Inf 431 – Cours 11

Synchronisation par mémoire partagée

jeanjacqueslevy.net

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

www.enseignement.polytechnique.fr/informatique/IF

Plan

1. Rappels
2. Conditions et moniteurs
3. Etats d'un processus
4. Ordonnancement
5. Les lecteurs et les écrivains
6. Implémentation de la synchronisation

Bibliographie

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

Mordechai 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.

Rappels

- un processus t (*Thread*) est un programme qui s'exécute ;
- t.`start` lance l'exécution concurrente de la méthode t.`run` de t ;
- t.`interrupt` signale qu'on demande l'interruption de t ;
- `isInterrupted` teste si un signal d'interruption est arrivé ;
- `InterruptedException` est levée quand t est en attente (et ne peut pas tester `isInterrupted` ;
- les méthodes synchronized d'un même objet s'exécutent en exclusion mutuelle ;
- `wait`, `notify`, `notifyAll` permettent la gestion concurrente de données.

Conditions et moniteurs (1/7)

- Paradigme de la concurrence $\equiv$ la file d'attente concurrente.
- Pour simplifier, supposons la file f de longueur ℓ t.q. $\ell \leq 1$.
- *ajouter* et *supprimer* s'exécutent de manière concurrente.

```
static synchronized void ajouter (int x, FIFO f) {
  if (f.pleine)
    // Attendre que la file se vide
  f.contenu = x;
  f.pleine = true;
}

static synchronized int supprimer (FIFO f) {
  if (!f.pleine)
    // Attendre que la file devienne pleine
  f.pleine = false;
  return f.contenu;
}
```

Conditions et moniteurs (2/7)

- *ajouter* attend que la file f se vide, si f est pleine.
- *supprimer* attend que la file f se remplisse, si f est vide
- il faut relâcher le verrou pour que l'autre puisse s'exécuter.

- 2 solutions :
  - ressortir de chaque méthode sans se bloquer et revenir tester la file $\Rightarrow$ attente active (*busy wait*) qui coûte cher en temps.
  - atomiquement relâcher le verrou + attendre sur une condition.
    Quand la condition est « notifiée », on repart en reprenant le verrou.

- en Java, une condition est une simple référence à un objet (son adresse). Par exemple, ici la file elle-même f.
- quand on remplit la file, on envoie un signal sur une condition et réveiller un processus en attente sur la condition.

Conditions et moniteurs (3/7)

- *wait* et *notify* sont deux méthodes de la classe *Object*. Tous les objets ont ces deux méthodes.

```
static void ajouter (int x, FIFO f) throws InterruptedException {
  synchronized (f) {
    while (f.pleine)
      f.wait();
    f.contenu = x;
    f.pleine = true;
    f.notify();
} }

static int supprimer (FIFO f) throws InterruptedException {
  synchronized (f) {
    while (!f.pleine)
      f.wait();
    f.pleine = false;
    f.notify();
    return f.contenu;
} } Exécution
```

Conditions et moniteurs (4/7)

- Même programme en programmation par objets.

```
synchronized void ajouter (int x) throws InterruptedException {
    while (pleine)
      wait();
    contenu = x;
    pleine = true;
    notify();
}

synchronized int supprimer () throws InterruptedException {
    while (!pleine)
      wait();
    pleine = false;
    notify();
    return contenu;
}
```
Exécution

Conditions et moniteurs (5/7)

- si plus de deux processus sont en jeu, *notifyAll* réveille tous les processus en attente sur la condition.

```
synchronized void ajouter (int x) throws InterruptedException {
    while (pleine)
      wait();
    contenu = x;
    pleine = true;
    notifyAll();
}

synchronized int supprimer () throws InterruptedException {
    while (!pleine)
      wait();
    pleine = false;
    notifyAll();
    return contenu;
} Exécution
```

Conditions et moniteurs (6/7)

