# Résumé du chapitre 3
# Synchronisation entre processus : de producteur/consommateur aux sémaphores

Jacques Mossière

22 septembre 2008

## 1 Introduction

Nous étudions dans ce chapitre la réalisation des processus et des mécanismes de synchronisation. Nous commençons par examiner comment faire exécuter en pseudo-parallélisme sur un ordinateur un ensemble de processus. Nous ne présentons pas une réalisation directe des moniteurs ; nous préférons introduire un mécanisme de synchronisation plus simple, les sémaphores, mécanisme qui permet de programmer facilement des moniteurs. La question de l'exclusion mutuelle est une question clé de la réalisation des noyaux. Nous y consacrons le paragraphe 5.

## 2 Synchronisation entre processus

### 2.1 Le modèle du producteur et du consommateur

#### 2.1.1 Le problème

De nombreuses applications sont construites sur le modèle client serveur. Dans ces applications, un ou plusieurs processus, les clients, envoient des requêtes à un ou plusieurs processus serveurs. Les serveurs doivent récupérer les requêtes stockées dans un tampon intermédiaire, les traiter et envoyer une réponse au client. Nous présentons ici le modèle d'interaction sous-jacent à toutes les relations client serveur, le modèle du producteur et du consommateur.

Dans ce modèle, un processus, le producteur, envoie des messages à l'intention d'un autre processus, le consommateur. Les cadences de production et de consommation peuvent varier au cours du temps. Pour augmenter la probabilité d'exécution parallèle du producteur et du consommateur, ceux-ci communiquent par l'intermédiaire d'un tampon de `N` cases, chaque case pouvant contenir un message. Dans ces conditions, le producteur peut déposer un message s'il existe au moins une case libre dans le tampon ; si ce n'est pas le cas, il se bloque pour être réveillé par le consommateur lorsque ce dernier a vidé une case. De façon symétrique, le consommateur peut retirer un message si au moins une case est pleine ; si ce n'est pas le cas, il se bloque et sera réveillé par le producteur après un dépôt de message.

### 2.1.2 Spécification du problème de synchronisation

Les points de synchronisation sont, pour le producteur, le début du dépôt de message et, pour le consommateur, le début du retrait de message. On peut décrire l'état du système en introduisant une variable entière `n` comptant le nombre de cases occupées du tampon.

- condition pour déposer : `n<N`
- condition pour retirer : `n>0`

### 2.1.3 Réalisation avec un moniteur

Les procédures `deposer` et `retirer` vont appartenir à un même moniteur `prod_cons`. Dans ce moniteur seront définies la variable `n`, initialement à `0` et deux conditions, `cprod` et `ccons` destinées à bloquer le producteur et le consommateur. Il vient :

```
prod_cons : moniteur ;
var n : entier ; cprod, ccons : condition ;
procedure deposer (m) ;
   debut
   si n=N alors cprod.attendre ;
   dépôt du message ;
   n := n+1 ;
   ccons.signaler
   fin ;
procedure retirer (m) ;
   debut
   si n=0 alors ccons.attendre ;
   retrait du message ;
   n := n−1 ;
   cprod.signaler
   fin ;
n := 0 ;            /* initialisation */
fin prod_cons ;
```

Les programme des processus producteur et consommateur peuvent être les suivants :

```
Producteur :

tantque vrai faire
   debut
   phase de calcul
   prod_cons.deposer (msg)
   fin ;
```

```
Consommateur :

tantque vrai faire
   debut
   prod_cons.retirer (mess) ;
   phase de traitement
   fin ;
```

**Remarque.** Dans le moniteur `prod_cons`, on effectue systématiquement un `signaler` à l'intention de l'autre processus : comme `signaler` ne fait rien si aucun processus n'est en attente, il est superflu de tester l'état de l'autre processus (ce qui demanderait des variables d'état supplémentaires).

### 2.1.4 Gestion du tampon

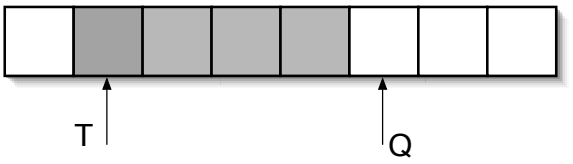


FIG. 1 – Tampon circulaire

Nous n'avons pas détaillé ci-dessus la gestion du tampon. Si le tampon est un tableau `BUF` de `N` cases numérotées de `0` à `N-1`, on peut le gérer de façon circulaire comme indiqué sur la figure 1.

