Threads

Motivation

Processus UNIX

- isolement – ne permettent pas de partager simplement des ressources
- coût important de l'opération de changement de contexte

Idée : raffiner la notion de processus afin de favoriser le partage de données et accélérer le changement du contexte

Thread : fil d'exécution, activité, processus léger (light-weight process)

Point de vue utilisateur : processus qui partagent l'espace d'adressage

Caractéristiques d'un processus

- machine virtuelle (tâche)
  - espace d'adressage
  - propriétaire et groupe
  - descripteurs de fichiers
  - handlers de signaux
  - ...
- activité (fil d'exécution)
  - pile
  - contenu de registres
  - attributs d'ordonnancement
  - ...

Un processus peut "héberger" plusieurs threads

Threads vs Processus

	communication	synchronisation
threads	facile	difficile
processus	difficile	facile

Threads noyau vs Threads utilisateur (1)

Threads noyau	Threads utilisateur
gestion	
nécessite la présence d'appels système spécifiques	portable sur tous les systèmes
création	
nécessite un appel système	ne nécessite pas d'appel système et donc moins coûteuse
commutation entre deux threads	
commutation par le noyau avec changement de contexte	commutation dans l'espace utilisateur, donc plus légère

Threads noyau vs Threads utilisateur (2)

Threads noyau	Threads utilisateur
appels système	
pas de problème	un appel bloquant bloque toutes les threads du processus
ordonnoncement	
chaque thread dispose de même temps CPU que les autres processus/threads	utilisation globale du CPU = celle du processus d'accueil
parallélisme	
oui	non

Implementation

- approche M-1 (many-to-one) : plusieurs threads utilisateur sont associées à une thread noyau
- approche 1-1 (one-to-one) : à chaque thread utilisateur correspond une thread noyau
- approche M-M (many-to-many) : différents threads utilisateur sont multiplexées sur un nombre limité de threads noyau

La norme Posix.1c définit des threads indépendantes de l'implémentation (*pthreads*). Les applications sont portables même sur des systèmes n'implémentant pas la notion de processus.

Les threads sous Linux

```
#include <sched.h>
int clone(int (*fn)(void *), void *child_stack,
   int flags, void *arg);
```

Un genre de `fork()` grâce auquel le fils partage l'espace d'adressage du père.

Cette fonction ne concerne jamais le programmeur applicatif. On utilise les fonctions de la bibliothèque *LinuxThreads*

Pour utiliser LinuxThreads :

- `#include <pthread.h>` dans les fichiers source
- utiliser l'option `-D_REENTRANT` du compilateur
- utiliser l'option `-lpthread` de l'éditeur de liens

Création d'une thread

```
#include <pthread.h>
int pthread_create(
   pthread_t *thread,                          identificateur de thread
   pthread_attr_t *attr,                       attributs
   void *(*start_routine)(void *),             fonction à exécuter
   void *arg);                                 argument de la fonction

pthread_t pthread_self(void);

int pthread_equal(pthread_t thread1,
   pthread_t thread2);
```


Hello, Thread!

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

void *print(void *ptr)
{ printf("%s\n", (char *)ptr);
   return NULL;
}

int main(void)
{  pthread_t th1, th2;
   char *msg1 = "hello";
   char *msg2 = "thread";

   pthread_create(&th1, NULL, print, (void *)msg1);
   pthread_create(&th2, NULL, print, (void *)msg2);
   return 0;
}
```


Terminaison et attente d'une thread

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
void pthread_exit(void *retval);
```

Hello, Thread! revisited

```
...
   void *ret;

   pthread_create(&th1, NULL, print, (void *)msg1);
   pthread_join(th1, &ret);
   pthread_create(&th2, NULL, print, (void *)msg2);
   pthread_join(th2, &ret);
...
```


Attributs d'une thread

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_getXXX(const pthread_attr_t *attr,...);
int pthread_attr_setXXX(pthread_attr_t *attr,...);
```

XXX ∈ {detachstate, schedpolicy, schedparam, inheritsched, scope}

Annulation d'une thread (1)