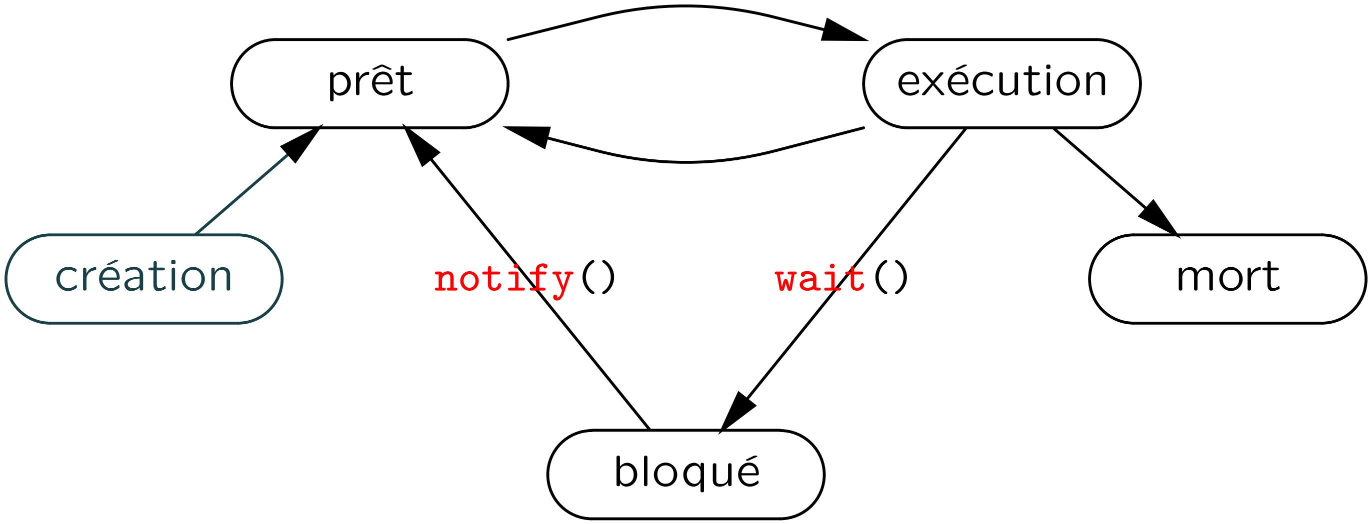
- *notify* n'a pas de mémoire. On réveille un quelconque des processus en attente sur la condition.
- l'action relâchant le verrou et de mise en attente engendrée par *wait* est atomique. Sinon, on peut perdre un signal émis par *notify* avant d'attendre sur la condition.
- de même *notify* est fait avant de relâcher le verrou.
- il faut utiliser un while et non un if dans la section critique, car un autre processus peut être réveillé et invalider le test.

Conditions et moniteurs (7/7)

```
class FIFO {
  int debut, fin; boolean pleine, vide; int[ ] contenu;
  ...
  synchronized void ajouter (int x) throws InterruptedException {
    while (pleine)
      wait();
    contenu[fin] = x;
    fin = (fin + 1) % contenu.length;
    vide = false; pleine = fin == debut;
    notifyAll();
  }

  synchronized int supprimer () throws InterruptedException {
    while (vide)
      wait();
    int res = contenu[debut];
    debut = (debut + 1) % contenu.length;
    vide = fin == debut; pleine = false;
    notifyAll();
    return res;
} } Exécution
```

Etats d'un processus (1/3)



- État prêt : un processus peut s'exécuter.
- État exécution : un processus qui s'exécute (*currentThread*).
- 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.

Etats d'un processus (2/3)

- Il peut y avoir plus de processus prêts que de processeurs.
- Les processus prêts forment un ensemble, souvent géré comme une FIFO ou une file de priorités (*runQ*)
- Le système (la JVM) en choisit un pour l'exécution. Lequel ?
- C'est le problème de l'ordonnancement des processus (*scheduling*).

- Systèmes préemptifs peuvent arrêter un processus en cours d'exécution et le mettre dans l'état prêt.
- Systèmes non préemptifs attendent qu'un processus relâche le processeur (par `Thread.yield()`).
- Dans les systèmes non préemptifs, les processus fonctionnent en *coroutines*.

Etats d'un processus (3/3)

- création → prêt : `start()`

- prêt → exécution : ordonnanceur (*scheduler*)
- exécution → prêt : `Thread.yield()` ou fin d'un quantum de temps (pour systèmes préemptifs)

- exécution → bloqué : `wait()`, début de synchronized
- bloqué → prêt : réveil sur `notify()` ou `notifyAll()`, fin de synchronized

- exécution → mort : fin de `run()`

Ordonnancement (1/2)

- 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 $\Rightarrow$ 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.

Ordonnancement (2/2)

- 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 $p = PUSER + 2p_{nice}$ est la priorité initiale statique.
  Tous les 40ms, la priorité est recalculée par :
  $$p = PUSER + \left\lceil \frac{p_{cpu}}{4} \right\rceil + 2\ p_{nice}$$
  Tous les 10 ms, p_{cpu} est incrémenté quand le processus s'exécute.
  Toutes les secondes, p_{cpu} est aussi modifié par le filtre suivant :
  $$p_{cpu} = \frac{2\ load}{2\ load + 1}\ p_{cpu} + p_{nice}$$
  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 relâché 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 (1/6)

Une ressource partagée est lue ou modifiée concurremment. En lecture, la ressource n'est pas modifiée.

