Programmation Concurrente, Réactive et Répartie

Cours N°1

Emmanuel Chailloux

Master d'Informatique
Université Pierre et Marie Curie

année 2012-2013

Informations sur le cours PC2R

Sites

`http://www-apr.lip6.fr/~chaillou/Public/enseignement/2011-2012/pc2r/`

Equipe pédagogique

- cours : **Emmanuel Chailloux**
- TD/TME groupe 1 (mercredi après-midi) : **Philippe Trébuchet**
- TD/TME groupe 2 (mercredi après-midi) : **Philippe Wang**
- TD/TME groupe 3 (vendredi après-midi) : **Tong Lieu**

Description de l'UE

- Comprendre la programmation concurrente (concepts fondamentaux, aspects théoriques et pratiques) et son utilisation pour l'expression d'algorithmes,
- justifier la programmation réactive pour la conservation du déterminisme ,
- maitriser le modèle client/serveur (protocoles, ressources utilisées, modèles à n-acteurs),
- savoir déployer des objets répartis (coût des appels distants, gestion mémoire, sécurité).

mots clés :

Concurrence. synchronisation. communication. déterminisme. réactivité. mémoire partagée. mémoire répartie. client-serveur. objets distants.

Plan du cours - 1ère partie

1. généralités sur la concurrence,
   modèle à mémoire partagée, perte du déterminisme
2. modèle coopératif : Fair threads
3. modèle préemptif : Threads en O'Caml et en Java
4. modèle préemptif : Applets Java
   canaux synchrones : Events en O'Caml
5. Esterel : langage réactif

Plan du cours - 2ème partie

1. Interneteries, client/serveur
2. persistance et communication
3. appels distants, RMI en Java
4. chargement dynamique, servlets, JSP
5. réactive ML, migration de calcul

Logiciels du cours

Installation dans : `/users/Enseignants/chaillou/usr/local/`

- O'Caml 3-11-2 (préinstallé)
- Java 1.7 (préinstallé)
- GCC 4.4.5 (préinstallé)
- FTthread pour C (à installer)
- Esterel (utiliser CEC)

Sources de certaines installations dans :
`/users/Enseignants/chaillou/install`

Sources de certains exercices dans : `/Vrac/PC2R`

Evaluation

- 1ère session
  - Examen réparti 1 (40%) :
    - une épreuve de 2h la semaine du 05/11/2012 (20%)
    - projet de programmaiton (20%) par binôme
  - Examen réparti 2 : semaine du 07/01/2013 (60%)
- 2ème session
  - semaine du 13/05/2013

pourquoi la concurrence ?

séquentialité & concurrence :

- séquentialité (dépendence causale) : une instruction s'exécute après une autre;
- la concurrence (indépendance causale) : plusieurs instructions s'exécutent en «même temps»;

1. expressivité : facilité l'écriture d'algorithmes
   - séparation des tâches, explicitation de la communication, ...
2. efficacité : machines multicœurs et en réseau
   - différence entre puissances théorique et réelle

Modèles de parallélisme

2 grands modèles de programmation parallèle (ou simultanée) :

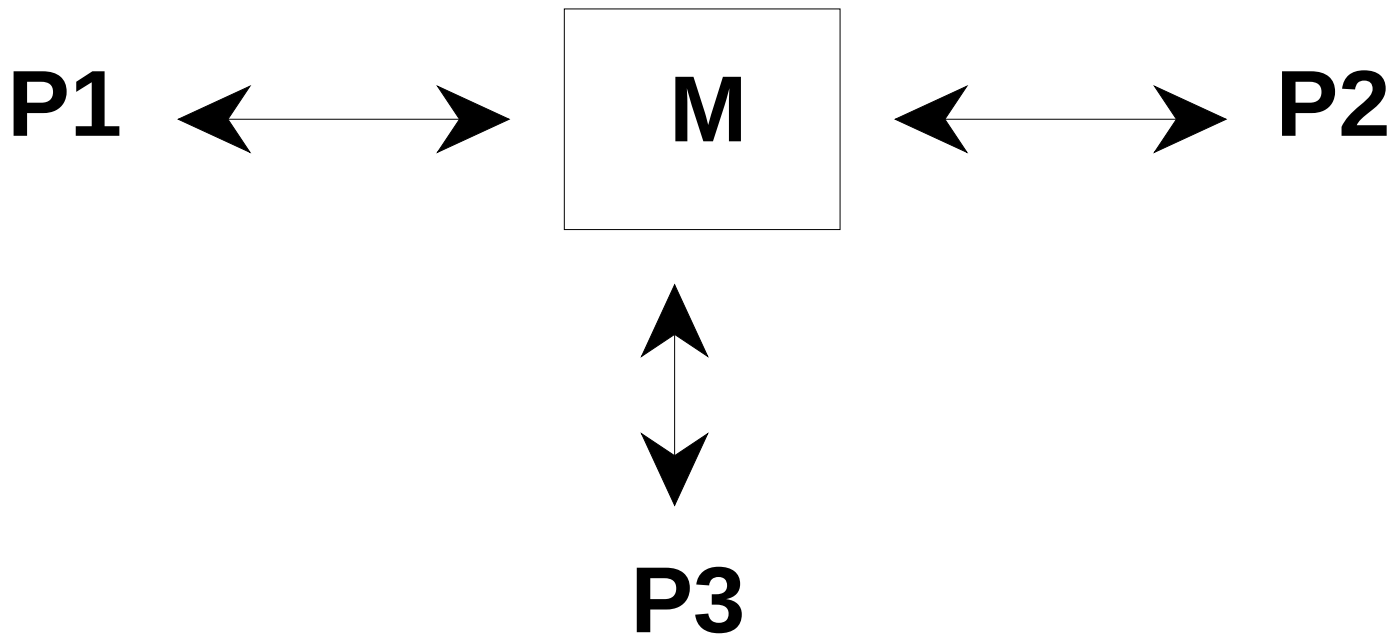
- systèmes à mémoire partagée
- systèmes répartis (à mémoire répartie)