```
int pthread_cancel(pthread_t thread);
```

Attention : l'annulation d'une thread peut être dangereuse!

Une thread annulée se comporte comme si elle invoquait `pthread_exit(PTHREAD_CANCELED);`

La demande peut être acceptée, refusée, ou repoussée jusqu'à un *point d'annulation*.

Annulation d'une thread (2)

```
int pthread_setcancelstate(int state, int *oldstate);
```

`state=PTHREAD_CANCEL_ENABLE` – accepter requêtes d'annulation (défault)
`state=PTHREAD_CANCEL_DISABLE` – refuser requêtes d'annulation

Les requêtes d'annulation ne sont pas mémorisées

```
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &olds);
...
/* arrivée d'une demande d'annulation */
...
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &olds);
/* la thread continue son exécution */
```


Annulation d'une thread (3)

```
int pthread_setcanceltype(int type, int *oldtype);
```

`type=PTHREAD_CANCEL_DEFERRED` – la thread se terminera dans le point d'annulation suivant (défault)
`type=PTHREAD_CANCEL_ASYNCHRONOUS` la thread se termine immédiatement

Points d'annulation

- `pthread_cond_wait()`, `pthread_cond_timedwait()`, `pthread_join()`, `pthread_testcancel()`
- certains appels système
- certaines fonctions de la bibliothèque standard

Attention : une thread configurée avec annulation asynchrone ne doit jamais invoquer des appels système

Configurations recommandées

- `PTHREAD_CANCEL_DISABLE` – zone critique où l'on ne peut pas supporter aucune annulation, même si on invoque un appel système
- `PTHREAD_CANCEL_ENABLE`, `PTHREAD_CANCEL_DEFERRED` – comportement habituel des threads
- `PTHREAD_CANCEL_ENABLE`, `PTHREAD_CANCEL_ASYNCHRONOUS` – zone de calculs lourds, sans appels système

Libération des ressources

Une thread peut être légitimement annulée à n'importe quel moment. Problèmes :

- la mémoire dynamique n'est pas automatiquement libérée
- les fichiers, tubes, sockets, ... ouverts ne sont pas fermés
- les verrous ne sont pas relâchés

```
#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *),
   void *arg);
void pthread_cleanup_pop(int execute);
```


Exemples (1)

```
FILE *fp;
fp = fopen("toto", "r");
pthread_cleanup_push(fclose, fp);

int fd;
fd = open("toto", O_RDONLY);
pthread_cleanup_push(close, (void *)fd);

char *buf;
buf = malloc(BUFSZ);
pthread_cleanup_push(free, buf);
```


Exemples (2)

`pthread_cleanup_push` et `pthread_cleanup_pop` doivent être exécutées en paires, dans le même bloc, au même niveau et ne peuvent pas être interlacés avec une autre paire.

```
buf = malloc(1024);
if (buf != NULL)
{  pthread_cleanup_push(free, buf);
   fd = open("toto", O_RDONLY);
   if (fd != -1)
   {  pthread_cleanup_push(close, (void *)fd);
      ...
      pthread_cleanup_pop(1); /* close(fd); */
   }
   pthread_cleanup_pop(1); /* free(buf); */
}
```


Threads et appels système

- `exit()` termine le processus (donc toutes les threads). Pour terminer la thread appelante, utiliser `pthread_exit()`
- `exec()` termine toutes les threads en exécution. Il reste une seule thread qui exécute le programme chargé
- `fork()` : dans le fils il y a une seule thread (copie de la thread appelante). Les piles, les zones de mémoire dynamique, les verrous, etc sont présents dans le fils.

```
int pthread_atfork(
   void (*prepare)(void),  avant le fork
   void (*parent)(void),   après le fork dans le père
   void (*child)(void));   après le fork dans le fils
```

Verrous d'exclusion mutuelle

- peut ête dans deux états : déverrouillé ou verrouillé (possédé par une thread).
- ne peut être pris que par une seule thread à la fois
- une thread qui tente de verrouiller un mutex déjà verrouillé est suspendue jusqu'à ce que le mutex soit déverrouillé

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *mutex,
   const pthread_mutexattr_t *mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

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


Exemple

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *fn(void *no)
{  ...
   pthread_mutex_lock(&mutex);
   printf("thread %d tient le mutex\n", (int)(no));
   section critique
   pthread_mutex_unlock(&mutex);
   ...
}

int main(void)
{  pthread_t th;
   int i;

   for (i = 0; i < 10; i++)
      pthread_create(&th, NULL, fn, (void *)i);
   pthread_exit(NULL);
}
```


Mutex et annulation (1)

Attention aux blocages !

