Programmation Concurrente, Répartie et Réactive. Cours 01 - Généralités et Exclusion Mutuelle

Emmanuel Chailloux
(présenté par Romain Demangeon)

PC2R 4I507 - M1STL

31/01/2019

Organisation

Ressources en Ligne

- Site du Master:
  `www-master.ufr-info-p6.jussieu.fr:8080/site-annuel-courant/PC2R`
- Site de l'UE:
  `www-apr.lip6.fr/~chaillou/Public/enseignement/2018-2019/pc2r/`

Equipe Pédagogique

- Cours: (jeudi 10h45-12h45) Emmanuel Chailloux
- TD1 (lundi 13h45-15h45) (24.34.205 puis 14.15.406): Romain Demangeon
- TD2 (lundi 13h45-15h45) (55.56.102 puis 14.15.408): Tong Lieu

Description de l'UE

- Comprendre la programmation concurrente (modèles, théorie et pratique) et son utilisation pour l'expression d'algorithme.
- Connaître la programmation synchrone, son style, son intérêt et ses subtilités.
- Maîtriser le modèle client-serveur.
- Savoir utiliser les objets répartis (appels, mémoire, sécurité).

Avant PC2R

Licence SU ou ailleurs:

- Cours de Systèmes d'Exploitation (mémoire partagée),
- Cours de Réseau ou de Web (client serveur).

Après PC2R

M1/M2 STL:

- CPS: *model-checking* concurrent.
- PPC: théorie de la concurrence et programmation synchrone.
- DAR: programmation réticulaire (web).

Mots-Clefs

Résumé du Cours

"La programmation concurrente, c'est difficile."

- Concurrence,

Mots-Clefs

Résumé du Cours

"La programmation concurrente, c'est difficile."

- Concurrence,
- Synchronisation,

Mots-Clefs

Résumé du Cours

"La programmation concurrente, c'est difficile."

- Concurrence,
- Synchronisation,
- Communication,

Mots-Clefs

Résumé du Cours

"La programmation concurrente, c'est difficile."

- Concurrence,
- Synchronisation,
- Communication,
- Déterminisme,

Mots-Clefs

Résumé du Cours

"La programmation concurrente, c'est difficile."

- Concurrence,
- Synchronisation,
- Communication,
- Déterminisme,
- Synchronisme,

Mots-Clefs

Résumé du Cours

"La programmation concurrente, c'est difficile."

- Concurrence,
- Synchronisation,
- Communication,
- Déterminisme,
- Synchronisme,
- Réactivité,

Mots-Clefs

Résumé du Cours

"La programmation concurrente, c'est difficile."

- Concurrence,
- Synchronisation,
- Communication,
- Déterminisme,
- Synchronisme,
- Réactivité,
- Mémoire partagée,

Mots-Clefs

Résumé du Cours

"La programmation concurrente, c'est difficile."

- Concurrence,
- Synchronisation,
- Communication,
- Déterminisme,
- Synchronisme,
- Réactivité,
- Mémoire partagée,
- Mémoire répartie,

Mots-Clefs

Résumé du Cours

"La programmation concurrente, c'est difficile."

- Concurrence,
- Synchronisation,
- Communication,
- Déterminisme,
- Synchronisme,
- Réactivité,
- Mémoire partagée,
- Mémoire répartie,
- Client-serveur.

Plan du Cours

1. Généralités sur la concurrence, modèle à mémoire partagée, perte du déterminisme,
2. Modèle coopératif : Fair threads, Lwt,
3. Modèle préemptif : Threads en OCaml et en Java,
4. Interneteries, client/serveur
5. Modèles avancés : canaux synchrones, futures
6. programmation synchrone (1) : Esterel
7. programmation synchrone (2) : Lustre & Scade
8. persistance et communication
9. appels distants, RPC et RMI en Java
10. chargement dynamique, servlets, JSP modèles pour le Web (multi-tiers)

Prérequis

Programmation en Java, C et OCaml.