Soit `T` et `Q` deux variables entières initialisées toutes deux à `0` (ou à n'importe quelle valeur comprise entre `0` et `N-1`). Le dépôt du message `m` s'écrit simplement

```
BUF[Q] := m; Q := Q+1 mod N;
```

Et le retrait d'un message dans une variable `msg` s'écrit de façon symétrique

```
msg := BUF[T]; T := T+1 mod N;
```

**Remarque.** On appelle souvent boîte aux lettres un tampon utilisé par des processus selon le schéma producteur consommateur.

# 3 Machine à processus

Nous avons étudié au chapitre 2 la notion de processus. Dans un système à processus parallèles, tout doit se passer, à des questions d'efficacité près, comme si l'on disposait d'un processeur par processus. Nous montrons dans ce paragraphe les principes de cette réalisation ; la structure complète d'un noyau de gestion de processus sera présentée au paragraphe **??**.

## 3.1 Processus et table des processus

Rappelons que l'état d'un programme en cours d'exécution est entièrement défini par :

- un espace de mémoire contenant son code, ses données et sa pile,
- les valeurs contenues dans les registres généraux du processeur, et en particulier du compteur ordinal et du registre d'état. Nous appellerons par la suite mot d'état de programme (mep) l'ensemble du compteur ordinal et du registre d'état.

**Remarque.** Cette définition suppose qu'on ne s'intéresse à l'état d'un programme qu'en des points observables séparant l'exécution des instructions les unes à la suite des autres. Dans le cas de processeurs permettant du pipe-line ou des anticipations, elle suppose que le matériel est capable de construire un état cohérent.

Nous admettons ici que les processus à exécuter sont en nombre fixe, que l'espace mémoire de chacun d'eux, dans lequel on a chargé code et données et alloué l'espace de la pile, est réservé une fois pour toutes en mémoire centrale. Le principe de réalisation d'une machine à processus est alors simple :

- quand un processus ne s'exécute pas, les valeurs de ses registres sont conservées dans un **descripteur** associé au processus ;
- arrêter un processus consiste à recopier les registres du processeur dans le descripteur du processus arrêté ;
- lancer un processus consiste à charger dans les registres du processeur les valeurs conservées dans le descripteur du processus.

On appelle **table des processus** l'ensemble des descripteurs des différents processus. Chaque processus est désigné par un identificateur, son pid, permettant de localiser le descripteur dans la table des processus.

Il reste à examiner comment déclencher les opérations d'arrêt du processus en cours et de lancement d'un nouveau processus. On utilise pour cela une horloge externe au processeur qui déclenche une interruption, dite interruption horloge.

## 3.2 Interruptions, horloge et interruption horloge

### 3.2.1 Les interruptions

Rappelons qu'un signal d'interruption est un signal qui provoque la sauvegarde du mot d'état de programme, soit dans une pile, soit dans un emplacement fixe, puis le chargement du mot d'état de programme à partir d'un emplacement en mémoire centrale associé à la cause ayant déclenché l'interruption.

Un signal d'interruption force donc un branchement à un programme associé à l'interruption ; ce programme est appelé le traitant de l'interruption. Quand le traitant de l'interruption se termine, il peut soit revenir au programme qu'il a interrompu, soit lancer l'exécution d'un nouveau programme.

Il est possible de différer le lancement d'un traitant en utilisant ce qu'on appelle le masquage des interruptions : quand les interruptions sont masquées, un signal d'interruption qui arrive est mémorisé ; le changement de mot d'état de programme ne sera déclenché qu'à l'instant où les interruptions seront démasquées. Concrètement, le masquage ou le démasquage sont représentés par un bit du registre d'état ; des instructions permettent de changer la valeur de ce bit. En règle générale, un traitant d'interruption s'exécute interruptions masquées.

L'étude détaillée des mécanismes de changement d'état et de leur utilisation dans les systèmes d'exploitation sera effectuée au chapitre 4.

### 3.2.2 Horloge et interruption horloge

Les calculateurs disposent d'une ou plusieurs horloges permettant de déclencher une interruption au bout d'un certain intervalle de temps. Une horloge fonctionne typiquement de la façon suivante : un emplacement en mémoire peut être chargé par programme à une valeur positive ; il est alors diminué de 1 avec une périodicité dépendant de l'horloge ; lorsque la valeur passe de 1 à 0, une interruption particulière, l'interruption horloge, est déclenchée et provoque l'exécution du traitant de l'interruption horloge.

## 3.3 Principes de réalisation de la machine à processus

À un instant donné, un processus s'exécute, dont le `pid` est contenu dans une variable globale du système que nous appellerons `proelu`. Les descripteurs des processus éligibles sont chaînés

dans une file d'attente, dite file des éligibles.

