Cours Programmation Concurrente

Master MIAGE M1

Jean-François Pradat-Peyre
Université Paris Nanterre - UFR SEGMI

2023-2024

1 : Synchronisation entre entités concurrentes, sémantique et illustrations en Java et C/Posix

Plan du cours

- Introduction aux problèmes de synchronisation
- Sémaphores : principe et utilisation
- Moniteurs : un concept plus évolué

Introduction

- Programme multi-tâches : coopération inter-tâches
  - Echange ou partage d'informations
  - Synchronisation pour le respect de la causalité et pour la protection des données
- Deux modèles
  - Système centralisé : privilégie la communication et la synchronisation par mémoire commune
  - Système distribué : privilégie la communication et la synchronisation par messages

Caractéristiques des systèmes distribués

- Ensemble de machines reliées en réseau (pouvant être temps réel)
- Pas de mémoire commune (et plusieurs ordonnanceurs)
- Coopération et synchronisation basées sur *l'échange de messages*
  - permet l'échange de données (un message transporte des données)
  - synchronisation implicite : on ne peut recevoir un message que s'il a été émis
  - permet aussi une synchronisation explicite : le message peut transporter une information de synchronisation

Mécanismes présents dans les systèmes distribués : mécanismes de communication

- Appel de procédures distantes
  - envoie d'une demande à un serveur
  - plusieurs sémantiques (appel bloquant ou non bloquant)
  - mécanisme système
  - exemple : RPC
- Invocation de méthodes distantes
  - appel d'une méthode d'un objet situé sur un site distant
  - mécanisme langage
  - exemple : Java RMI (Remote Method Invocation)

Caractéristiques des systèmes centralisés

- Une seule machine pour plusieurs tâches : *partage obligatoire*
- Mémoire commune
- Communication et synchronisation par *partage de données* en mémoire
  - simple à mettre en œuvre
  - pose le problème de la protection des données par rapport à des accès multiples (problème de cohérence) lorsque les opérations ne sont pas atomiques

Mécanismes présents dans les systèmes centralisés : mécanismes de protection

- Masquage des interruptions
  - dangereux, à réserver à des zones très ciblées (noyau système)
- Test and Set
  - bas niveau, plutôt matériel (au niveau du processeur)
- Sémaphore (Dijkstra - 1965)
  - assez bas niveau mais très commun et simple dans le cas général
- Moniteur (Hoare – 1974)
  - haut-niveau, le plus commode et le plus sûr (au niveau langage)
- Mécanismes basés sur l'échange de données également possible

Pourquoi des mécanismes de protection ?

- Considérons l'exemple suivant :
  - Deux tâches partagent un registre
  - Le registre est en fait un tableau de N entiers
  - Les tâches doivent régulièrement incrémenter chaque case du registre
  - L'incrémentation doit se faire en N opérations élémentaires

Nous étudions un programme implémentant cet exemple en C avec la norme Posix puis avec les langages Java et Ada

Exemple incorrecte (C/Posix) : entête et le codes tâches

```
#include <stdio.h>
#include <pthread.h>

#define TAILLE_REGISTRE 10
pthread_t id1, id2;

typedef int ValeurRegistre[TAILLE_REGISTRE];
ValeurRegistre LeRegistre;
```

```
void f(void){
     int i, j;
     for(j=0; j < 1000000; j++){
       for(i=0; i<TAILLE_REGISTRE; i++){
              LeRegistre[i] ++;
       }
     }
}
```

./...

Exemple incorrect (C/Posix) : le principal

```
main(){
    int i;

    for(i=0; i<TAILLE_REGISTRE; i++)

      LeRegistre[i] = 0;

    pthread_create(&id1, NULL, (void *(*)()) f, NULL);
    pthread_create(&id2, NULL, (void *(*)()) f, NULL);


    pthread_join(id1, NULL);
    pthread_join(id2, NULL);

    printf("Valeur finale du registre : \n");


    for(i=0; i<TAILLE_REGISTRE; i++)
      printf("%d \n", LeRegistre[i]);

}
```

