Master d'Informatique **Programmation Concurrente,** **Université Pierre et Marie Curie**
M1 - Module PC2R **Réactive et Répartie** **Année 2005-20066**

Notes du cours 1

1 Systèmes à mémoire partagée

Objectifs :*Présentation du modèle à mémoire partagée de la programmation parallèle : section critique, exclusion mutuelle, sémaphores*

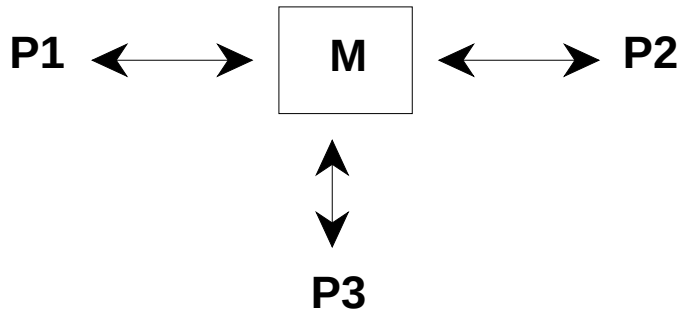
1.1 Généralités

On s'intéresse à deux grands modèles de programmation parallèle (ou simultanée) qui sont : les systèmes à mémoire partagée et les systèmes répartis (à mémoire répartie) que l'on étudiera au prochain cours. Chaque modèle met en valeur plusieurs notions fondamentales et étudie leurs relations. Les notions les plus importantes sont :

- la séquentialité (dépendance causale) : une instruction s'exécute après une autre ;
- la concurrence (indépendance causale) : plusieurs instructions s'exécutent en "même temps" ;
- le non-déterminisme (une cause peut avoir plusieurs effets, mutuellement exclusifs) : un même programme ne termine pas ou termine en produisant des résultats différents ;
- la synchronisation (plusieurs causes indépendantes doivent s'être produites avant que l'effet puisse avoir lieu) : attente d'une condition sur plusieurs processus ;
- la communication (transfert d'informations) : envoi et réception d'informations d'un ou plusieurs processus à un ou plusieurs processus.

1.2 Systèmes à mémoire partagée

On considère un ensemble S de processus séquentiels P_i interagissant sur une mémoire commune (ou partagée) que l'on note $S = [P_1 \ ||...|| \ P_n]$. Ces processus peuvent être aussi bien physiquement indépendants (un processus correspond à un processeur) que simulés logiquement par un unique processeur (comme les Threads en O'Caml).



La communication dans ce modèle est implicite. L'information est transmise lors de l'écriture dans une zone de la mémoire partagée, puis quand un autre processus vient lire cette zone. Ce mécanisme est asynchrone, i.e. il ne nécessite pas que le récepteur soit prêt à écouter l'émetteur. Par contre la synchronisation doit être explicite, en utilisant des instructions élémentaires.

Sans synchronisation explicite, le résultat d'un programme est imprévisible. Par exemple, soit l'ensemble S de processus (avec x valant 0) défini ainsi : $S = [x := x + 1; x := x + 1 || x := 2 * x]$. Après l'exécution de S, x peut valoir 2,3, ou 4.

La synchronisation la plus simple est l'attente d'une condition. On la note *wait* b, où b est une expression booléenne. Un processus ne peut exécuter cette instruction que si b est vraie. En reprenant l'exemple précédent :
$S = [x := x + 1; x := x + 1 || wait(x = 1); x := 2 * x]$ on obtient comme valeur pour x que 3 ou 4. Par contre il est possible que le second processus reste bloqué s'il n'a testé x que pour les valeurs 0 ou 2.

Cela amène le problème de l'atomicité. Il peut être utile de manipuler l'atomicité de manière explicite. L'instruction *await* b *do* P attend que la condition b soit vraie pour exécuter les instructions de P de manière atomique dans le même état mémoire que le test de b.

1.3 Section critique, exclusion mutuelle

On appelle *section critique* une ressource qui ne doit être utilisée que par un processus au plus. Par exemple, on désire qu'un seul processus puisse utiliser une imprimante. C'est le cas du système Unix qui gère une queue d'impression sur les périphériques d'impression.

Pour cela les processus doivent s'exclure mutuellement de la section critique. On dit que l'activité A_1 du processus P_1 et l'activité A_2 du processus P_2 sont en *exclusion mutuelle* lorsque l'exécution de A_1 ne doit pas se produite en même temps que celle de A_2.

