

# Systemes informatiques et applications concurrentes

Partie 5 - SGF - Paradigmes de la concurrence

Ivan KURZWEG

24 mai 2010

**Systemes informatiques et applications concurrentes**

by Ivan KURZWEG

Copyright © 2010 Ivan KURZWEG, [ivan.kurzweg@gmail.com](mailto:ivan.kurzweg@gmail.com), 2010

Permission to use, copy, modify, and distribute this documentation *for any purpose with or without fee* is here by granted, provided that *the above copyright notice and this permission notice appear in all copies*.

The documentation is provided "as is" and the author disclaims all warranties with regard to this documentation including all implied warranties of merchantability and fitness. In no event shall the author be liable for any special, direct, indirect, or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this documentation.

## Table des matières

<b>1</b>	<b>Notions de bases</b>	<b>1</b>
1.1	Modèle d'applications et de processus . . . . .	1
1.1.1	Applications concurrentes . . . . .	1
1.1.2	Rappels sur les processus . . . . .	1
1.2	Interblocage . . . . .	1
1.3	Famine . . . . .	2
<b>2</b>	<b>Paradigmes</b>	<b>2</b>
2.1	Exclusion mutuelle . . . . .	2
2.2	Producteurs - consommateurs . . . . .	4
2.3	Lecteurs rédacteurs . . . . .	4
2.4	Diner des philosophes . . . . .	4
<b>3</b>	<b>Exclusion mutuelle</b>	<b>5</b>
3.1	Exclusion mutuelle avec attente active . . . . .	5
3.1.1	Algorithme de Dekker . . . . .	5
3.1.2	Algorithme de Peterson . . . . .	6
3.1.3	Inconvénients de l'attente active . . . . .	7
3.2	Exclusion mutuelle avec attente passive . . . . .	7
3.2.1	Sémaphores . . . . .	7
3.2.2	Moniteurs . . . . .	7
<b>4</b>	<b>Exercices</b>	<b>7</b>
4.1	Conditions de concurrence : exclusion mutuelle (Source : Ivan Boule) . . . . .	7
4.1.1	Questions . . . . .	8
4.2	Algorithme de Dekker (Source Ivan Boule, Cnam) . . . . .	8
4.2.1	Question 1 . . . . .	9
4.2.2	Question 2 . . . . .	9
4.2.3	Question 3 . . . . .	9
4.2.4	Question 4 . . . . .	9
<b>5</b>	<b>Corrections de la partie 4</b>	<b>9</b>
<b>6</b>	<b>Références</b>	<b>10</b>
6.1	Références . . . . .	10

### Résumé

Partie 5 du cours "Systèmes informatiques et applications concurrentes". Cette cinquième partie du cours pose les bases des problèmes soulevées par la programmation concurrente.

## 1 Notions de bases

Un paradigme de programmation est un style fondamental de programmation informatique qui traite de la manière dont les solutions aux problèmes doivent être formulées dans un langage de programmation (à comparer à la méthodologie, qui est une manière de résoudre des problèmes spécifiques de génie logiciel). Un paradigme de programmation fournit (et détermine) la vue qu'a le développeur de l'exécution de son programme (source Wikipedia). Dans ce cadre, nous allons dans le premier chapitre rappeler quelques notions de base, et (re)définir les deux problèmes majeurs engendrés par la programmation concurrente : l'interblocage et la famine.

### 1.1 Modèle d'applications et de processus

On distingue généralement trois types de concurrences :

- *disjointe* : pas de communication ni d'interactions
- *compétitive* : l'accès à certaines ressources (CPU, E/S, mémoire, ..) est mis en compétition entre les applications.
- *coopérative* : au sein d'une application, des processus coopèrent, il y a interaction : des échanges ont lieu entre les processus. .

#### 1.1.1 Applications concurrentes

Une application concurrente est une application non séquentielle : elle est découpée en un ensemble de processus séquentiels, s'exécutant en parallèle, et capables de communiquer entre eux. On introduit ainsi de l'indéterminisme dans l'ordre d'exécution, la politique d'ordonnancement des processus étant généralement inconnue, et sous la responsabilité du noyau du système d'exploitation (cf. partie 2 du cours) : si dans un programme séquentiel (non concurrent), l'ordre des instructions élémentaires est un *ordre total*, qui reste identique à chaque exécution, dans un programme concurrent, l'exécution forme un *ordre partiel*.