Exemple incorrect (Java) : la classe Registre

```
class Registre {
    private long[] Tab;

    public Registre(int n) {
      Tab = new long[n]; // initialisé à 0 par défaut
    }
    public void inc() {
      for (int i = 0; i < Tab.length; i++) {Tab[i]++;}
    }
    public int taille(){
      return Tab.length;
    }
    public long get(int i){
      return Tab[i];
    }
}
```

Exemple incorrect (Java) : code des tâches

```
class monThread extends Thread {
     Registre leRegistre;

     monThread(Registre R) {
       leRegistre = R;
     }
     public void run() {
       for (int turn=0; turn<1000000; turn++) {
              leRegistre.inc();
       }
     }
}
```

Exemple incorrect (C/Posix) : le principal

```
public class Prog1 {

public static void main(String args[]) throws InterruptedException{
      public static void main(String args[]) throws InterruptedException {
        Registre R = new Registre(10);

        monThread Th1 = new monThread(R);
        monThread Th2 = new monThread(R);

        Th1.start(); // démarrage des threads
        Th2.start();

        Th1.join(); // attente de la terminaison des threads
        Th2.join();
        System.out.println("Valeur finale du Registre");
        for (int i = 0; i < R.taille(); i++) {
            System.out.println(R.get(i));
        }
    }}
```

Exemple incorrecte : exécution du programme

Voici un exemple d'exécution (C/Posix, Java ou autre)

```
Valeur finale du registre
   1143280
   1543282
   1177867
   1177867
   1177867
   1177867
   1177867
   1177867
   1177867
   1177867
```

Que des valeurs inférieures à 2.10^6
Explication **?**

On aurait du avoir 10 lignes avec la valeur 2000000 (deux millions)

Il est nécessaire de garantir qu'une séquence lecture / incrémentation / écriture soit finie avant qu'une autre ne puisse commencer !

Mécanismes de synchronisation

- Deux mécanismes classiques peuvent être utilisés pour protéger des données accédées par plusieurs tâches :
  - Les **sémaphores** qui ne peuvent être pris qu'un nombre déterminé de fois (si le sémaphore est déjà pris une nouvelle tentative bloquera l'appelant)
  - Les **moniteurs** qui permettent « d'encapsuler » des données en définissant des règles d'accès exclusif à ces données

Les sémaphores (Dijkstra 1965)

- Un sémaphore est un élément de synchronisation auquel est associées (au moins) quatre éléments :
  - un compteur interne entier (on le note S_CPT)
  - une file d'attente
  - deux opérations atomiques (exécution indivisible) nommées P et V
  - une opération atomique d'initialisation (et création) que l'on notera E0
- Si S est un sémaphore, on notera P(S) ou V(S) ou E0(S,V) l'appel d'une opération sur le sémaphore S
- On dit qu'un sémaphore est binaire si son compteur reste inférieur ou égal à 1 et on parle de Mutex ou de Lock

Les sémaphores : sémantique

P(S) : prend le sémaphore; *bloquant* si déjà pris

```
S_CPT --;
Si (S_CPT < 0) Alors
   Bloquer l'appelant et le mettre dans la file d'attente associée à S;
fin si
```

V(S) : rend le sémaphore; jamais bloquant

```
S_CPT ++;
Si (S_CPT <= 0) Alors
   Choisir un processus dans la file d'attente, le retirer de celle-ci et
   le réveiller (le débloquer);
fin si
```

E0(S, V) : initialise le compteur interne à la valeur V

```
S_CPT <-- V;
```

Les sémaphores : sémantique (suite)

- On a l'invariant suivant *S_CPT < 0 => Abs(S_CPT) = Lg(File Attente)*

- Dans le cas d'un sémaphore binaire, l'opération V(S) n'aura aucun effet si le compteur du sémaphore est déjà à un 1

- Il n'est pas forcément précisé quelle tâche est réveillée dans le cas où il y en a plusieurs en attente
  - gestion FIFO (on réveille la plus ancienne dans la file)
  - gestion par priorité (on réveille la plus prioritaire)

- La manipulation de sémaphores peut introduire le phénomène d'inversion de priorité

Les sémaphores : exemple d'utilisation

- Dans l'exemple précédent d'exécution incorrecte il faut "encadrer" les actions de
  - lecture du registre
  - incrémentation des cases du registres
  - écriture du registre

par la prise et le relâchement d'un sémaphore binaire (initialisé à 1)

```
P(S);
   lecture du registre
   incrémentation des cases constituant le registre
   écriture du registre
V(S);
```