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

- OCaml 4 (préinstallé)
- Java 1.8 (préinstallé)
- GCC 4.4.5 (préinstallé)
- FTthread pour C (à installer)
- Esterel (v5.100)
- Scade (sur machines Windows)

Sources de certaines installations dans :
`/users/Enseignants/chaillou/install`
Sources de certains exercices dans : `/Vrac/PC2R`

Evaluation

- 1ère session:
  - Examen réparti 1 (40%) :
    - épreuve de 2h en mars (20%),
    - devoir de programmation (20%) par binôme.
  - Examen réparti 2 (60%) :
    - épreuve de 2h en mai
- 2ème session
  - épreuve de 2h en juin (60/100%)

Evaluation (II)

Examens

- Annales disponibles (site de l'UE).
- Exercices couvrant les différentes parties.
- multi-langages (impératif, objet, fonctionnel, synchrone).

Devoir en binômes

- Souvent un jeu multijoueur.
- Architecture client-serveur (répartition canonique déconseillée).
- Deux langages différents obligatoires.
- Sujet en Semaine 3. Projet à rendre pour la Semaine 10 (à confirmer).

Plan du Cours 1

- Concurrence:
  - Modèle à mémoire partagée.
  - Modèle à mémoire répartie.
- Section critique, exclusion mutuelle:
  - Algorithme de Dekker.
  - Algorithme de Peterson.
  - Sémaphores.

Concurrence

Séquentialité et Concurrence

- Séquentialité (dépendance causale): les instructions s'exécutent les unes après une autre.
- Concurrence/Parallèlisme (indépendance causale): plusieurs instructions s'exécutent en même temps.

Parallelisme et Concurrence

Des instructions sont exécutées par plusieurs unités de calcul.

- Parallèlisme: les flots de calculs sur chaque unité sont indépendants les uns des autres.
- Concurrence: les flots de calculs partagent de l'information (ressources, messages, synchronisations).

Souvent on utilise parallèle pour les deux, mais on garde concurrent pour le partage (étymologie).

Programmation Concurrente

Points forts

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

Prix à payer: difficulté de la programmation.

- Répartition des tâches entre plusieurs unités. Compromis entre parallélisation et surcoût en messages.
- Non-déterminisme: un même jeu de programmes distribués peut exhiber des comportements différents (plusieurs effets possibles pour une cause, terminaison).
- Non-confluence: certains non-déterminismes sont irréversibles (consommation de ressources, choix d'un partenaire, non-commutativité de deux actions).
- Comportements indésirables: interbloquages, cycles non-productifs, divergence.

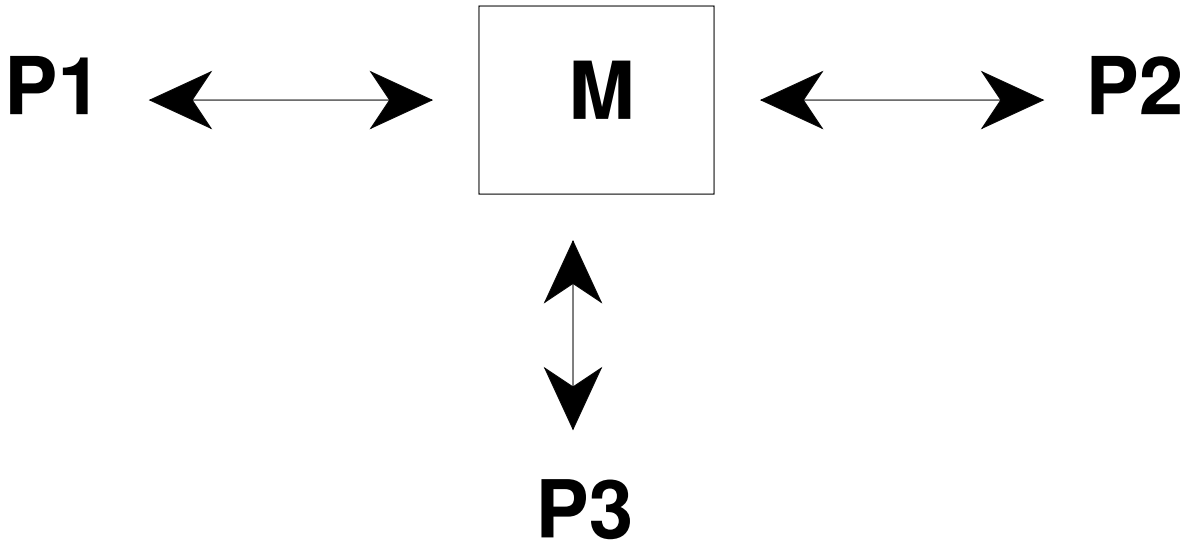
Programmation Concurrente (II)