Lors du lancement de l'exécution d'un processus, l'horloge est chargée avec une valeur correspondant au temps maximum pendant lequel le processus a le droit de s'exécuter sans être arrêté. Lorsque le traitant de l'interruption horloge est déclenché, il commence par ranger les registres généraux et le mot d'état du processeur dans le descripteur du processus élu et il chaîne ce descripteur en queue de la file des éligibles. Le traitant appelle alors une procédure chargée de définir le nouveau processus élu (en général le premier de la file des éligibles), de ranger son `pid` dans `proelu`, et de charger registres généraux et mot d'état du processeur à partir du descripteur du nouveau `proelu`. Avant de lancer concrètement `proelu`, c'est-à-dire de charger le registre `mep` du processeur, cette procédure fixe la valeur de l'horloge.

De même, lorsqu'un processus se bloque, le programme qui gère le blocage exécute les mêmes actions que ci-dessus pour définir et lancer le nouveau processus élu.

# 4 Les sémaphores

## 4.1 Définition

Un sémaphore est une structure de données composée d'un compteur entier et d'une file d'attente.

Soit `s` un sémaphore ; nous noterons `s.c` son compteur et `s.f` sa file d'attente.

Sur un sémaphore `s` sont définies deux instructions, qui s'exécutent en exclusion mutuelle, notées `P(s)` et `V(s)`. Ces deux instructions sont définies comme suit.

```
P(s) : s.c -- ; si s.c<0 alors le processus qui exécute P est bloqué dans s.f
V(s) : s.c ++ ; si s.c<=0 alors activation d'un processus de s.f
```

**Remarque.** La définition que nous avons donnée pour `V` implique qu'il y a au moins un processus dans `s.f` lorsque `s.c` est négatif. En raisonnant par récurrence, on peut vérifier que c'est bien le cas dès qu'un sémaphore est créé avec une valeur initiale positive ou nulle, la valeur absolue de `s.c` donnant alors le nombre de processus bloqués dans `s.f`.

## 4.2 Exemples d'utilisation

Nous donnons ici les solutions, en termes de sémaphores, aux problèmes de synchronisation vus au chapitre 2. Nous ajoutons un dernier exemple pour montrer comment programmer des sections critiques.

### 4.2.1 Lecture écriture

Soit `s1` un sémaphore de valeur initiale `0`.

```
Processus de calcul                 Processus d'impression

calcul ;                            P(s1) ;
dépôt du résultat dans le tampon ;  retrait du résultat ;
V(s1) ;                             affichage ;
```

Si le processus d'impression démarre en premier, il se bloque sur `P(s1)` et sera réveillé lorsque le processus de calcul exécutera `V(s1)`. Inversement, si le processus de calcul se déroule en premier, l'exécution de `V(s1)` a pour effet d'augmenter de `1` la valeur de `s1.c`. Autrement dit, la valeur d'un sémaphore « mémorise » le nombre de `V` qui ont été effectués sur le sémaphore et on n'a plus besoin d'introduire un booléen comme dans la solution avec moniteur.

### 4.2.2 Rendez-vous de 2 processus

Soit `s1` et `s2` deux sémaphores de valeur initiale `0`. À leur point de rendez-vous, les deux processus exécutent

```
V(s2) ;                    V(s1) ;
P(s1) ;                    P(s2) ;
```

L'exécution de `V` « mémorise » l'arrivée du processus et l'exécution de `P` provoque éventuellement le blocage dans l'attente de l'autre processus.

### 4.2.3 Producteur consommateur

Les deux processus communiquent par l'intermédiaire d'un tampon de `N` cases. Soit `sprod` (valeur initiale `N`) et `scons` (valeur initiale `0`) deux sémaphores. Les procédures deposer et retirer deviennent :

```
procedure deposer ;
   debut
   P(sprod) ;
   dépôt du message ;
   V(scons)
   fin ;
```

```
procedure retirer ;
   debut
   P(scons) ;
   retrait du message ;
   V(sprod)
   fin ;
```

Le sémaphore `sprod`, dont le compteur comptabilise le nombre de cases libres, permet de bloquer le producteur lorsque le tampon est plein ; le sémaphore `scons` compte les cases occupées et permet de bloquer le consommateur.

### 4.2.4 Exclusion mutuelle

Soit un ensemble de processus dont l'exécution se déroule en deux phases, une phase normale (c'est-à-dire sans contrainte vis à vis des autres processus) et une phase critique :

- à un instant donné, un processus au plus peut être dans sa phase critique ;
- si plusieurs processus souhaitent entrer en même temps dans leur section critique, un seul d'entre eux doit s'y engager et les autres doivent se bloquer ;
- un blocage d'un processus hors de sa section critique ne doit pas perturber le fonctionnement des autres processus ;
- l'ordre dans lequel les processus accèdent à leur section critique ne doit pas être imposé.