Malgré l'augmentation de la complexité de la programmation concurrente, il est intéressant de la mettre en oeuvre dans de nombreux cas, dont les suivants :

- *pour améliorer l'utilisation de la machine* : sur une machine monoprocesseur en particulier, on peut ainsi par exemple "paralléliser" les entrées/sorties et les calculs
- dans le cas d'une architecture parallèle, on peut lancer *concurrentement des sous-calculs*, en accélérant ainsi le temps de réponse
- pour utiliser des *architecture multiprocesseurs*, ou des architectures réparties
- utiliser des *ressources de calculs à distance* (via le réseau, local ou Internet)
- gérer des utilisateurs mobiles
- ...

Nous le verrons dans les paragraphes suivants, la programmation concurrente engendre deux problèmes importants : l'**interblocage** et la **famine**. Pour prévenir et résoudre ces problèmes, la programmation concurrente utilise des techniques de synchronisation des processus : les sémaphores, les moniteurs (nous les verrons dans les chapitres suivants), ou encore les *messages* permettent ainsi de vérouiller l'accès à certaines données, de manière à exprimer un comportement correct des processus. Ces techniques s'appliquent à une série de paradigmes de programmation et d'algorithmes y correspondant, que nous verrons au chapitre 2.

#### 1.1.2 Rappels sur les processus

Nous l'avons vu dans la partie 2 du cours, un processus  
Dans l'ensemble

## 1.2 Interblocage

Un interblocage (*deadlock* ou *étreinte fatale*) se produit quand deux ou plus processus concurrents s'attendent mutuellement, le plus souvent sur une libération de ressources. Les processus bloqués dans cet état le sont définitivement.

Nous verrons dans la partie 8 du cours plus précisément les interblocages, mais nous pouvons définir dès maintenant les points suivants :

- Les conditions nécessaires à l'interblocage sont :
  - Des ressources allouées de manière exclusive à un processus (exclusion mutuelle)
  - Une allocation dynamique et une accumulation possible des ressources
  - Pas de préemption des ressources acquises (seul le processus peut libérer sa ressource)
  - Un attente circulaire : un processus  $P_0$  qui attend un processus  $P_1$  qui attend un processus  $P_2$  ...  $P_n$  qui attend  $P_0$
- Il existe des interblocages dits :
  - *Actifs (deadlocks)* : les processus en attente de ressources sont bloqués par l'allocateur
  - *Passifs (livelocks)* : les processus se bloquent eux-même en testant la disponibilité de la ressource

A ces deux catégories, on peut ajouter des interblocages non liés aux ressources (plus rares), et dûs à une mauvaise synchronisation : par exemple deux processus attendant chacun que l'autre ait terminé sa tâche.

### 1.3 Famine

Nous avons déjà abordé la notion de famine dans la partie 2 du cours consacrée à l'ordonnancement. Dans ce cadre, si l'ordonnancement des processus n'était pas équitable, un processus pouvait être perpétuellement en l'état prêt, sans jamais pouvoir être élu.

Appliqué à la programmation concurrente, ce concept représente les cas où un processus est indéfiniment en attente d'une ressource, sans pour cela être bloqué.

## 2 Paradigmes

Selon la définition du Larousse un paradigme est "*un choix de problèmes à étudier et des techniques propres à leur étude*". Un paradigme de programmation correspond ainsi à une manière d'aborder un problème de programmation. Dans le cas de la programmation concurrente, on distingue généralement quatre *patterns* (patrons, modèles de base) : l'exclusion mutuelle, les producteurs / consommateurs, les lecteurs / rédacteurs, et le dîner des philosophes. Chaque application concurrente doit donc résoudre ses problèmes de synchronisation et de partage de ressources en implémentant un ou plusieurs de ces *patterns*, que nous allons décrire dans les paragraphes suivants :

### 2.1 Exclusion mutuelle

L'exclusion mutuelle est le principe selon lequel plusieurs processus partageant des objets ne peuvent être dans une section critique en même temps. On appelle une *section critique* une partie du programme où peut se produire un conflit d'accès sur une ressource partagée.