`pthread_mutex_lock(&mutex);`
thread1 verrouille le mutex

    `pthread_mutex_lock(&mutex);`
    thread2 est suspendu

`pthread_cancel(thread2);`
thread1 fait une requête d'annulation de thread2. `pthread_mutex_lock` n'est pas un point d'annulation, donc thread2 continue à attendre

`pthread_mutex_unlock(&mutex);`
thread1 déverrouille le mutex, thread2 le verrouille

    `pthread_testcancel();`
    thread2 atteint un point d'annulation et se termine. Le mutex reste verrouillé

`pthread_mutex_lock(&mutex);`
thread1 est indéfiniment bloquée

Mutex et annulation (2)

Comportement correct : utiliser la pile d'annulation

```
pthread_mutex_lock(&mutex);
pthread_cleanup_push(pthread_mutex_unlock, &mutex);
section critique
pthread_cleanup_pop(1);/*pthread_mutex_unlock(&mutex);*/
```

Variables de condition

(Variable de) condition – mécanisme de synchronisation permettant à une thread de suspendre son exécution jusqu'à ce qu'une certaine condition (un prédicat) soit vérifiée.

Opérations fondamentales:

- signaler la condition (quand le prédicat devient vrai)
- attendre la condition (suspendre la thread jusqu'à ce qu'une autre thread signale la condition

Une condition doit toujours être associée à un mutex pour éviter des blocages

Fonctions de gestion

```
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *cond,
   pthread_condattr_t *cond_attr);
int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond,
   pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
   pthread_mutex_t *mutex,
   const struct timespec *abstime);
```

Rappel : `pthread_cond_wait` et `pthread_cond_timedwait` sont points d'annulation

Exemple (1)

```
double temperature;
phtread_mutex_t mhaut, mnormal;
pthread_cond_t chaut, cnormal;
```

Thread surveillant les variations de temperature

```
while (1)
{  temperature = mesurer_temperature();
   if (temperature > TMAX)
   {  pthread_mutex_lock(&mhaut);
      pthread_cond_broadcast(&chaut);
      pthread_mutex_unlock(&mhaut);
   }
   else
   {  pthread_mutex_lock(&mnormal);
      pthread_cond_broadcast(&cnormal);
      pthread_mutex_unlock(&mnormal);
   }
}
```


Exemple (2)

Thread gérant les alarmes

```
while (1)
{  pthread_mutex_lock(&mhaut);
   pthread_cond_wait(&chaut);
   pthread_mutex_unlock(&mhaut);
   lancer_alarme();
   pthread_mutex_lock(&mnormal);
   pthread_cond_wait(&cnormal);
   pthread_mutex_unlock(&mnormal);
   annuler_alarme();
}
```

Utilisation typique

```
int x,y; /* données partagées */
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
/* correspond à la condition  x > y */
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
/* le mutex associé */
```

Thread qui attend la condition

```
pthread_mutex_lock(&mut);
while (x <= y)
   pthread_cond_wait(&cond, &mut);
/* modifier x et y */
pthread_mutex_unlock(&mut);
```

Thread qui signale la condition

```
pthread_mutex_lock(&mut);
/* modifer x et y */
if (x > y)
   pthread_cond_signal(&cond);
   /* ou pthread_cond_broadcast(&cond); */
pthread_mutex_unlock(&mut);
```


