l'accès à une section critique

- const Max = 3
- range Int = 0..Max
- PROC = (l.lock->enter->exit->l.unlock->PROC).
- VERROU = (lock -> unlock -> VERROU).
- ||APPLI = (p[1..3]:PROC || {p[1..3]}::l:VERROU).
- property MUTEX = (p[i:1..3].enter->p[i].exit->MUTEX).
- ||CHECK = (APPLI || MUTEX)<<{p[i:1..3].enter}.
- progress FAMINE = {p[1..3].enter}
- Check -> safety (absence d'état puit et d'état -1)
  - No deadlocks/errors
- Check -> progress (absence de famine)
  - No progress violations detected.

Sémaphore

- SEMAPHORE(N=0) = SEMA[N],
  SEMA[v:Int] = (up->SEMA[v+1]
  |when(v>0) down->SEMA[v-1]),
  SEMA[Max+1] = ERROR.



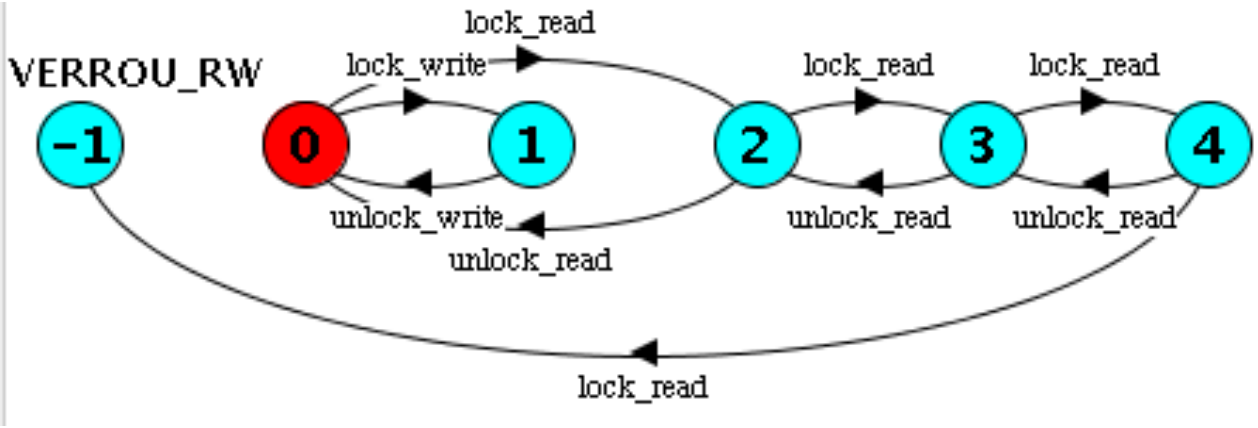
- Utilisation :
  - Contrôle de section critique (mutex = sémaphore initialisé à 1)
  - Bloquer un processus (initialisé à 0)
  - Contrôle de l'accès à une ressource dupliquée (initialisé à N)

Section critique et sémaphore

- A vous de prouver que la valeur d'initialisation d'un sémaphore à
  - 0 ou 2 ne permet pas
  - 1 permet
- de contrôler correctement l'accès à une section critique

Moniteur

- VERROU_RW = VERROU_RW[0][0],
  VERROU_RW[r:T][w:T] =
  (when (w==0) lock_read -> VERROU_RW[r+1][w]
  |unlock_read -> LOCK_R[i-1][w]
  |when (r+w==0) lock_write -> LOCK_W[r][w+1]
  |unlock_read -> LOCK_R[r][w-1]).



Moniteur en Java

```
Class VERROU_RW {
    int r=0, w=0;

    public synchronized void lock_read() {
        while (w!=0) wait();
        r += 1;
    }
    public synchronized void unlock_read() { r -= 1; notifyAll();}
    public synchronized void lock_write() {
        while (w+r!=0) wait();
        w += 1;
    }
    public synchronized void unlock_write() { w -= 1; notifyAll();}
}
```

Les outils à notre disposition : approche sans blocage

- Spin-lock : c'est un verrou qui remplace le fait de bloquer la thread si le verrou est déjà pris par une boucle ('spin')
  - Efficace uniquement si les threads ne sont que rarement bloquées que pour une très courte période
  - Plus avantageux en multi-cœur
- Mise en œuvre en l'absence d'instruction processeur spécifique
  - Pas très aisée : dekker, peterson, dijsktra
  - Nécessite une bonne compréhension de la mise à jour des caches mémoire
  - En java, nécessité d'utiliser des variables volatiles
- Mise en œuvre avec une instruction atomique processeur
  - TestAndSet : processeur des années 70
  - CompareAndSwap (CAS) : multi-coeur actuel