**Exemple 2.1** Exemple de section critique avec ressource partagée

On considère dans une application bancaire un compte simplement implémenté par un fichier contenant le solde du compte. Si l'on définit la procédure qui permet de créditer un compte comme :

```

Procédure crediter_compte( entier numero_client, entier somme)
entier solde ;
debut
    /* lecture du solde dans le fichier du client */
    solde=lire_dans_fichier(numero_client);
    solde = solde + somme ;
    /* écrire dans le fichier le nouveau solde */
    ecrire_dans_fichier(numero_client, solde) ;
fin;

```

Sur un système en temps partagé, que se passe-t'il si deux processus tentent d'exécuter cette procédure "simultanément" ?

<pre> P0 : crediter_compte( 1233, 1000) solde = lire_dans_fichier(1233) ; /* P0 bloqué car P1 s'exécute */  /* P0 reprend son exécution */ solde=solde+1000 ; Ecrire_dans_fichier(1233,solde) ; </pre>	<pre> P1 : crediter_compte( 1233, 500) /* P1 est bloqué, P0 s'exécute */  /* P1 reprend son exécution */ solde = lire_dans_fichier(1233) ; /* P1 bloqué car P0 s'exécute */  /* P1 reprend son exécution */ solde=solde+500 ; Ecrire_dans_fichier(1233,solde) ; </pre>
--	--

On voit que dans ce cas précis, les deux processus s'exécutent de manière concurrente sur le système d'exploitation. Le cas présenté peut donc se produire : le "mauvais" enchaînement des processus provoque une écriture finale dans le fichier de 500 euros au lieu des 1500 dûs !

**Exemple 2.2** Exemple de section critique avec partage de variable

Soit deux processus (threads) partageant une même variable `compte` initialisée à 0, et dont le code est le suivant :

<pre> processus_P entier x = 0; // variable locale pour j de 1 à 16 faire     x = compte; //P1     x = x +1 ; //P2     compte = x ; //P3 fin </pre>	<pre> processus_Q entier x = 0; // variable locale pour j de 1 à 16 faire     x = compte; //Q1     x = x +1 ; //Q2     compte = x ; //Q3 fin </pre>
---	---

Le résultat attendu par le programme utilisant les deux processus concurrents seraient 32. Ce résultat est effectivement atteint si l'on a l'ordre d'exécution suivant sur le processeur : [P1,P2,P3,Q1,Q2,Q3]<sup>16</sup> Ou encore si le système d'exploitation ordonne les processus de la manière suivante : [P1,P2,P3]<sup>16</sup>, [Q1,Q2,Q3]<sup>16</sup>. Mais dans le cas où l'ordonnancement arriverait à une exécution de type [P1Q1P2P3Q2Q3]<sup>16</sup>, le résultat serait donc de 16 !

L'exclusion mutuelle sert donc lorsque plusieurs processus peuvent avoir accès à une ressource critique (qui ne dispose que d'un seul point d'entrée). Elle sert à garantir que l'accès en section critique n'est possible que pour un seul processus à la fois. Elle doit ainsi réunir les conditions suivantes :

- *Exclusion* : à tout moment, un processus au plus se trouve en section critique
- *Accès*: Si des processus demandent la section critique, et si la section critique est libre, alors l'un d'entre eux doit y rentrer au bout d'un certain temps. (pas d'interblocage)
- *Indépendance*: Le blocage par cette section critique doit être indépendant des autres types de blocage (nn processus bloqué hors de sa section critique ne doit pas empêcher un autre d'y entrer)

- *Uniformité*: Aucun processus ne doit jouer de rôle privilégié.

Nous verrons dans les prochains chapitres différentes manières de programmer les sections critiques et l'exclusion mutuelle.

## 2.2 Producteurs - consommateurs

Le modèle des producteurs-consommateurs représentent une classe de problèmes où des activités (processus) produisent des informations, consommées par d'autres activités. La coopération des deux types de processus se fait via un buffer (tampon) ou une autre voie de communication, pouvant accepter un nombre maximal de messages. Les producteurs sont en concurrence entre eux, et les consommateurs aussi.

On peut citer plusieurs exemples de problèmes de cette classe :

- la saisie au clavier produit des caractères qui sont consommés (supprimés du buffer) par le processus responsable de l'affichage à l'écran.
- Le pilote de l'imprimante produit des lignes de caractères, qui sont consommées par l'imprimante
- Un compilateur produit des lignes de codes consommées par l'assembleur
- ...

