.

Inf 431 – Cours 12

Concurrence

jeanjacqueslevy.net

secrétariat de l'enseignement:
Catherine Bensoussan
cb@lix.polytechnique.fr
Aile 00, LIX
01 69 33 34 67

http://www.enseignement.polytechnique.fr/informatique/

Plan

1. **Etat d'un processus**
2. **Ordonnancement**
3. **Implémentation des sections critiques**
4. **Sémaphores**
5. **Les 5 philosophes**

Bibliographie

Greg Nelson, *Systems Programming with Modula-3*, **Prentice Hall, 1991.**

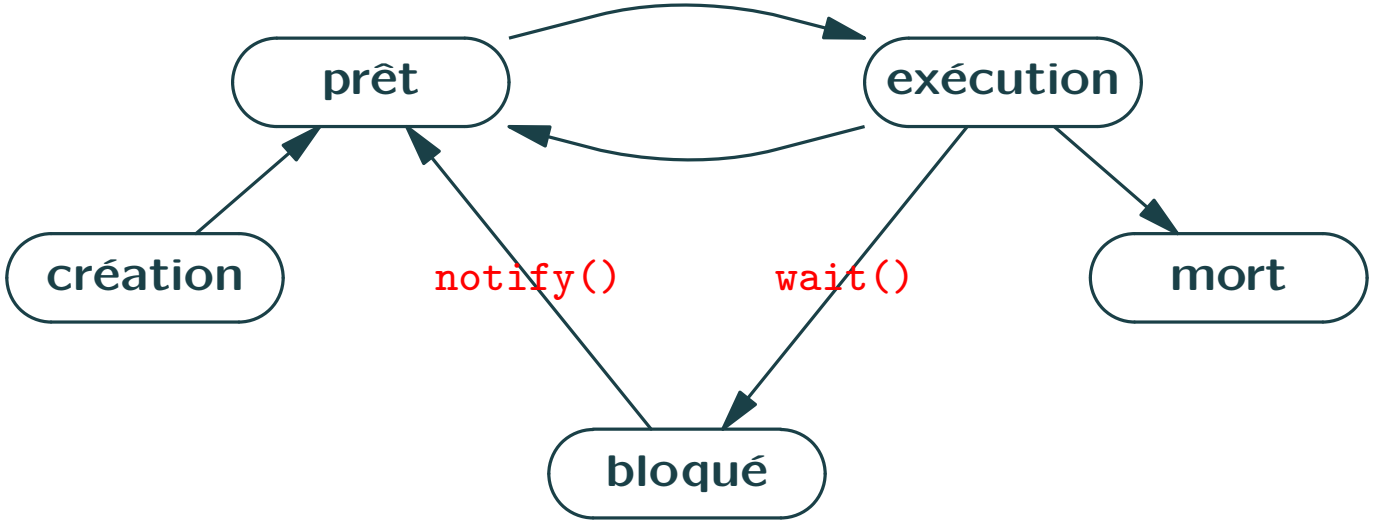
Mordecai Ben-Ari, *Principles of Concurrent Programming*, **Prentice Hall, 1982.**

Fred B. Schneider, *On Concurrent Programming*, **Springer, 1997.**

Jean Bacon, *Concurrent Systems*, **Addison-Wesley, 1998.**

S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quartermann, *The Design and Implementation of the 4.3 BSD Unix Operating System*, **Addison Wesley 1989.**

Etats d'un processus



- Dans l'état **prêt**, un processus peut s'exécuter.
- Dans l'état **exécution**, un processus est en train de s'exécuter (*currentThread*).
- Dans l'état **bloqué**, on attend sur une condition.
- Un signal émis sur une condition peut réveiller le processus et le faire passer dans l'état prêt.

Préemption ou coroutines

- Il peut y avoir plus de processus **prêts** que de processeurs.
- Les processus **prêts** sont rangés dans un ensemble en d'attente.
- Le système (ou la JVM) en choisit un pour l'**exécution**. Lequel?
- C'est le problème de l'**ordonnancement** des processus (*scheduling*).
- Systèmes **préemptifs** autorisent à stopper un processus en cours d'exécution pour le remettre dans l'état prêt.
- Systèmes non préemptifs attendent qu'un processus relache le processeur (par `Thread.yield()`).
- Dans les systèmes non préemptifs, on dit souvent que les processus fonctionnent en *coroutines*.