Le problème de l'exclusion mutuelle, dans un cas simple de deux processus, n'est pas si évident que cela. L'algorithme de Dekker est une solution qui fonctionne. Il utilise une variable globale `turn` que chaque processus peut consulter et changer dans la section critique.

```
let turn = ref 1;;
let c = Array.create 2 1;;

let crit i = ();;                  (* action dans la section critique *)
let suite i = ();;                 (*        hors section critique    *)

let p i =
  while true do
  begin
    c.(i)<-0;                       (* desire entrer dans la section critique *)
    while c.((i+1) mod 2) = 0 do    (* tant que l'autre processus
                                       desire aussi entrer dans la section critique *)
      if !turn = ((i+1) mod 2) then (* si c'est au tour de l'autre *)
      begin
        c.(i)<-1;                               (* abandon *)
        while !turn = ((i+1) mod 2) do done;    (* et attente de son tour *)
        c.(i)<-0                                (* puis reprise *)
      end;
    done;
    crit i;
    turn := ((i+1) mod 2);          (* passe le droit \'a l'autre processus *)
    c.(i)<-1;                       (* remise \'a 1 : sortie de la section critique *)
    suite i
  end;;

(* initialisation *)
c.(0)<-1;;
c.(1)<-1;;
```

```
turn:=1;;

(* lancement des processus *)

Thread.create p 0;;
Thread.create p 1;;
```

Les processus indiquent leur volonté d'entrer dans la section critique en mettant à 0 l'élément de tableau c les concernant. Après avoir marqué son élément de tableau le processus va regarder si l'autre processus est dans le même état (volonté d'entrer dans la section critique). Si ce n'est pas le cas, il entre dans la section critique, sinon il doit consulter l'arbitre (*turn*) qui indique à qui est le tour. Cet arbitre ne peut être modifié que dans la section critique (ici à la sortie). De ce fait, seul le processus étant entré dans la section critique modifiera l'arbitre à la fin de son travail en lui indiquant l'autre processus.

1.4 Sémaphores

Un sémaphore est une variable entière s ne pouvant prendre que des valeurs positives (ou nulles). Une fois s initialisé, les seules opérations admises sont : $wait(s)$ et $signal(s)$, notées respectivement $P(s)$ et $V(s)$. Elles sont définies ainsi :

- $wait(s)$: si $s > 0$ alors $s := s - 1$ ($await\ s\ do\ s := s - 1$), sinon l'exécution du processus ayant appelé $wait(s)$ est suspendue.
- $signal(s)$: si un processus a été suspendu lors d'une exécution antérieure d'un $wait(s)$ alors le réveiller, sinon $s := s + 1$.

s correspond au nombre de ressources d'un type donné.

remarques Un sémaphore ne prenant que les valeurs 0 ou 1 est appelé *sémaphore binaire*.

Les primitives $wait(s)$ et $signal(s)$ s'excluent mutuellement si elles portent sur le même sémaphore (l'ordre n'est donc pas connu).

La définition de *signal* ne précise pas quel processus est réveillé s'il y en a plusieurs.

On peut utiliser les sémaphores pour l'exclusion mutuelle. Les deux processus `p 1` et `p 2` sont exécutés en parallèle grâce à la bibliothèque de `threads` d'OCaml.

```
let s = ref 1;;

let p i =
  while true do
  begin
    wait(s);
    crit();
    signal(s);
    suite()
  end
;;

Thread.create p 1;;
Thread.create p 2;;
```

Dans cet exemple, si un processus veut entrer en section critique, il entrera en section critique si :
- il n'y a que 2 processus (si P_1 est suspendu alors P_2 est en section critique ;
- si aucun processus ne s'arrête en section critique (si P_2 est dans `crit` alors il exécutera $signal(s)$.

Cette vérification ne fonctionne plus à partir de 3 processus. Il peut y avoir privation si le choix du processus se fait toujours en faveur de certains processus. Par exemple, si le choix s'effectue toujours en

faveur du processus d'indice le plus bas, P_1 et P_2 pourraient se liguer pour se réveiller mutuellement, P_3 étant alors indéfiniment suspendu.

1.5 Le classique "dîner des philosophes"

Le "dîner des philosophes", du à Dijkstra, illustre les différents pièges du modèle à mémoire partagée.

L'histoire se passe dans un monastère reculé où 5 moines se consacrent exclusivement à la philosophie. Ils passeraient bien tout leur temps à la réflexion s'ils ne devaient manger de temps en temps. La vie d'un philosophe se résume en une boucle infinie : penser - manger. Ils possèdent une table commune ronde. Au centre se trouve un plat de spaguettis qui est toujours rempli. Il y s 5 assiettes et cinq fourchettes. Le philosophe qui veut manger sort de sa cellule, s'assoit à table, mange et retourne ensuite à ses pensées. Les spaguettis sont si enchevétrés qu'il faut deux fourchettes pour pouvoir les manger. Un philosophe ne peut utiliser que les deux fourchettes autour de son assiette.

Les problèmes posés sont :

- l'interblocage : chaque philosophe tient une fourchette et attend qu'une autre se libère ;
- privation (ou famine) : un philosophe n'arrive jamais à obtenir 2 fourchettes.

1.6 Autres lectures

Ce cours s'inspire des documents suivants :

- "Processus Concurrents", Ben Ari, Masson 1986 ;
- "Concurrent Programming : Principles and practices", G. Andrews, Benjamin Cumming, 1991 ;
- Sémantique du parallélisme : un tour d'horizon en PostScript compressé de Luc Bougé.
- Le cours de Jean-Jacques Levy au MPRI (2004-20005) sur la concurrence et mémoire partagée.