## 2.3 Lecteurs rédacteurs

Le problème des lecteurs-rédacteurs se pose quand il y a compétition d'accès à un ensemble de données par un ensemble de processus, répartis en deux catégories :

- des processus lecteurs qui ont accès aux ressources en lecture uniquement
- des processus rédacteurs qui y ont accès en écriture et en lecture

Il s'agit dans ce cas de garantir la cohérence des données, sachant que les lecteurs peuvent accéder simultanément aux ressources, mais qu'un écrivain doit être seul à y accéder (en exclusion mutuelle).

On distingue plusieurs variantes classiques :

1. *Priorité aux lecteurs* : s'il existe des lecteurs sur la ressources, toute nouvelle demande de lecture est acceptée. Le risque est qu'un rédacteur puisse ne jamais accéder à la ressources (famine)
2. *Priorité aux lecteurs, sans famine des rédacteurs* : s'il existe des lecteurs sur la ressources, toute nouvelle demande de lecture est acceptée sauf si il y a un rédacteur en attente
3. *Priorité aux rédacteurs* : un lecteur ne peut lire que si aucun rédacteur n'est présent ou en attente. Il y a risque de famine des lecteurs
4. *FIFO* : les demandes d'accès sont servies dans l'ordre d'arrivée, et si il y a plusieurs lecteurs consécutifs, ils sont servis ensemble. Le regroupement est inefficace si les demandes sont alternées.

## 2.4 Diner des philosophes

Le problème des philosophes a été posé par Dijkstra en 1965. Il s'agit de partager plusieurs classes de ressources entre des processus concurrents asynchrones. L'illustration qui en a été faite consiste à représenter 5 philosophes assis autour d'une table, parlant et mangeant. Il y a 5 assiettes (une par philosophe) 5 baguettes (une entre deux assiettes voisines) et un plat de riz jamais vide.

Quand un philosophe parle, il ne tient aucune baguette. S'il désire manger, il doit prendre séquentiellement les deux baguettes bordant son assiette. Quand il a fini de manger, il repose les deux baguettes.

Sans mécanisme de synchronisation, deux cas dramatiques peuvent se produire :

- si chacun des philosophes prend une baguette sur sa gauche, et attend que l'autre baguette soit libre, ils sont tous bloqués.
- si deux philosophes entourant un philosophe s'entendent pour manger alternativement, le philosophe entouré meurt de faim.

Il faut donc éviter ces deux cas, l'interblocage et la famine.

### 3 Exclusion mutuelle

Nous avons présenté dans le chapitre précédent l'exclusion mutuelle. Nous allons nous y intéresser cette fois du point de vue de la recherche de solutions, en examinant les problèmes induits par le partage de mémoire entre processus (threads) : l'accès à une zone mémoire étant exclusif, les solutions devront garantir que :

- Un processus au plus est en section critique
- Il n'y a pas d'interblocage
- Un processus bloqué hors de sa section critique ne doit pas empêcher un autre d'y entrer.

Il existe deux types de solutions pour garantir l'exclusion mutuelle :

- Avec attente active : Les processus se détectent en conflit d'exclusion mutuelle en utilisant des mécanismes matériels, en cas d'attente ils exécutent une boucle
- Avec attente passive : les processus libèrent le processeur en cas d'attente. On utilise alors des mécanismes tels que les sémaphores, les moniteurs, les MSQ, ....

#### 3.1 Exclusion mutuelle avec attente active

Dans le cas d'une attente active, on définit un ensemble de variables partagées reflétant l'état d'occupation de la ressource et l'état des processus concurrents pour cette ressource. Avant d'entrer en section critique, un processus exécute un protocole testant et modifiant les variables d'état. Si la ressource n'est pas libre, il exécute une boucle d'attente. En sortant de section critique, un processus "libère" la ressource en modifiant les variables d'état.

Une première solution pourrait donc être du type suivant :

```

1 //déclaration et initialisation des variables communes
2 booléen occup = faux; // état d'occupation de la ressource
3
4 processus P0                                processus P1
5 tant que vrai faire                          tant que vrai faire
6 //entrée en SC                              //entrée en SC
7 tant que (occup);                            tant que(occup);
8 //attente active                            //attente active
9 occup=vrai;//ressource occupée              occup=vrai;//ressource occupée
10 Section_critique;                          Section_critique;
11 //sortie de SC                              //sortie de SC
12 occup=faux;                                 occup=faux;
13 hors SC;                                    hors SC;
14 fait;                                       fait;
```

Mais seule la lecture ou l'écriture d'une variable est atomique ... ainsi, entre deux instructions machines, le processeur peut-être alloué à un autre processus. Ainsi, dans l'exemple précédent, les deux processus peuvent être ensemble dans la section critique, si ils sortent l'un après l'autre des lignes 7, avant de pouvoir signifier l'occupation de la ressource ! L'exclusion mutuelle n'est pas respectée.