Pour programmer les entrées et sorties de section critique avec des sémaphores, on peut introduire un sémaphore `mutex`, de valeur initiale `1`.

L'entrée en section critique se résume à exécuter `P(mutex)`, la sortie à `V(mutex)`.

**Remarque 1.** Une erreur classique de la programmation avec des sémaphores consiste à oublier de relâcher les sections critiques.

**Remarque 2. Interblocage**. Supposons que deux processus aient besoin de deux sections critiques imbriquées dont l'accès est contrôlé par deux sémaphores `mutex1` et `mutex2`, de valeur initiale `1`. Que se passe-t-il si les programmmes sont les suivants ?

```
Processus  1                    Processus 2

P(mutex1) ;                     P(mutex2) ;
P(mutex2) ;                     P(mutex1) ;
Section critique ;              Section critique ;
V(mutex2) ;                     V(mutex1) ;
V(mutex1) ;                     V(mutex2) ;
```

Isolément, chacun des processus est correct. L'exécution simultanée des deux processus peut conduire à un blocage infini dès que le processus `1` a pu franchir le `P(mutex1)` et le processus `2` le `P(mutex2)`. Une telle situation est appelée un interblocage.

# 5 Exclusion mutuelle

Dès qu'un système fait intervenir des exécutions parallèles de processus qui partagent des données en lecture et en écriture, on doit éviter que des accès incontrôlés puissent conduire à des valeurs incohérentes des données (penser aux systèmes de réservation de place). Une solution à ce problème consiste à ne faire exécuter qu'un seul processus à la fois pendant les phases critiques ou, ce qui revient à peu près au même, à rendre indivisibles les séquences d'instructions d'accès aux données sensibles (penser à l'affectation $x = x + 1$ qui peut être compilée comme trois instructions). Pour prendre en compte cette exigence, nous avons admis que les procédures d'un moniteur s'exécutaient en exclusion mutuelle, tout comme les opérations `P` et `V`. Il nous reste à rendre crédibles ces hypothèses, c'est à dire à montrer concrètement comment réaliser cette exclusion mutuelle sur les ordinateurs courants. Nous allons examiner tout d'abord le cas de l'ordinateur monoprocesseur où le masquage des interruptions suffit pour assurer l'indivisibilité ; nous montrerons ensuite comment réaliser une exclusion mutuelle par une boucle d'attente aux entrées de section critique ; nous terminerons en montrant comment la conjugaison du masquage des interruptions et de l'introduction d'une instruction spéciale permet de résoudre le problème dans le cas des multiprocesseurs à mémoire commune.

**Remarque.** Le cas des systèmes répartis sur plusieurs calculateurs sans mémoire commune est laissé de côté.

## 5.1 Masquage des interruptions

Dans un système monoprocesseur, et en l'absence d'interruption, un processus s'exécute de façon continue, instruction après instruction. Autrement dit, en l'absence d'interruption, le problème de l'exclusion mutuelle pour l'accès aux données n'existe pas. En conséquence, une façon simple et brutale d'assurer l'indivisibilité d'une suite d'instructions consiste à masquer les interruptions au début de la séquence critique et à les démasquer à la fin.

**Remarque 1.** Le masquage ou le démasquage des interruptions ne peuvent se faire que si l'ordinateur s'exécute en mode maître. Dans un processus utilisateur qui s'exécute en mode esclave, il faut impérativement appeler les séquences à masquer par des appels système permettant le passage en mode maître.

**Remarque 2.** Pour éviter la perte de signaux d'interruptions, les séquences masquées ne peuvent pas être trop longues. C'est en particulier pour cela qu'on réalise l'exclusion mutuelle en deux niveaux :

- exclusion mutuelle par masquage d'interruptions pour les opérations P et V,
- exclusion mutuelle globale à l'aide de sémaphores.

## 5.2 Exclusion mutuelle par attente active

Considérons maintenant le cas d'un multiprocesseur à mémoire commune. Plusieurs processus (un par processeur) pouvant s'exécuter en même temps, la solution précédente n'est plus valide. En revanche, l'existence d'une mémoire commune fournit deux opérations élémentaires indivisibles : la lecture et l'écriture d'une même cellule de mémoire. On a donc cherché à utiliser cette indivisibilité pour réaliser l'exclusion mutuelle.