Autres caractéristiques

- 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.

Systèmes à mémoire partagée (1)

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).



Systèmes à mémoire partagée (2)

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.

Systèmes à mémoire partagée (3)

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.

Systèmes à mémoire partagée (4)

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*.

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.

- algorithmes d'exclusion mutuelle
  - Dekker, Peterson, Lamport (Bakery)

Algorithme de Dekker (1)

Le problème de l'exclusion mutuelle, dans un cas simple de deux processus, n'est pas si évident que cela.

algorithme de Dekker:
Il utilise une variable globale `turn` que chaque processus peut consulter et changer dans la section critique.

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.

Algorithme de Dekker (2)

```
1  let turn = ref 1 and  c = Array.create 2 1;;
2  let crit i = ();;     (* action dans la section critique *)
3  let suite i = ();;    (*          hors section critique    *)
4  let p i =
5    while true do
6      c.(i)<-0; (* desire entrer dans la section critique *)
7      (* tant que l'autre processus  desire aussi entrer ↩
           dans la section critique *)
8      while c.((i+1) mod 2) = 0 do
9        if !turn = ((i+1) mod 2) then
10       (* si c'est au tour de l'autre *)
11       begin
12         c.(i)<-1;  (* abandon *)
13         while !turn = ((i+1) mod 2) do done;   (* et ↩
               attente de son tour *)
14         c.(i)<-0   (* puis reprise *)
15       end;
16     done;
17     crit i;
18     turn := ((i+1) mod 2);(* passe le droit au 2ème proc*)
19     c.(i)<-1;      (* remise \'a 1 : sortie de la SC *)
20     suite i
21   done ;;
```

Algorithme de Dekker (3)

Lancement :

```
1
2  (* initialisation *)
3  c.(0)<-1;;
4  c.(1)<-1;;
5  turn:=1;;
6
7  (* lancement des processus *)
8  Thread.create p 0;;
9  Thread.create p 1;;
```

algorithme de Peterson (1)

Il utilise une variable globale `turn` que chaque processus peut consulter et changer dans la section critique.

Les processus indiquent leur volonté d'entrer dans la section critique en mettant à 0 l'élément de tableau c les concernant.

Puis il donne la priorité (le tour) à l'autre processsus et attend que l'autre processus signale qu'il ne veut pas y aller ou qu'il lui (re)donne la priorité (atomique). .

algorithme de Peterson (2)

```
1  let turn = ref 1;;
2  let c = Array.create 2 1;;
3
4  let crit i = ();;  (* action dans la section critique *)
5  let suite i = ();; (*          hors section critique    *)
6
7  let p i =
8    while true do
9      c.(i) <- 0; (*désire entrer dans la section critique*)
10     turn := (i + 1) mod 2; (*   donne le tour à l'autre *)
11     (* tant que l'autre processus  désire entrer et que c'↩
           est son tour *)
12     while ( c.(i+1 mod 2) = 0 && !turn == (i+1) mod 2 ) do
13     done;
14     crit i;
15     c.(i) <- 1;
16     suite i
17   done ;;
```

Sémaphores (1)

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.

Sémaphores (2)

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.

Sémaphores (3)

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.

```
1  let crit () = ...
2  let suite () = ...
3  let s = ref 1;;
4
5  let p i =
6    while true do
7    begin
8      wait(s);
9      crit();
10     signal(s);
11     suite()
12   end
13 ;;
14
15 Thread.create p 1;;
16 Thread.create p 2;;
```

Sémaphores (4)

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.

Le dîner des philosophes (1)

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 tou jours rempli.

Le dîner des philosophes (2)

Il y a 5 assiettes et 5 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.