Deux solutions ont été proposées par Dekker en 1965 et Peterson en 1981, sur le principe suivant :

- chaque processus annonce sa candidature à l'autre processus
- en cas de candidatures simultanées, le conflit est réglé en donnant la priorité à un processus
- pour une solution équitable la priorité doit être variable

##### 3.1.1 Algorithme de Dekker

Pour éviter le cas précédent, Dekker introduit la notion de candidature à l'entrée en section critique : chaque processus annonce sa candidature à l'autre grâce à une variable partagée (un tableau). Si plusieurs candidatures surviennent simultanément, une priorité (variable) gère le conflit. La famine est évitée, l'exclusion mutuelle est garantie, il n'y a pas d'interblocage.

```

1 //déclaration et initialisation des variables communes
2 booléen C[2]={faux,faux}; //candidature
3 entier T=0; //priorité
4
```

```

5 processus P0
6 tant que vrai faire
7   //entrée en SC
8   C[0]=vrai;
9   tant que (C[1])faire           //P1 est en SC ou demande à y entrer
10    tant que(T==1)faire C[0]=faux; //P1 est prioritaire
11    C[0]=vrai;                   //T=0 P0 recandidate
12  fait;
13  Section_critique;
14  //sortie de SC
15  C[0]=faux;
16  T=1;
17  Hors SC;
18  fait;
19
20 processus P1
21 tant que vrai faire
22   //entrée en SC
23   C[1]=vrai;
24   tant que (C[0])faire           //P1 est en SC ou demande à y entrer
25    tant que(T==0) faire C[1]=faux; //P1 est prioritaire
26    C[1]=vrai;                   //T=0 P0 recandidate
27  fait;
28  Section_critique;
29  //sortie de SC
30  C[1]=faux;
31  T=0;
32  Hors SC;
33  fait;

```

### 3.1.2 Algorithme de Peterson

Peterson propose longtemps après Dekker une solution plus simple d'exclusion mutuelle par attente active.

```

1 //déclaration et initialisation des variables communes
2 booléen C[2]={faux,faux}; //candidature
3 entier T=0; //priorité
4
5 processus P0
6 tant que (vrai)faire
7   //entrée en SC
8   C[0]=vrai;T=1;
9   tant que (C[1] and T==1); //attente en cas de conflit
10  Section_critique;
11  //sortie de SC
12  C[0]=faux;
13  Hors SC;
14  fait;
15
16 processus P1
17 tant que (vrai)faire
18   //entrée en SC
19   C[1]=vrai;T=0;
20   tant que (C[0] and T==0); //attente en cas de conflit
21  Section_critique;
22  //sortie de SC
23  C[1]=faux;
24  Hors SC;
25  fait;

```

### 3.1.3 Inconvénients de l'attente active

Il existe d'autres mécanismes pour réaliser l'exclusion mutuelle par attente active : le masquage des interruptions en entrée de section critique, utilisation de verrous, ... Mais toutes les solutions présentent les inconvénients suivants :

- mauvaise utilisation du processeur
- ralentissement des accès mémoires (congestion du bus par des accès répétés aux variables de synchronisation)
- difficulté de conception et complexité des algorithmes

Ces solutions sont donc réservées à des sections critiques très courtes, et doivent être utilisées avec prudence. Heureusement, il existe également des mécanisme d'exclusion mutuelle avec attente passive.

## 3.2 Exclusion mutuelle avec attente passive

L'exclusion mutuelle avec attente passive se base sur des mécanisme tels que les sémaphores et les moniteurs, que nous allons présenter rapidement et sur lesquels nous reviendront dans les cahpitres suivants :

### 3.2.1 Sémaphores

Les sémaphores sont des mécanismes de base, fournis par le système d'exploitation ou par le langage de programmation. Il permetre une attente passive des processus par blocage, libérant ainsi le processeur le temps de ce blocage. Le contrôle de la concurrence se base sur le nombre d'autorisations d'accès, les accès n'étant accordés qu'en cas d'autorisations suffisantes.