- Plusieurs processus (lecteurs) peuvent lire simultanément la donnée.
- Un seul processus (écrivain) peut modifier la donnée.
- Quand un lecteur s'exécute, aucun écrivain ne peut s'exécuter en parallèle.
- Quand un écrivain s'exécute, aucun autre écrivain, ni aucun lecteur, ne peut s'exécuter.

```
void lecture() {
  accesPartage();
  // lire la donnée partagée
  retourPartage();
}
```

```
void ecriture() {
  accesExclusif();
  // modifier la donnée partagée
  retourExclusif();
}
```

Les lecteurs et les écrivains (2/6)

En lecture, on a $nLecteurs$ simultanés ($nLecteurs > 0$).

En écriture, on a $nLecteurs = -1$.

```
synchronized void accesPartage() {
  while (nLecteurs == -1)
    wait();
  ++ nLecteurs;
}

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

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

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

Les lecteurs et les écrivains (3/6)

notifyAll réveille trop de processus se retrouvant immédiatement bloqués. Avec des variables de condition Posix `cLecture` et `cEcriture`, on a un contrôle plus fin :

```
synchronized void accesPartage() {
  ++ nLecteursEnAttente;
  while (nLecteurs == -1)
    waitPosix(cLecture);
  -- nLecteursEnAttente;
  ++ nLecteurs;
}

synchronized void retourPartage() {
  -- nLecteurs;
  if (nLecteurs == 0)
    notifyPosix(cEcriture);
}
```

```
synchronized void accesExclusif() {
  while (nLecteurs != 0)
    waitPosix(cEcriture);
  nLecteurs = -1;
}

synchronized void retourExclusif() {
  nLecteurs = 0;
  if (nLecteursEnAttente > 0)
    notifyAllPosix(cLecture) ;
  else
    notifyPosix(cEcriture) ;
}
```

Les conditions Posix n'existent pas en Java ! !
Elles existent en C, C++, Ocaml, Modula-3, ADA, etc

Les lecteurs et les écrivains (4/6)

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 relâché.

Il vaut mieux faire *notify* à l'extérieur de la section critique (Ce qu'on ne fait jamais !), ou au moment de la sortie du synchronized :

```
void retourPartage() {
  boolean faireNotify;
  synchronized (this) {
    -- nLecteurs;
    faireNotify = nLecteurs == 0;
  }
  if (faireNotify)
    notifyPosix(cEcriture) ;
}
```

Les lecteurs et les écrivains (5/6)

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) {
    ++ nLecteursEnAttente;
    while (nLecteurs == -1)
      waitPosix(cLecture);
    -- nLecteursEnAttente;
    ++ nLecteurs;
  }
  notifyPosix(cLecture);
}
```

```
synchronized void retourExclusif() {
  nLecteurs = 0;
  if (nLecteursEnAttente > 0)
    notifyPosix(cLecture) ;
  else
    notifyPosix(cEcriture);
}
```

Les lecteurs et les écrivains (6/6)

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) {
    ++ nLecteursEnAttente;
    if (nEcrivainsEnAttente > 0)
      waitPosix(cLecture);
    while (nLecteurs == -1)
      waitPosix(cLecture);
    -- nLecteursEnAttente;
    ++ nLecteurs;
  }
  notifyPosix(cLecture);
}
```

```
synchronized void accesExclusif() {
  ++ nEcrivainsEnAttente;
  while (nLecteurs != 0)
    waitPosix(cEcriture);
  -- nEcrivainsEnAttente;
  nLecteurs = -1;
}
```

Contrôler finement la synchronisation peut être complexe.

Exclusion mutuelle : implémentation (1/4)

- 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 machine ininterruptible *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) == false)
    ;
  // section critique
  m = 0;
}
```

  attente active $\Rightarrow$ beaucoup de temps machine (*spin locks*).
- Parfois, des adresses mémoire particulières servent de verrous. Leur lecture/écriture est atomique (ininterruptible).

Exclusion mutuelle : implémentation (2/4)

- Au lieu de faire une attente active, on range le processus dans un ensemble de processus en attente sur la libération d'un verrou.
- Souvent, cet ensemble W_m de processus en attente est une FIFO. Mais tenir compte que c'est une FIFO est non portable.
- ```
  while (true) {
    while (TAS(m.verrou) == false)
      Ranger currentThread dans W_m ; Attendre sur m.

    // section critique
    m.val = 0;
    Signaler sur m.
  }
  ```
- *Attendre* et *Signaler* sont ici des opérations internes à la JVM.
  ```

Exclusion mutuelle : implémentation (3/4)

- Sans `TAS`, peut-on y arriver ?
- Solution injuste

```
while (true) {
  while (tour != 0)
    ;
  // section critique
  tour = 1;
}
```

```
while (true) {
  while (tour != 1)
    ;
  // section critique
  tour = 0;
}
```

- Solution fausse (cf. la réservation des places d'avions)