- Synchronisation: plusieurs causes indépendantes produisent un unique effet: attente d'une condition, message synchrone.
- Communication: transfert d'information entre des unités différentes (par mémoire partagée, message, ou signaux).

1. Mémoire partagée: plusieurs unités partagent une même zone mémoire (elles peuvent avoir en plus des mémoires indépendantes). Synchronisation explicite (utilisation d'instructions élémentaire). Communication implicite (écriture/lecture) asynchrone.
2. Mémoire répartie: plusieurs unités communiquent par des messages (elles disposent de mémoires indépendantes). Synchronisation implicite (attente de message). Communication explicite (primitive d'envoi/réception de messages).

Systèmes à Mémoire Partagée (1)

- On considère un ensemble S de processus (programmes) séquentielles 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 OCaml).



Sémantique d'entrelacement

On utilise un langage impératif simple pour décrire les processus..

Définition

La sémantique d'un système de processus à mémoire partagée est donnée par l'entrelacement des actions atomiques des processus.

- entrelacement (*shuffle*) non-déterministe qui donne une séquence de toutes les instructions atomiques des processus. L'ordre entre instructions du même processus doit être respecté.
- instruction atomique: instruction qu'on ne peut pas diviser (étymologie). Entre le début et la fin d'une telle instruction, aucune autre manipulation mémoire ne peut avoir lieu.

Exemples

- Sans synchronisation explicite, le résultat d'un programme est imprévisible. Par exemple, soit l'ensemble S de processus suivant : $S = [x := x + 1; x := x + 1 || x := 2 * x]$. Après l'exécution de S sur une zone mémoire ou x vaut 0, x peut valoir

Exemples

- Sans synchronisation explicite, le résultat d'un programme est imprévisible. Par exemple, soit l'ensemble S de processus suivant : $S = [x := x + 1; x := x + 1 || x := 2 * x]$. Après l'exécution de S sur une zone mémoire ou x vaut 0, x peut valoir 2,3, ou 4 (affectation atomique)

Exemples

- Sans synchronisation explicite, le résultat d'un programme est imprévisible. Par exemple, soit l'ensemble S de processus suivant : $S = [x := x + 1; x := x + 1 || x := 2 * x]$. Après l'exécution de S sur une zone mémoire ou x vaut 0, x peut valoir 2,3, ou 4 (affectation atomique) ou 0,1,2,3 ou 4 (division lecture/écriture).
- 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, sinon il doit répéter cette instruction.
- 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.

Atomicité et Limite

- Il peut être utile de manipuler l'atomicité de manière explicite.
- Exemple: 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*.

Limite de la sémantique d'entrelacement

- Considérer [`write Y 1; read X||write X 1; read Y`] dans une mémoire ou `X` et `Y` valent 0.
- Résultats possibles des lectures:

Atomicité et Limite

- Il peut être utile de manipuler l'atomicité de manière explicite.
- Exemple: 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.

Limite de la sémantique d'entrelacement

- Considérer [`write Y 1; read X || write X 1; read Y`] dans une mémoire ou `X` et `Y` valent 0.
- Résultats possibles des lectures: $(1,1), (0,1), (1,0)$.

Atomicité et Limite

- Il peut être utile de manipuler l'atomicité de manière explicite.
- Exemple: 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.

Limite de la sémantique d'entrelacement

- Considérer [`write Y 1; read X || write X 1; read Y`] dans une mémoire ou `X` et `Y` valent 0.
- Résultats possibles des lectures: $(1,1), (0,1), (1,0)$.
- Sur architecture x86, une fois sur 10 millions: $(0,0)$.