C'est Dijkstra en 1965 qui a proposé une première définition des sémaphores : un sémaphore est un objet partagé constitué d'un entier E initialisé à une valeur supérieure ou égale à 0, et d'une file d'attente des processus bloqués. La manipulation du sémaphore se fait par 3 primitives atomiques :

- $P(S)$  : *Puis-je (Proberen)*, c'est le contrôle d'autorisation, qui peut le cas échéant bloquer le demandeur
- $V(S)$  : *Vas-y (Verhogen)*, qui rajoute une autorisation, et débloquent éventuellement un demandeur
- $E0(S,I)$  : *l'initialisation du sémaphore S*, à I autorisations, avec une file d'attente de processus vide

Le schéma général d'utilisation d'un sémaphore pour la réalisation d'une exclusion mutuelle est le suivant :

```

1 Contexte commun
2 mutex : sémaphore;E0(mutex,1)//1 seule autorisation
3 processus P :
4   tant que (vrai)faire
5     P(mutex);//entrée en section critique
6     Section critique;
7     V(mutex);//sortie de section critique
8     fait;
```

### 3.2.2 Moniteurs

Les sémaphores sont très efficaces dans la résolution des problèmes d'exclusion mutuelle et de synchronisation, mais peuvent induire des erreurs de programmation difficilement détectables. Les moniteurs proposent une solution alternative fournie par le langage de programmation, garantissant un accès approprié à des sections critiques. Le code placé avant et après une section critique est généré directement par le compilateur.

## 4 Exercices

### 4.1 Conditions de concurrence : exclusion mutuelle (Source : Ivan Boule)

Nous nous intéressons au problème d'allocation de ressources banalisées par exemple de pages de mémoire dans un système paginé. Nous supposons disposer de plusieurs exemplaires d'une même

ressource, dont un nombre quelconque peut être demandé par un processus à un moment donné. Nous considérons un seul type de ressource. Nous nous plaçons dans un contexte de multiprogrammation, plusieurs processus peuvent exécuter notamment, « simultanément » des codes partagés. Tout processus actif qui demande des exemplaires de cette ressource doit solliciter les services d'un allocateur via deux procédures :

- request (m) : pour demander l'allocation de m exemplaires de la ressource
- release (m) : pour libérer les m exemplaires de la ressource

L'allocateur maintient une variable qui compte le nombre d'exemplaires disponibles afin de satisfaire éventuellement de nouvelles demandes.

*Comportement d'un processus :*

```
Tant que (true) {
  request (m) ;
  utiliser les m ressources
  release (m) ;
}
```

*Allocateur :*

```
entier NbRessDisponibles :=Max ;
Procédure request (entier m) {
  (1) Tant que (NbRessDisponibles < m) attente du processus appelant;
  (2) NbRessDisponibles = NbRessDisponibles - m ;
}
Procédure release(entier m) {
  (3) Libération des m ressources
  (4) NbRessDisponibles = NbRessDisponibles + m ;
}
```

#### 4.1.1 Questions

1. Identifier un exemple d'incohérence dans ce pseudo-programme.
2. Pour éviter le problème précédent, on suppose qu'on dispose d'un mécanisme dit d'exclusion mutuelle qui n'autorise qu'un seul processus à la fois à exécuter un code partagé, la phase d'utilisation des codes est appelée section critique.

2. Discuter les solutions suivantes :

- a. la première solution suppose que chaque processus intègre la section critique suivante :

```
{
  request (m),
  utilisation de m ressources,
  release (m)
}
```

- b. la deuxième solution suppose que le processus fait appel à deux sections critiques séparées :

```
{
  request (m)
}
```

et

```
{
  release (m)
}
```

## 4.2 Algorithme de Dekker (Source Ivan Boule, Cnam)

On se propose de programmer l'exclusion mutuelle entre deux tâches parallèles pour l'accès à une section critique : les seules opérations indivisibles sont l'affectation d'une valeur à une variable et le test de la valeur d'une variable. Principe de la solution :

- Définir un ensemble de variables d'état, communes aux contextes des deux tâches.
- L'autorisation d'entrée en Section Critique sera définie par des tests sur ces variables, et l'attente éventuelle sera programmée comme une attente active (répétition cyclique des tests).