- De la sorte, la séquence des opérations de lecture, d'incrémentation et d'écriture devient une opération atomique : il n'y a plus d'incohérence de la valeur commune du registre (une nouvelle opération ne peut commencer que lorsque celle qui est en cours est terminée)

Les sémaphores : guide d'utilisation

- Assurer une bonne protection (no trop ni trop peu)
  - N'utiliser les sémaphores que sur de petites sections critiques
  - Faire en sorte que celui qui rend le sémaphore soit celui qui l'a pris
  - Isoler les actions en section critique du reste du code
- Eviter l'interblocage en ordonnant la prise des sémaphores ou utiliser des tableaux de sémaphores à la Unix
  - ✓ exemple d'interblocage :
    - la tâche A prend S1, la tâche B prend S2 (*S1 et S2 sont deux sémaphores binaires*)
    - la tâche A tente de prendre S2 : elle se bloque
    - la tâche B tente de prendre S1 : elle se bloque

    -> ***on est dans une situation d'interblocage***

Sémaphores en C / Posix : Création

- Les prototypes des fonctions et les types sont définis dans "semaphore.h"
- Un sémaphore est une variable de type "sem_t"
- Il est initialisé par un appel de la primitive

```
int sem_init(sem_t *sem, int pshared, unsigned int valeur);
```

  - si "phsahed" vaut 0 le sémaphore ne peut pas être partagé entre tâches de différents processus (partage uniquement au sein d'un même processus)
  - valeur définit la valeur initiale de ce sémaphore (positif ou nul)
- Cette primitive correspond à la primitive E0 définit dans le chapitre précédent

Sémaphores en C / Posix : Prise et relâche

- Les deux opérations P et V sont implémentées par

```
P : sem_wait(sem_t * sem);
V : sem_post(sem_t * sem);
```

avec les mêmes comportements que les primitives génériques P et V

- Il existe également une version non bloquante de la primitive P :

```
int sem_trywait(sem_t * sem);
```

qui retourne 0 si quand la prise est possible (et non bloquante) et qui retourne EAGAIN sinon (dans le cas où l'appel normal serait bloquant)

Sémaphore binaire en C / Posix : les mutex

- Un « mutex » est un sémaphore binaire pouvant prendre un des deux états

  "lock" (verrouillé) ou "unlock" (dévérrouillé)

- Un « mutex » ne peut être partagé que par des thread d'un même process

- Un « mutex » ne peut être verrouillé que par un seul thread à la fois.

- Un thread qui tente de verrouiller un « Mutex » déjà verrouillé est suspendu jusqu'à ce que le « Mutex » soit déverrouillé.

Déclaration et initialisation d'un mutex

- Un mutex est une variable de type "thread_mutex_t"

- Il existe une constante `PTHREAD_MUTEX_INITIALIZER` de ce type permettant une déclaration avec initialisation statique du mutex (avec les valeurs de comportement par défaut)

```
pthread_mutex_t monMutex = PTHREAD_MUTEX_INITIALIZER;
```

- Un mutex peut également être initialisé par un appel de la primitive

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
```

avec une initialisation par défaut lorsque `mutexattr` vaut NULL

ex :`pthread_mutex_init( &monMutex, NULL);`

Prise (verrouillage) d'un mutex

- Un mutex peut être verrouillé par la primitive

```
int pthread_mutex_lock(pthread_mutex_t *mutex));
```

- Si le mutex est déverrouillé il devient verrouillé

- Si le mutex est déjà verrouillé par un autre thread la tentative de verrouillage suspend l'appelant jusqu'à ce que le mutex soit déverrouillé.

- Si le mutex est déjà verrouillé *par le même thread* l'appel peut être
  - soit bloquant (comportement par défaut) : attention risque d'interblocage
  - soit non bloquant avec incrémentation d'un compteur définissant le nombre d'exemplaires du mutex possédé par le thread (comportement à la Java)

Relâchement (déverrouillage) d'un mutex

- Un mutex peut être déverrouiller par la primitive

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Si le mutex est déjà déverrouillé, cet appel n'a aucun effet (comportement par défaut)

- Si le mutex est verrouillé, un des threads en attente obtient le mutex (qui reprend alors l'état verrouillé) et ce thread redevient actif (il n'est plus bloqué)

- L'opération est toujours non bloquante pour l'appelant

Exemple d'utilisation de mutex