Les variables atomiques

- Pour faciliter le partage de types simples Java a introduit les variables atomiques
- Utilise l'opération processeur CAS pour effectuer sans blocage diverses opérations
- CAS effectue une séquence lecture-comparaison-écriture de manière atomique
  - CAS (X, $v_{désiré}$, $v_{nouveau}$) est équivalent à la séquence de code suivante exécutée de manière atomique
    - tmp = X
    - si tmp== $v_{désiré}$ alors X = $v_{nouveau}$
    - renvoie tmp

Les variables atomiques

- Les types disponibles sont (liste non exclusive) :
  - AtomicBoolean, AtomicInteger, AtomicIntegerArray, AtomicLong, AtomicLongArray, AtomicReference<V>
- Les principales méthodes (certaines ne sont que pour les types entier ou tableau)
  - <T> addAndGet(<T> delta)
  - <T> compareAndSet(<T> expect, <T> update)
  - <T> decrementAndGet()
  - <T> get()
  - <T> getAndAdd(<T> delta)
  - <T> getAndSet(<T> newValue)
  - <T> incrementAndGet()
  - void set(<T> newValue)

Le compteur

```
public class Counter {
    private int value;
    public synchronized int getValue() { return value; }
    public synchronized int increment() { return ++value; }
}
```

```
public class Counter {
    private AtomicInteger value = new AtomicInteger();
    public int getValue() { return value.get(); }
    public int increment() {
        int readValue = value.get();
        while (!value.compareAndSet(readValue, readValue+1))
            readValue = value.get();
            return readValue+1;
    }
}
```

- Quel était votre meilleure implémentation ?

Tâches, futur

- Tâches
  - Runnable
    - ```
      public interface Runnable {
          public void run();
      }
      ```
  - Callable<V>
    - ```
      public interface Callable<V> {
          public V call() throws Exception;
      }
      ```
    - Le résultat de la méthode call() ainsi que l'exception sont renvoyés dans un objet Future
- Future<T>
  - `Future.get()` permet de récupérer le résultat
    - null si la tâche est un Runnable
  - `Future.getCause()` permetde récupérer l'exception
    - Pas d'exception si la tâche est un Runnable
  - `Future.isDone()` permet de savoir que la méthode call/run est terminée

Les executors java

- Interface ExecutorService
  - Décrit les fonctionnalités pour l'exécution de taches
  - `ExecutorService executor = Executors.newSingleThreadExecutor();`
  - `ExecutorService executor = Executors.newFixedThreadPool(3);`

  - `void execute(Runnable command)`
  - `Future<T> submit(Callable<T> task)`
  - `Future<?> submit(Runnable task)`
  - `List<Future<T>> invokeAll(Collection<? Extends Callable<T>> tasks)`

  - `Boolean awaitTemination(long timeout)` // attend la fin de l'éxécution des tâches soumises
  - `void shutdown()` // n'accepte plus de nouvelle tache
- Une série d'exemple disponible ici : https://www.jmdoudoux.fr/java/dej/chap-executor.htm

Le framework Fork/Join java

- ForkJoinPool: une implémentation de l'ExecutorService qui gère les ForkJoinTasks.
  - fournit des méthodes de soumission de tâches,
    - `<T> invoque(ForkJoinTask<?> task)` // appel synchrone
    - `void execute(ForkJoinTask<?> task)` // appel asynchrone
    - `Future<T> submit(ForkJoinTask<?> task)` // appel asynchrone avec futur
- ForkJoinWorkerThread: une classe qui décrit un thread géré par une instance de ForkJoinPool.
  - responsable de l'exécution des ForkJoinTasks.

Le framework Fork/Join java

- ForkJoinTask: une classe de base abstraite pour les tâches qui s'exécutent dans un contexte ForkJoinPool.
  - décrit des entités de type thread qui ont un poids beaucoup plus léger que les threads normaux.
  - De nombreuses tâches et sous-tâches peuvent être hébergées par très peu de threads réels dans une instance de ForkJoinPool.
- RecursiveAction: une classe abstraite qui décrit une ForkJoinTask récursive sans résultat.
- RecursiveTask: une classe abstraite qui décrit une ForkJoinTask récursive avec résultat.

Utilisation du framework Fork/Join java

```
Résultat résoudre(Problème problème) {
    if (problème is petit)
        résoudre directement le problème
    else {
        découper le problème in parties indépendantes
        fork new sous-tache to résoudre chaque partie
        join all sous-tâche
        composer le résultat from sous-résultat
} }
```

- Cf. exemple tri d'un tableau (ForkJoin-sample.zip)
  - Attention au chargement dynamique des classes dans la comparaison des temps d'exécution.