On supposera par la suite que les tâches sont cycliques et que leur comportement est le suivant :

```
While (true) {
    Entrée_en_section_critique
    Section_critique
    Sortie_de_section_critique
    Section_non_critique
}
```

La solution doit comporter les propriétés suivantes :

- A tout instant une tâche au plus peut se trouver en section critique.
- Si plusieurs tâches sont bloquées en attente de la ressource critique, alors qu'aucune tâche ne se trouve en section critique, l'une d'elles doit pouvoir y entrer au bout d'un temps fini (pas de blocage mutuel indéfini).
- Si une tâche est bloquée hors d'une section critique, ce blocage ne doit pas empêcher l'entrée d'une autre tâche en section critique.
- La solution doit être la même pour toutes les tâches (aucune tâche ne doit jouer de rôle privilégié).

#### 4.2.1 Question 1

On utilise une seule variable booléenne  $M$  telle que  $M = \text{vrai}$  si une des tâches se trouve dans sa section critique, faux sinon. *Ecrire le programme ; Vérifier que l'exclusion mutuelle ne peut être ainsi programmée.*

#### 4.2.2 Question 2

On utilise une variable commune unique  $T$  telle que  $T = i$  si et seulement si la tâche  $P_i$  est autorisée à entrer en sa section critique ( $i = 0, 1$ ). *Écrire le programme d'une tâche. Montrer que la solution ne vérifie pas la condition : "le blocage d'une tâche hors de sa section critique peut empêcher l'autre d'entrer en sa section critique"*

#### 4.2.3 Question 3

On utilise  $C(i)$  variable booléenne attachée à la tâche  $P_i$  ( $i = 0, 1$ )  $C(i) = \text{vrai}$  si  $P_i$  est dans sa section critique ou demande à y entrer  $C(i) = \text{faux}$  si  $P_i$  est hors de sa section critique  $P_i$  peut lire et modifier  $C(i)$ , peut lire seulement  $C(j)$  si  $j$  différent de  $i$ . *Écrire le programme de la tâche  $P_i$  Vérifier qu'on ne peut obtenir qu'une solution satisfaisant aux conditions a), c), d) ou b), c), d) mais non aux quatre.*

#### 4.2.4 Question 4

On peut obtenir une solution correcte en combinant les solutions précédentes et en introduisant une variable supplémentaire  $T$  servant à régler les conflits à l'entrée de la section critique,  $T$  n'est modifiée qu'en fin de section critique. L'ensemble des variables est:  $C(i)$  avec la signification précédente (Question 3) et  $T$  avec la signification précédente (Question 2). S'il y a un conflit ( $C(i) = C(j) = \text{vrai}$ ),  $P_i$  et  $P_j$  exécutent une séquence d'attente où  $T$  a une valeur constante. Si  $T = j$ , alors  $P_i$  annule sa demande en positionnant  $C(i)$  à faux,  $P_j$  peut alors entrer en section critique.  $P_i$  attend que  $T = i$  et refait sa demande en positionnant  $C(i)$  à vrai. *Écrire le programme de  $P_i$ . Vérifier les 4 conditions.*

## 5 Corrections de la partie 4

Pas de corrections, exercices simples à réaliser sur machine.

## 6 Références

### 6.1 Références

- [1] Paolo ZanellaYves Ligier, *Architecture et technologie des ordinateurs* , **Dunod** , isdn : ISBN 2 10 003801 X, 199,2002.
- [2] Matt WelshKalle DalheimerLar Kaufman, *Le systeme Linux* , **O'Reilly** , 1995, 1997, 1999, 2000.
- [3] Joelle Delacroix, *Linux* , **Dunod** , 2003.
- [4] Jean-Marie Rifflet, *La programmation sous Unix*, EdiScience , 1986, 1989, 1993, 2002.
- [5] Michael W. Lucas, *Absolute Freebsd: The Complete Guide to Freebsd*, **No Starch Press** , 2007.
- [6] Michael W. Lucas, *FreeBSD 7.0*, **No Starch Press** , 2007.
- [7] Nicholas P. Carter, *Architecture de l'ordinateur*, **EdiScience - Schaum's** , 2002.
- [8] J. Archer Harris, *Systèmes d'exploitation*, **EdiScience - Schaum's** , 2002.
- [9] Eric Steven Raymond, *The art of Unix Programming*, **Thyrsus Enterprises** , 2003.
- [10] Ivan BouleC. Coquery, *Cours SMB137*, **Cnam** , 2008, 2009.