```
typedef int ValeurRegistre[TAILLE_REGISTRE];
pthread_mutex_t monMutex = PTHREAD_MUTEX_INITIALIZER;

ValeurRegistre LeRegistre;
void f(void){
    int i, j;
    for(j=0; j < 1000000; j++){
      for(i=0; i<TAILLE_REGISTRE; i++){
            pthread_mutex_lock( &monMutex );
            LeRegistre[i] ++;
            pthread_mutex_unlock( &monMutex );
      }
    }
}
```

Création et initialisation d'un mutex

On verrouille le mutex : accès exclusif

Section critique (Atomique)

On déverrouille le mutex

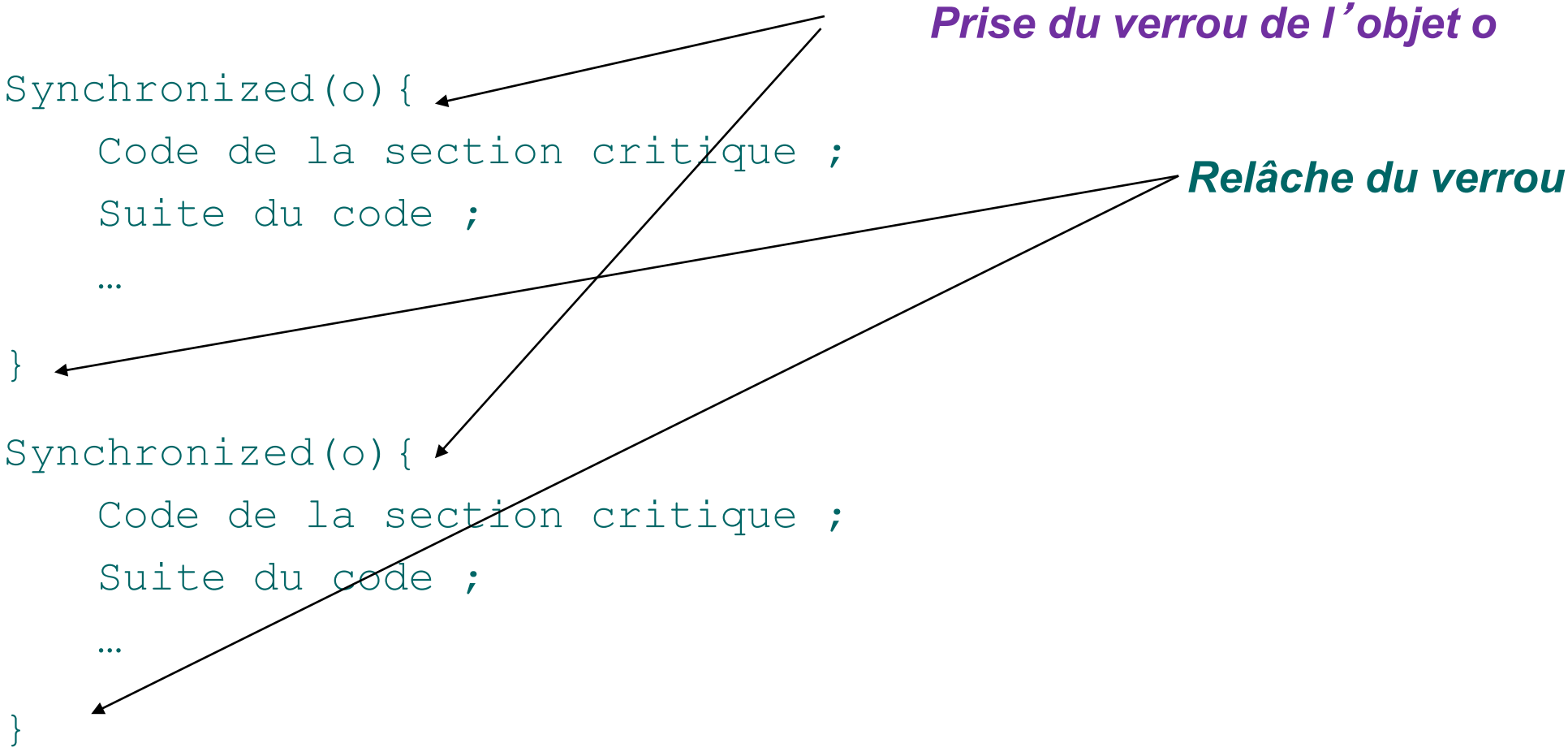
Les mutex (implicites) en Java

- La notion de sémaphore ou de mutex ne fait pas partie du langage (comme en C)
- Par contre chaque objet possède un verrou *récursif* (lock) et une file d'attente associée à ce verrou qui peut être manipuler comme un mutex
- Ce verrou ne peut, à un moment donné, n'être possédé que par un seul thread
  - Ce thread peut le posséder en plusieurs exemplaires
  - Ce verrou peut être possédé par différentes thread à déférents moments

- Ce verrou est pris lorsque l'on accède à une méthode (ou à un bloc) marqué par le mot clef « synchronized »
- Ce verrou est rendu (entre autres) lorsque la thread qui le détenait termine la méthode ou le bloc « synchronized »
- Lorsque le verrou est pris et qu'une thread tente de le prendre, le thread est mis en attente dans la file correspondante; il sera débloqué automatiquement lorsque le verrou sera relâché

Les mutex (implicites) en Java (suite)

- Chaque bloc ou méthode « synchronized » définit donc une section critique au niveau de l'objet



Les moniteurs

- Les moniteurs proposent une solution de "haut-niveau" pour la protection de données partagées (Hoare 1974)

- Ils simplifient la mise en place de sections critiques

- Ils sont définis par
  - des données internes (appelées aussi variables d'état)
  - des primitives d'accès aux moniteurs (points d'entrée)
  - des primitives internes (uniquement accessibles depuis l'intérieur du moniteur)
  - une ou plusieurs files d'attentes internes (différences importantes)

Les moniteurs : sémantique

- Seul un processus (ou tâche ou thread) peut être actif à un moment donné à l'intérieur du moniteur