L'idée consiste à utiliser des variables en mémoire pour décrire l'état des processus (présence d'un processus en section critique par exemple). Quand un processus veut entrer en section critique, il teste ces variables d'état et se met dans une boucle d'attente s'il n'est pas autorisé à y entrer. Si le principe est simple, la mise en œuvre est délicate. Considérons par exemple la tentative de solution ci-dessous. Soit `c` une variable booléenne, initialement à `faux` ; l'idée est de donner à `c` la valeur `vrai` si un processus est en section critique. Le programme de chaque processus est alors le suivant.

```
tantque c faire /* rien */ ;  /* répète le test jusqu'à c=faux  */
c := vrai ;
section critique ;
c := faux ;
```

Dans cette solution, deux processus peuvent tester la valeur de `c`, constater qu'elle est à `faux` et décider d'entrer en section critique, le problème venant de la séparation entre le test de la valeur de `c` effectué dans le tantque et l'affectation de la valeur vrai à `c`. Il est très facile de développer des solutions fausses, soit parce qu'elles ne garantissent pas l'exclusion mutuelle, soit qu'elles présentent des risques d'attente infinie à l'entrée de la section critique, soit qu'elles imposent un ordonnancement pour l'accès des différents processus à la section critique.

La première solution correcte à l'exclusion mutuelle par attente active, dans le cas de deux processus, a été introduite par Dekker (1965) et simplifiée par Peterson (1981). Nous la présentons ci-après. Soit `moi` et `lui` les pid des deux processus ; chaque processus indique dans une variable `intention` son souhait d'entrer en section critique ; une variable commune, `tour`, évite les attentes infinies à l'entrée de la phase critique. Chaque processus exécute le programme suivant.

```
intention[moi] := vrai ;
tour := moi ;
tantque (tour=moi) et (intention[lui]) faire /* rien */ ;
```

```
section critique ;
intention[moi] := faux ;
```

Lorsque les deux processus sont hors section critique, les variables `intention` ont toutes deux la valeur `faux`. Lorsqu'un seul processus tente d'entrer en section critique, il met à `vrai` sa variable `intention` et « se donne le tour ». Si l'autre processus est en section critique, celui qui tente d'entrer attend dans le tantque la sortie du second ; si la section critique est disponible, il y entre immédiatement. En cas de tentative d'entrée simultanée, où les deux processus exécutent en parallèle le tantque, la variable `tour` est utilisée pour permettre à un seul processus d'entrer en section critique (celui dont le pid n'est pas dans `tour`), l'autre se mettant en boucle d'attente.

La complexité de la solution ci-dessus est une conséquence directe de nos hypothèses : seules sont indivisibles la lecture et l'écriture d'une cellule de mémoire. Entre le test d'une variable et sa modification par un processus, l'autre processus peut aussi accéder à la même variable. Les constructeurs de multiprocesseurs ont donc introduit une instruction spéciale permettant d'assurer de façon indivisible le test et la modification d'une variable. C'est l'utilisation de ce type d'instruction que nous présentons maintenant.

## 5.3 Exclusion mutuelle par test and set

Supposons que notre ordinateur préféré contienne l'instruction suivante

```
Test&Set adr
```

Cette instruction réalise, de façon indivisible, le positionnement des codes de condition conformément à la valeur de la cellule de mémoire d'adresse `adr` puis force dans cette cellule une valeur positive. La programmation d'une attente active pour l'entrée en section critique devient triviale : on utilise une cellule de mémoire ayant la valeur `0` si aucun processus n'est en section critique, et une valeur positive sinon. Chaque processus exécute (en pseudo-assembleur)

```
entree :        Test&Set                adr
                jnz                     entree
                section critique
                move                    #0,adr
```

L'indivisibilité de l'instruction `Test&Set` garantit qu'en cas de conflit à l'entrée un seul processus trouvera dans adr une valeur initiale égale à `0`.

Pour réaliser des sections critiques courtes sur un multiprocesseur, on peut combiner cette solution avec le masquage des interruptions : le masquage des interruptions, fait en premier lieu, garantit qu'une fois entré en section critique, le processeur ira jusqu'au bout de la section ; une boucle d'attente sur `Test&Set` règle les conflits d'accès entre processeurs.

**Remarque.** Les constructeurs ont introduit différents types d'instructions offrant les mêmes fonctions que l'instruction présentée ici. Par exemple, l'intel pentium dispose d'une instruction appelée `XCHG` qui échange de façon indivisible les contenus d'un registre et d'un emplacement de mémoire. Il suffit de placer dans le registre une valeur positive, puis de tester la valeur retournée par `XCHG` pour réaliser le même effet qu'un `test&Set`.