Transitions d'un processus

- **création → prêt:** `start()`
- **prêt → exécution: ordonnanceur (*scheduler*)**
- **exécution → prêt:** `Thread.yield()` **ou fin d'un quantum de temps (pour systèmes préemptifs)**
- **exécution → bloqué:** `wait()`
- **bloqué → prêt: réveil sur** `notify()` **ou** `notifyAll()`**.**
- **exécution → mort: fin de** `run()`

Ordonnancement

- **Priorités statiques. Les méthodes** *getPriority*. **et** *setPriority* **donnent les valeurs de la priorité d'un processus.**
- **Les priorités suivantes** `MIN_PRIORITY`, `NORM_PRIORITY`, `MAX_PRIORITY` **sont prédéfinies.**
- **En fait, l'ordonnanceur peut aussi affecter des priorités dynamiques pour avoir une politique d'allocation du (des) processeur(s).**
- **La spécification de Java dit qu'en général les processus de forte priorité vont s'exécuter avant les processus de basse priorité.**
- **L'important est d'écrire du code portable, donc de comprendre ce que la JVM garantit sur tous les systèmes.**
- **Ecrire du code contrôlant l'accès à des variables partagées en faisant des hypothèses sur la priorité des processus ou sur leur vitesse relative (*race condition*) est très dangereux.**

Exemples de politiques d'ordonnancement

- **priorité sur l'age. Au début, $p = p_0$ est la priorité statique donnée au processus. Si un processus prêt attend le processeur pendant k secondes, on incrémente p. Quand il s'exécute, on fait $p = p_0$.**

- **priorité décroit si on utilise beaucoup de processeur. (Unix 4.3BSD) Au début $p = p_{nice}$ est la priorité initiale statique. Tous les 40ms, la priorité est recalculée par:**

$$p = PUSER + \left\lceil \frac{p_{cpu}}{4} \right\rceil + 2\ p_{nice}$$

p_{cpu} est incrémenté toutes les 10 ms quand le processus s'exécute. Toutes les secondes, p_{cpu} est aussi modifié par le filtre suivant:

$$p_{cpu} = \frac{2\ load}{2\ load + 1} + p_{nice}$$

qui fait baisser sa valeur en fonction de la charge globale du système $load$ (nombre moyen de processus prêts sur la dernière minute).

Pour Unix, la priorité croit dans l'ordre inverse de sa valeur (0 est fort, 127 est bas).

Synchronisation et priorités

- Avec les verrous pour l'exclusion mutuelle, une inversion de priorités est possible.
- 3 processus P_b, P_m, P_h s'exécutent en priorité basse, moyenne, et haute.
- P_b prend un verrou qui bloque P_h.
- P_m s'exécute et empêche P_b (de priorité plus faible) de s'exécuter.
- le verrou n'est pas relaché et un processus de faible priorité P_m peut empêcher le processus P_h de s'exécuter.
- on peut s'en sortir en affectant des priorités sur les verrous en notant le processus de plus haute priorité attendant sur ce verrou. Alors on fait monter la priorité d'un processus ayant pris ce verrou (Win 2000).
- tout cela est dangereux.

Les lecteurs et les écrivains

Souvent une ressource partagée est lue ou modifiée.

En lecture, la ressource n'est pas modifiée, i processus peuvent lire simultanément les données de cette ressource ($i \geq 0$).

En écriture, le seul processus écrivain peut y accéder et la modifier ($i = -1$).

```
synchronized void accesPartage() {
  while (i < 0)
    wait();
  ++ i;
}

synchronized void retourPartage() {
  -- i;
  if (i == 0)
    notify();
}
```

```
synchronized void accesExclusif() {
  while (i != 0)
    wait();
  i = -1;
}

synchronized void retourExclusif() {
  i = 0;
  notifyAll();
}
```

Les lecteurs et les écrivains (2)

NotifyAll **est pratique, il évite de compter les lecteurs en attente. Pourtant il peut réveiller trop de processus qui vont se retrouver immédiatement bloqués.**

Avec deux conditions, contrôle plus fin:

```
synchronized void accesPartage() {
  ++ nLecteurs;
  while (i < 0)
    lecture.wait();
  -- nLecteurs;
  ++ i;
}

synchronized void retourPartage() {
  -- i;
  if (i == 0)
    ecriture.notify();
}
```

```
synchronized void accesExclusif() {
  while (i != 0)
    ecriture.wait();
  i = -1;
}

synchronized void retourExclusif() {
  i = 0;
  if (nLecteurs > 0)
    lecture.notifyAll();
  else
    ecriture.notify();
}
```

nLecteurs **est le nombre de lecteurs en attente.**

Les lecteurs et les écrivains (3)

Exécuter *notify* à l'intérieur d'une section critique n'est pas très efficace. Avec un seul processeur, ce n'est pas un problème car les réveillés passent dans l'état prêt attendant la disponibilité du processeur.