- La demande d'entrée dans un moniteur (ou d'exécution d'une primitive du moniteur) sera bloquante tant qu'il y aura un processus actif à l'intérieur du moniteur

-> L'accès à un moniteur construit donc implicitement une exclusion mutuelle

Les moniteurs : sémantique (suite)

- Lorsqu'une tâche active au sein d'un moniteur ne peut progresser dans son travail (une certaine condition est fausse, il lui manque certaines ressources) il faut pouvoir "rendre" l'accès au moniteur et se mettre « en attente » d'une évolution de la condition de progression

- De même il faudra pouvoir « réveiller » une tâche en attente lorsque l'on modifie les variables internes du moniteur

- Il existe pour cela deux types de primitives
  - wait : qui met en attente l'appelant et libère l'accès au moniteur
  - signal : qui réveille une des tâches en attente à l'intérieur du moniteur (une tâche qui a exécuté précédemment un wait)

Les moniteurs : sémantique (suite)

- Selon les langages (ou les normes) ces mécanismes peuvent être implémentés de différentes façons
  - primitives « pthread_cond_wait / pthread_cond_signal » en Posix et variables conditionnelles
  - méthodes « wait / notify / notifyAll » en Java et méthodes « synchronized »
  - gardes associées aux entrées en Ada, instruction « requeue » et objets protégés
  - Une ou plusieurs files d'attente associées aux condition de progression

- La sémantique des réveils peut varier :
  - Qui réveille t-on (le plus ancien, le plus prioritaire, un choisi au hasard, …)
  - Quand réveille t-on (dès la sortie du moniteur, au prochain ordonnancement, …)

Les moniteurs : exemple

- Dans l'exemple précédent il faut encapsuler le registre et les opérations de manipulations du registre dans un moniteur

- Définition du moniteur

Données Internes :
valeur locale du registre (R_Local)

Points d'entrée :
Incrémente_Le_Registre
Lecture du registre

Primitives_Internes :
Ecriture du registre

Moniteurs Posix : primitives associées

- Un moniteur Posix est l'association
  - d'un mutex (type `pthread_mutex_t`) qui sert à protéger la partie de code où l'on teste les conditions de progression
  - d'une variable conditionnelle (type pthread_cond_t) qui sert de point de signalisation :
    - ✓ on se met en attente sur cette variable par la primitive (le mutex est relâché)