Que se passe-t-il ?

```
pthread_mutex_lock(&mut);
thread1 verrouille le mutex
pthread_cond_wait(&cond, &mut);
thread1 déverrouille le mutex comme si elle
invoquait pthread_mutex_unlock(&mut)
thread1 est suspendu
    pthread_mutex_lock(&mut);
    thread2 verrouille le mutex
    pthread_cond_signal(&cond);
    thread2 signale la condition. thread1 essaye de vérrouiller le mutex
    comme si elle invoquait pthread_mutex_lock(&mut)
    le mutex est déjà verrouillé par thread2. thread1 est suspendu
    pthread_mutex_unlock(&mut);
    thread2 déverrouille le mutex
    thread1 verrouille le mutex et continue son exécution
```


Pourquoi un mutex associé ?

```
x = 1; y = 2;

while (x <= y)
la condition n'est pas vérifiée, donc thread1 va
exécuter pthread_cond_wait(&cond);
    x = 10; y = 5;
    if (x > y)
    la condition est vérifiée
    pthread_cond_signal(&cond);
    thread2 signale la condition.
    il n'y a aucune thread en attente. rien ne se passe.
pthread_cond_wait(&cond);
thread1 reste bloquée
```


Pourquoi `while` ?

```
x = 1; y = 2;

pthread_mutex_lock(&mut);
thread1 verrouille le mutex
if (x <= y) pthread_cond_wait(&cond, &mut);
la condition n'est pas vérifiée, thread1 déverouille le mutex et se met en attente
    pthread_mutex_lock(&mut);
    thread2 verrouille le mutex
    if (x <= y) pthread_cond_wait(&cond, &mut);
    la condition n'est pas vérifiée, thread2 déverouille le mutex et se met en attente
        x = 10; y = 5;
        if (x > y) pthred_cond_broadcast(&cond);
        thread3 signale la condition. thread1 et thread2 sont réveillées
        elles essayent de verrouiller le mutex. thread1 le verrouile, thread2 attend
x = 3; y = 4;
pthread_mutex_unlock(&mut);
thread1 modifie les données et déverrouille le mutex
thread2 verouille le mutex et continue son exécution
même si la condition n'est plus vérifiée
```

Exemple – philosophes

```
char baguette[N];
pthread_mutex_t mut;
pthread_cond_t peutmanger[N];

void *philosophe(void *ptr)
{  ...
   /* avant de manger */
   pthread_mutex_lock(&mut);
   while (!baguette[i] || !baguette[(i + 1) % N])
      pthread_cond_wait(&peutmanger[moi], &mut);
   baguette[i] = baguette[(i + 1) % N] = 0;
   pthread_mutex_unlock(&mut);
   ...
   /* après avoir fini de manger */
   pthread_mutex_lock(&mut);
   baguette[i] = baguette[(i + 1) % N] = 1;
   pthread_cond_signal(&peutmanger[(i − 1 + N) % N]);
   pthread_cond_signal(&peutmanger[(i + 1) % N]);
   pthread_mutex_unlock(&mut);
   ...
}
```


Sémaphores Posix.1b

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared,
   unsigned int value);
int sem_wait(sem_t *sem); /* opération P() */
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem); /* opération V() */
int sem_getvalue(sem_t *sem, int *sval);
int sem_destroy(sem_t *sem);
```

Attention : ne pas confondre sémaphores Posix.1b (préfixe `sem_`) et sémaphores IPC System V (préfixe `sem`)

Exemple

```
/* permettre aux trois threads maximum d'être dans la section critique */
sem_t sem;

void *fn(void *ptr)
{  while(1)
   {  sem_wait(&sem);
      section critique
      sem_post(&sem);
      section restante
   }
}

int main(void)
{  ...
   sem_init(&sem, 0, 3);
   for(i = 0; i < n; i++)
      pthread_create(&th, NULL, fn, (void *)i);
   ...
}
```


Implémentation simplifiée (1)

```
typedef struct
{  unsigned int val;
   pthread_mutex_t mut;
   pthread_cond_t cond;
} sem_t;

int sem_init(sem_t *sem, int shared, unsigned int v)
{  sem->val = v;
   pthread_mutex_init(&sem->mut, NULL);
   pthread_cond_init(&sem->cond, NULL);
}

int sem_destroy(sem_t *sem)
{  pthread_cond_destroy(&sem->cond);
   pthread_mutex_destroy(&sem->mut);
}
```

Implémentation simplifiée (2)