Avec plusieurs processeurs, le processus réveillé peut retomber rapidement dans l'état bloqué, tant que le verrou n'est pas relaché.

Le mieux est de faire *notify* à l'extérieur de la section critique.

```
void retourPartage() {
  boolean faireNotify;
  synchronized (this) {
    -- i;
    faireNotify = i == 0;
  }
  if (faireNotify)
    ecriture.notify();
}
```

Les lecteurs et les écrivains (4)

Des blocages inutiles sont possibles (avec plusieurs processeurs) sur le *notifyAll* de fin d'écriture.

Comme avant, on peut le sortir de la section critique.

Si plusieurs lecteurs sont réveillés, un seul prend le verrou. Mieux vaut faire *notify* en fin d'écriture, puis refaire *notify* en fin d'accès partagé pour relancer les autres lecteurs.

```
void accesPartage() {
  synchronized (this) {
    ++ nLecteurs;
    while (i < 0)
      lecture.wait();
    -- nLecteurs;
    ++ i;
  }
  lecture.notify();
}
```

Les lecteurs et les écrivains (5)

Famine possible d'un écrivain en attente de fin de lecture. La politique d'ordonnancement des processus peut aider. On peut aussi logiquement imposer le passage d'un écrivain.

```
void accesPartage() {
  synchronized (this) {
    ++ nLecteurs;
    if (nEcrivains > 0)
      lecture.wait();
    while (i < 0)
      lecture.wait();
    -- nLecteurs;
    ++ i;
  }
  lecture.notify();
}
```

```
synchronized void accesExclusif() {
  ++ nEcrivains;
  while (i != 0)
    ecriture.wait();
  -- nEcrivains;
  i = -1;
}
```

Contrôler finement la synchronisation peut être complexe.

Implémentation de la synchronisation

Comment réaliser une opération atomique? Comment implémenter la prise d'un verrou? ou l'attente sur une condition?

Dépend du matériel. Souvent sur un ordinateur, il a une instruction non interruptible *Test and Set*.

`TAS(m)` teste si $m = 0$. Si oui, alors on répond vrai et on met m à 1. Sinon on répond faux.

On programme une section critique ainsi:

```
while (true) {
  while (TAS(m) != 0)
    ;
  // section critique
  m = 0;
}
```

$\Rightarrow$ attente active. On peut partir de cette solution pour construire des systèmes de file d'attente et suspendre les processus.

Algorithme de Peterson

Sans `TAS`, on peut y arriver.

```
class Peterson extends Thread {
  static int tour = 0;
  static boolean[] actif = {false, false};
  int i, j;
  Peterson (int x) { i = x; j = 1 - x; }

  public void run() {
    while (true) {
      actif[i] = true;
      tour = j;
      while (actif[j] && tour == j)
        ;
      // section critique
      actif[i] = false;
  } }

  public static void main (String args[]) {
    Thread t0 = new Peterson(0), t1 = new Peterson(1);
    t0.start(); t1.start();
} }
```

Algorithme de Peterson (2)

Preuve de sureté. (*safety*)

Si P_0 et P_1 tous les deux dans leur section critique. Alors $actif[0] = actif[1] = true$.

Impossible car les deux tests auraient été franchis en même temps alors que $tour$ favorise l'un deux. Donc un des deux est entré. Disons P_0.

Cela veut dire que P_1 n'a pu avoir trouvé le $tour$ à 1 et qu'il n'est par entré en section critique.

Preuve de vivacité. (*lifeness*)

Supposons P_0 bloqué dans le while**.**

Cas 1: P_1 non intéressé à rentrer dans la section critique. Alors $actif[1] = false$. Et donc P_0 ne peut être bloqué par le while**.**

Cas 2: P_1 est aussi bloqué dans le while**. Impossible car $tour$ vaut 0 ou 1. Donc l'un de P_0 ou P_1 ne peut rester dans le** while**.**

Sémaphores

- l'algorithme de Peterson n'est jamais utilisé. Il n'a qu'un intéret théorique.
- l'algorithme de Dekker est une autre forme de protocole pour assurer une section critique.
- comme primitives de plus bas niveau, la litérature de la concurrence considère la notion de sémaphore [Dijkstra 65].
- un sémaphore est une variable s booléenne, sur laquelle on fait deux opérations
  - (prendre "*Proberen*" le sémaphore) $P(s)$ qui teste s et qui si s est $true$ le met à $false$ de manière atomique. Sinon l'instruction attend sur s.
  - (libérer "*Verhogen*" le sémaphore) $V(s)$ réveille un processus en attente sur s s'il existe un tel processus, sinon met s à $true$.