```
pthread_cond_wait(&laVariableConditionnelle, &leMutex);
```

    - ✓ on signale sur cette variable avec la primitive (la tâche réveillée aura le mutex)

```
pthread_cond_signal(&laVariableConditionnelle);
```

Moniteurs Posix : Schéma d'utilisation

Soit la condition de progression ***C***

Le schéma d'utilisation des moniteurs Posix est le suivant :

```
pthread_mutex_lock(&leMutex);
évaluer C;
while ( ! C ){
   pthread_cond_wait(&laVariableConditionnelle, &leMutex);
   ré-évaluer la valeur booléenne de C // presque toujours nécessaire
}
Faire le travail;// sous protection du Mutex
pthread_mutex_unlock(&leMutex);
```

Moniteurs en Java

- Lorsqu'un thread, exécutant une méthode d'un objet marquée « `synchronized` », ne peut plus progresser du fait d'une condition logique non remplie (par exemple le thread vient chercher un message qui n'est pas encore déposé) il peut se mettre en attente d'une évolution de cette condition en faisant un « wait() » (ce qui évite une attente active inutile et un potentiel blocage du programme)

- Sur exécution du « wait() »
  - Le thread appelant est suspendu
  - Le verrou associé à l'objet concerné (qui était obligatoirement détenu par le thread appelant) est relâché
  - Les autres verrous que pouvait détenir le thread **ne sont pas relâchés**
  - Le thread est mis dans une file d'attente interne **différente** de celle associée au verrou

Moniteurs en Java (suite)

- Lorsqu'un thread, exécutant une méthode d'un objet marquée « `synchronized` », *modifie* une condition qui bloquait un thread (par exemple le thread dépose un message qui est peut être attendu) il peut tenter de réveiller un thread en attente en faisant un « notify() » (ou un « notifyall() » s'il pense que plusieurs thread était bloqués et qu'ils pourraient maintenant progresser)

- Sur exécution du « notify() »
  - Le thread appelant continu son exécution normalement
  - Il garde donc tous les verrous qu'il possède (**dont celui de l'objet concerné**)
  - Un thread **qui est dans la file d'attente interne associée au « wait »** (et non dans la file d'attente du verrou) est débloqué
  - Ce thread débloqué reprendra son exécution lorsqu'il aura réussi à obtenir le verrou de l'objet concerné **: compétition possible sur l'obtention de ce verrou**

Moniteurs en Java (suite)

- Sur exécution du « notifyAll() »
  - Le thread appelant continu son exécution normalement
  - Il garde donc tous les verrous qu'il possède (dont celui de l'objet concerné)
  - TOUS les thread **qui sont dans la file d'attente interne associée au « wait »** (et non dans la file d'attente du verrou) sont débloqués
  - Les thread débloqués reprendront leur exécution lorsqu'ils auront réussi à obtenir le verrou de l'objet concerné **: compétition sur l'obtention de ce verrou**
- Si aucun thread n'est en attente, le signal associé au « notify() » ou « notifyAll() » est perdu (pas de mémorisation des signaux)
- Du fait de la compétition pour l'obtention du verrou après le réveil le code d'attente des thread sera la plupart du temps encapsulé dans une boucle

Moniteurs en Java (suite)

Exemple typique de code d'attente (barrière logique) :