```
int sem_wait(sem_t *sem)
{  pthread_mutex_lock(&sem->mut);
   while (sem->val == 0)
      pthread_cond_wait(&sem->cond, &sem->mut);
   sem->val--;
   pthread_mutex_unlock(&sem->mut);
}

int sem_trywait(sem_t *sem)
{  int r;
   pthread_mutex_lock(&sem->mut);
   if (sem->val > 0)
   {  sem->val--;
      r = 0;
   }
   else r = EAGAIN;
   pthread_mutex_unlock(&sem->mut);
   return r;
}
```


Implémentation simplifiée (3)

```
int sem_post(sem_t *sem)
{  pthread_mutex_lock(&sem->mut);
   sem->val++;
   pthread_cond_broadcast(&sem->cond);
   pthread_mutex_unlock(&sem->mut);
}

int sem_getvalue(sem_t *sem, int *sval)
{  pthread_mutex_lock(&sem->mut);
   *sval = sem->val;
   pthread_mutex_unlock(&sem->mut);
}
```


Threads JAVA

Introduction

- La plupart de langages de programmation ne permettent pas aux programmeurs de spécifier des activités simultanées
- Pour faire du multithreading, les programmeurs utilisent appels système ou bibliothèques spécifiques
- Ada introduit mécanismes de simultanéité de haut niveau intégrés dans le langage de programmation
- Java est l'un des rares langages de programmation généralistes qui mettent des primitives de simultanéité à la disposition des programmeurs d'applications

Utilisation typique

- Interfaces utilisateur : permettre à une application de faire des calculs et de répondre à l'utilisateur en même temps
- Serveurs : un serveur multithread peut servir plusieurs clients en même temps (un thread par connexion)
- ...

Création

```
//Hériter la classe Thread
class MyThread1 extends Thread
{  ...
   public void run() {...}
   ...
}

//Implémenter l'interface Runnable
class MyThread2 implements Runnable
{  ...
   public void run() {...}
   ...
}
...
Thread th1 = new MyThread1();
Thread th2 = new Thread(new MyThread2());
// lancer un thread
th1.start(); // fait les initialisations nécéssaires et appelle run()
```


Exemple

```
public class ThrTest1 extends Thread
{  public void run()
   {  for (int i = 0; i < 10; i++)
      {  ralentir(100000);
         System.out.println(getName() + " :  " + i);
      }
   }

   private void ralentir(int n)
   {  double x = 123.45;
      for (int i = 0; i < n; i++)
         x = Math.exp(Math.log(x));
   }

   public static void main(String[] args)
   {  for (int i = 0; i < 5; i++)
         new ThrTest1().start();
   }
}
```


Synchronisation – motivation (1)

```
class ToujoursPair
{  private long cpt = 0;
   public void suiv() {cpt++; cpt++;}
   public long getCpt() {return cpt;}
}


public class SynTest2
{  public static void main(String[] args)
   {  final ToujoursPair tp = new ToujoursPair();

      new Thread()
      {  public void run()
         {  while (true) tp.suiv();
         }
      }.start();
```

Synchronisation – motivation (2)

```
      new Thread()
      {  public void run()
         {  long v;
            while (true)
            {  v = tp.getCpt();
               if (v % 2 == 1)
               {  System.out.println(v);
                  System.exit(1);
               }
            }
         }
      }.start();
   }
}
```


Sychronisation

- méthodes synchronisées

  ```
  synchronized void toto(...)  {...}
  ```

- blocs synchronisés

  ```
  synchronized(Objet) {...}
  ```

```
class ToujoursPair
{  private long cpt = 0;
   public synchronized void suiv()
   {  cpt++; cpt++;
   }
   public synchronized long getCpt()
   {  return cpt;
   }
}
```


Synchronisation – Comment ça marche ? (1)

- chaque *objet* possède un moniteur (verrou, mutex)
- avant d'entrer dans une méthode / un bloc synchronisé(e), la thread essaye de verrouiller le moniteur de l'objet concerné
  - si le moniteur est déjà verrouillé, la thread est bloquée
  - si le moniteur est disponible, la thread verrouille le moniteur et commence l'exécution de la méthode / du bloc synchronisé(e)

Synchronisation – Comment ça marche ? (2)

- à la fin de l'exécution d'une méthode / un bloc synchronisé(e), le moniteur est déverrouillé et les threads en attente sont réveillées
- si plusieurs threads essayent de verrouiller le même moniteur en même temps, une et une seule entre elles est choisie

Comparaison avec pthreads

- Threads Java