```
while (true) {
  while (actif[1])
    ;
  actif[0] = true;
  // section critique
  actif[0] = false;
}
```

```
while (true) {
  while (actif[0])
    ;
  actif[1] = true;
  // section critique
  actif[1] = false;
}
```

Exclusion mutuelle : implémentation (4/4)

- Solution avec interblocage

```
while (true) {
  actif[0] = true;
  while (actif[1])
    ;
  // section critique
  actif[0] = false;
}
```

```
while (true) {
  actif[1] = true;
  while (actif[0])
    ;
  // section critique
  actif[1] = false;
}
```

- Solution correcte possible ?
  ⇒ algorithmes de [Dekker, Peterson]

Algorithme de Peterson (1/5)

```
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 (2/5)

Preuve de sureté. (*safety*)

Si t_0 et t_1 sont 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 seul est entré. Disons t_0.

Cela veut dire que t_1 n'a pu trouver le $tour$ à 1 et n'est pas entré en section critique.

Preuve de vivacité. (*lifeness*)

Supposons t_0 bloqué dans le while.

Cas 1 : t_1 non intéressé à rentrer dans la section critique. Alors $actif[1] = false$. Et donc t_0 ne peut être bloqué par le while.

Cas 2 : t_1 est aussi bloqué dans le while. Impossible car selon la valeur de $tour$, l'un de t_0 ou t_1 ne peut rester dans le while.

Algorithme de Peterson (3/5)

- avec des assertions où on fait intervenir la ligne des programmes c_0 et c_1 (compteur ordinal) exécutée par t_0 et t_1

```
.  public void run() {
.    while (true) {
       {¬ actif[i] ∧ c_i ≠ 2}
1      actif[i] = true;
       {actif[i] ∧ c_i = 2}
2      tour = j;
       {actif[i] ∧ c_i ≠ 2}
3      while (actif[j] && tour == j)
.        ;
       {actif[i] ∧ c_i ≠ 2 ∧ (¬ actif[j] ∨ tour = i ∨ c_j = 2)}
.      // section critique
.      ...
.      // fin de section critique
5      actif[i] = false;
       {¬ actif[i] ∧ c_i ≠ 2}
6      Thread.yield();
.  } }
```

Algorithme de Peterson (4/5)

Preuve de sureté :

- Si t_0 et t_1 sur la ligne 5, on a :

$$\begin{array}{rl} & actif[0] \ \wedge \ c_0 \neq 2 \ \wedge \ (\neg\, actif[1] \vee tour = 0 \vee c_1 = 2) \\ \wedge & actif[1] \ \wedge \ c_1 \neq 2 \ \wedge \ (\neg\, actif[0] \vee tour = 1 \vee c_0 = 2) \end{array}$$

  équivaut à

$$\begin{array}{rl} & actif[0] \wedge c_0 \neq 2 \wedge tour = 1 \\ \wedge & actif[1] \wedge c_1 \neq 2 \wedge tour = 0 \end{array}$$

  équivaut à

$$tour = 0 \wedge tour = 1 \wedge P$$

  implique

$$tour = 0 \wedge tour \neq 0 \wedge P \ \equiv \ \textsf{false}$$

  impossible.

- Preuve par énumération des cas (*model checking*)

Algorithme de Peterson (5/5)

Preuve de vivacité.

- Si t_0 et t_1 dans la boucle while :

$$\begin{aligned} & \mathit{actif}[1] \wedge \mathit{tour} = 1 \wedge \mathit{actif}[0] \wedge \mathit{tour} = 0 \\ \equiv \quad & \mathit{tour} = 0 \wedge \mathit{tour} = 1 \wedge P \\ \equiv \quad & \textsf{false} \end{aligned}$$

- Si t_0 en dehors de la boucle while et t_1 dedans, la preuve se complique car faisant intervenir l'évolution dans le temps (logique temporelle ou modale).
  Par exemple :

  $$\neg\, \mathit{actif}[0] \wedge \mathit{tour} = 0 \wedge \mathit{actif}[0] \wedge c_0 = 2$$

  équivaut à

  $$\neg\, \mathit{actif}[0] \wedge \mathit{tour} = 0 \wedge c_0 = 2$$

  alors le programme évolue vers $\mathit{tour} = 1$ et le tout devient faux. On quitte donc la boucle while.

Exercices

Exercice 1 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 2 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 3 Toute variable peut être partagée entre plusieurs processus, quel est le gros danger de la communication par variables partagées ?

Exercice 4 Un ordonnanceur est juste s'il garantit à tout processus prêt de passer dans l'état exécution. Comment le construire ?

Exercice 5 (difficile) Généraliser l'algorithme de Peterson au cas de n processus ($n \geq 2$).