Exercice 1 Généraliser l'algorithme de Peterson à n processus.

Sémaphores et sections critiques

A la différence des (variables) conditions de Java, les sémaphores ne sont pas attachés à un verrou.

On doit donc les utiliser pour programmer des sections critiques.

```
static Semaphore s;
while (true) {
  P(s);
  section critique
  V(s);
}
```

La classe `Semaphore` reste à définir car non fournie en Java.

Exercice 2 **Programmer les conditions de Java avec les sémaphores.**

La file d'attente concurrente

Avec les seuls sémaphores, on peut programmer la file d'attente concurrente de longeur 1.

```
static void ajouter (int x, FIFO f) {
  P(s_vide);
  f.contenu = x;
  f.pleine = true;
  V(s_plein);
}

static int supprimer (FIFO f) {
  int res;
  P(s_plein);
  f.contenu = x;
  f.pleine = true;
  res = f.contenu;
  V(s_vide);
  return res;
}
```

Cas particulier du problème plus général du producteur - consommateur.

Sémaphores généralisés

Pour programmer la file de longueur n, il est plus simple de se servir de la notion de sémaphore généralisé.

Un sémaphore généralisé a une valeur entière positive ou nulle.

- **(prendre la sémaphore) $P(s)$ teste $s > 0$. Si oui, on décrémente s. Sinon l'instruction attend sur s.**
- **(libérer le sémaphore) $V(s)$ réveille un processus en attente sur s s'il existe un tel processus, sinon on incrémente s**

Un sémaphore booléen revient donc à considérer un sémaphore initialisé à 1.

La file d'attente concurrente (2)

Soit n la taille de la file. Au début on fait $s\_libres = 0$ et $s\_occupes = n$.

```
static void ajouter (int x, FIFO f) {
  P(s_libres);
  f.contenu[f.fin] = x;
  f.fin = (f.fin + 1) % f.contenu.length;
  f.vide = false; f.pleine = f.fin == f.debut;
  V(s_occupes);
}

static int supprimer (FIFO f) {
  P(s_occupes);
  int res = f.contenu[f.debut];
  f.debut = (f.debut + 1) % f.contenu.length;
  f.vide = f.fin == f.debut; f.pleine = false;
  V(s_libres);
  return res;
}
```

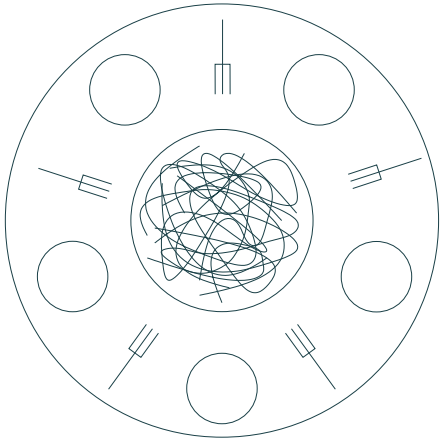
A nouveau une instance du producteur − consommateur.

Les 5 philosophes

Problème classique de [Dijkstra] à tester avec toutes les primitives concurrentes vues (verrous, conditions, sémaphores, sémaphores généralisés).

5 moines philosophes ne font que penser et manger. Pour manger, il vont dans la salle commune où se trouve un plat de spaghettis avec 5 assiettes et 5 fourchettes.

Les spaghettis sont bien longues, il faut deux fourchettes pour manger le plat (il y a pénurie de cuillères dans le monastère, qui ne dispose que de 5 fourchettes au total).



Exercice 3 Comment arriver à ce qu'aucun moine ne meure de faim? et que chacun d'entre eux puissent manger régulièrement?

Exercices

Exercice 4 Implanter les sémaphores booléennes en Java avec les verrous et conditions.

Exercice 5 Idem avec les sémaphores généralisés.

Exercice 6 Dans le langage Ada, la communication se fait par rendez-vous. Deux processus P et Q font un rendez-vous s'ils s'attendent mutuellement chacun à un point de son programme. On peut en profiter pour passer une valeur. Comment organiser cela en Java?

Exercice 7 Une barrière de synchronisation entre plusieurs processus est un endroit commun que tous les processus actifs doivent franchir simultanément. C'est donc une généralisation à n processus d'un rendez-vous. Comment la programmer en Java?

Exercice 8 Toute variable peut être partagée entre plusieurs processus, quel est le gros danger de la communication par variables partagées?

Exercice 9 Un ordonnanceur est juste s'il garantit à tout processus prêt de passer dans l'état exécution. Comment le construire?