```
while (! conditionDeProgression ){
  try {
    wait(); // attente d'un notify ou notifyAll puis du verrou
   } catch (InterruptedException e) {
     e.printStackTrace();
   }
}
```

API java.util.Concurrent

- De (très) nombreux outils disponibles dans le **Package java.util.concurrent**

- Entres autres :
  - Sémaphore
  - Barrières
  - Buffers et paradigme producteurs / consommateurs

Classe Semaphore

Classe de sémaphores à compte avec une sémantique usuelle de sémaphores

- Constructeurs :
  - **Semaphore(int permits)**
  - **Semaphore(int permits, boolean fair)** (gestion FIFO des threads en attente)
- Méthodes utiles
  - void **acquire()** Acquires a permit from this semaphore, blocking until one is available, or the thread is ***interrupted***. (ou acquireUninterruptibly())
  - void **acquire(int permits)** Acquires the given number of permits from this semaphore, blocking until all are available, or the thread is interrupted. (ou acquireUninterruptibly(int permits))
  - void **release()** Releases a permit, returning it to the semaphore.
  - void **release(int permits)** Releases the given number of permits, returning them to the semaphore.

Classe ArrayBlockingQueue<E>

- Buffer Producteurs / Consommateurs borné (utilise un tableau)
- Constructeurs
  - **ArrayBlockingQueue**(int capacity) Creates an ArrayBlockingQueue with the given (fixed) capacity and default access policy.
  - **ArrayBlockingQueue**(int capacity, boolean fair) Creates an ArrayBlockingQueue with the given (fixed) capacity and the specified access policy (*fair - if true then queue accesses for threads blocked on insertion or removal, are processed in FIFO order; if false the access order is unspecified*).
- Méthodes utiles
  - void **put**(E e) Inserts the specified element at the tail of this queue, waiting for space to become available if the queue is full.
  - E **take**() Retrieves and removes the head of this queue, waiting if necessary, until an element becomes available.

Voir aussi le classe **LinkedBlockingDeque<E>** (buffer non borné)

Classe CyclicBarrier

- Permet à un ensemble fixe (en nombre) de threads de s'attendre sur une barrière
- Constructeurs
  - **CyclicBarrier**(int parties) Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it and does not perform a predefined action when the barrier is tripped.
  - **CyclicBarrier**(int parties, Runnable barrierAction) Creates a new CyclicBarrier that will trip when the given number of parties (threads) are waiting upon it, and which will execute the given barrier action when the barrier is tripped, performed by the last thread entering the barrier.
- Méthodes utiles
  - int **await**() Waits until all parties have invoked await on this barrier.
  - int **await**(long timeout, TimeUnit unit) Waits until all parties have invoked await on this barrier, or the specified waiting time elapses.
- Voir aussi les classes **Phaser** (plus complexe) ou **CountDownLatch** (plus restreinte)

Synchronisation en mémoire partagée : bilan

- Sémaphores :
  - mécanisme simple mais qui peut conduire à des erreurs subtiles
  - demande une grande rigueur dans leur utilisation
  - à réserver pour la mise en place de petites sections critiques

- Moniteurs :
  - mécanisme de plus haut niveau
  - simplifie la mise en place de section critique
  - peut néanmoins conduire à des erreurs (mauvaise protection, trop de points d'entrées, oubli de réévaluation des conditions de progression)

Conclusion sémantique

- En modèle réparti la communication et la synchronisation entre tâches se fait par l'échange de messages; on utilise principalement deux mécanismes :
  - L'appel de procédures à distance (RPC), mécanisme plutôt système
  - L'invocation de méthodes distantes (RMI), mécanisme plutôt langage
- En modèle centralisé la communication et la synchronisation reposent sur le partage de données communes; il faut alors « **protéger** » l'accès ces données à l'aide de deux mécanismes :
  - Les sémaphores, plutôt système
  - Les moniteurs, plutôt langage

Conclusion langage

- Java offre (maintenant) plusieurs mécanismes de synchronisation
  - Section critiques grâce aux blocs et méthodes « synchronized »
  - moniteurs grâce aux blocs et méthodes « synchronized » et aux méthodes wait / notify / notifyall
  - Sémaphores, barrières, buffers Prod/Cons (API Concurrent depuis Java 1.5)

- Posix propose plusieurs mécanismes
  - Mutex : pour les sections critiques
  - Sémaphores : pour les sections critiques et les coopérations élaborées
  - Variables conditionnelles : pour l'utilisation de moniteur

Les prochaines séances

- S2 : Créer des entités concurrentes, sémantique et mise en pratique
- S3 : Synchronisation entre entités concurrentes, sémantique et illustrations en Java et C/Posix
- S4 : Paradigmes de la concurrence 1
- S5 : Paradigmes de la concurrence 2
- S6 : Programmation concurrente en Python