```
class Partage
{   ...
    public synchronized void toto() {...}
    ...
}
...
Partage p = new Partage();
p.toto();
```

- pthreads

```
void toto() {...}
...
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
...
pthread_mutex_lock(&mut);
toto();
pthread_mutex_unlock(&mut);
```


Synchronisation et performance

```
class Partage
{  public synchronized void bidon() {} }

public class PerfTest extends Thread
{  private Partage p;
   PerfTest(Partage p) {super(); this.p = p;}
   public void run()
   {  for (int i = 0; i < 10000000; i++) p.bidon(); }
   public static void main(String[] args)
   {  Partage p = new Partage();
      Thread th1 = new PerfTest(p);
      Thread th2 = new PerfTest(p);
      long t1 = System.currentTimeMillis();
      th1.start(); th2.start();
      try { th1.join(); th2.join(); }
      catch (InterruptedException e) {}
      long t2 = System.currentTimeMillis();
      System.out.println("Temps " + (t2 - t1) + " ms");
   }
}
```


`wait()`

Soit t une thread qui appelle la méthode `wait()` d'un objet o

- t doit posséder le moniteur de o
- t déverrouille le moniteur de o et attend jusque ce que
  - une autre thread appelle la méthode `notifyAll()` de o
  - une autre thread appelle la méthode `notify()` de o et t est la thread choisie parmi les threads en attente
  - t est interrompue par une autre thread
- t essaye de ré-verrouiller le moniteur, en compétition avec des autres threads qui essayent de faire la même chose

`notify()` et `notifyAll()`

Soit t une thread qui appelle la méthode `notify()` ou `notifyAll()` d'un objet o

- t doit posséder le moniteur de o
- `notify()` : une thread parmi celles en attente (exécutant `wait()` de o) est choisie et réveillée (choix dépendant de l'implémentation)
- `notifyAll()` : toutes les threads en attente sont réveillées
- la thread réveillée ne peut pas continuer tant que t tient le moniteur

Comparaison avec pthreads

- Threads Java

```
Object o = new ...;
...
synchronized(o)
{   ...
    o.wait();
    ...
}
```

- pthreads

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
...
pthread_mutex_lock(&mut);
...
pthread_cond_wait(&cond, &mut);
...
pthread_mutex_unlock(&mut);
```


Comparaison avec pthreads

- Threads Java

```
synchronized(o)
{   ...
    o.notify();
    // ou o.notifyAll();
    ...
}
```

- pthreads

```
pthread_mutex_lock(&mut);
...
pthread_cond_signal(&cond);
/* ou pthread_cond_broadcast(&cond); */
...
pthread_mutex_unlock(&mut);
```


Exemple – Sémaphores de Dijkstra

```
class Semaphore
{  private volatile int cpt = 0;
   Semaphore(int n) {cpt = n; }

   public synchronized void P()
      throws InterruptedException
   {  while (cpt == 0) wait();
      cpt--;
   }

   public synchronized void V()
   {  cpt++;
      notifyAll();
   }
}
```


Exemple – baguettes des philosophes

```
class Baguettes
{  private boolean [] dispo;

   Baguettes()
   {  dispo = new boolean[5];
      for (int i = 0; i < 5; i++) dispo[i] = true;
   }

   public synchronized void prendre()
   {  int i = Integer.intValue(Thread.currentThread().getName());
      while (!dispo[i] || !dispo[(i + 1) % 5]) wait();
      dispo[i] = dispo[(i + 1) %5] = false;
   }

   public synchronized void laisser()
   {  int i = Integer.intValue(Thread.currentThread().getName());
      dispo[i] = dispo[(i + 1) %5] = true;
      notifyAll();
   }
}
```