Atomicité et Limite

- Il peut être utile de manipuler l'atomicité de manière explicite.
- Exemple: 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.

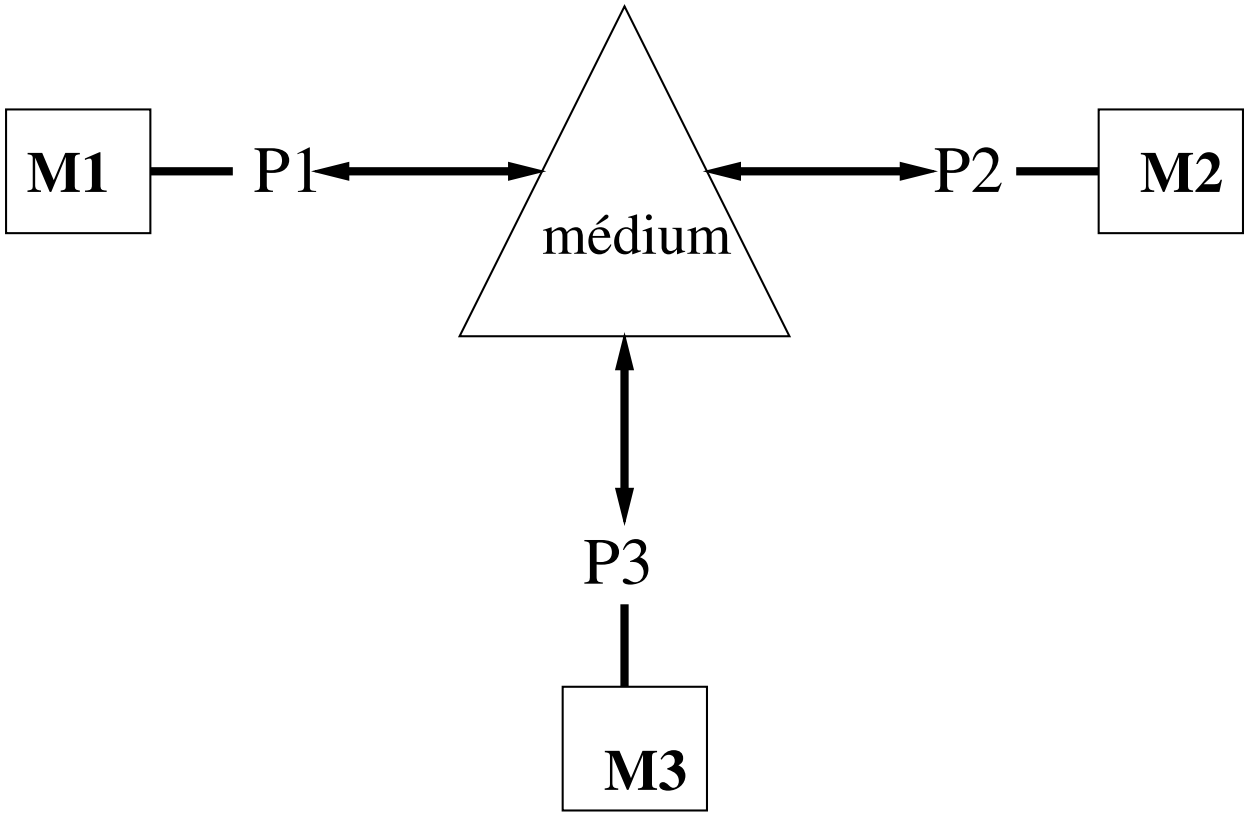
Limite de la sémantique d'entrelacement

- Considérer [`write Y 1; read X||write X 1; read Y`] dans une mémoire ou `X` et `Y` valent 0.
- Résultats possibles des lectures: $(1,1), (0,1), (1,0)$.
- Sur architecture x86, une fois sur 10 millions: $(0,0)$.
- *Write buffering* (modèle mémoire faibles).

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Modèle à mémoire répartie

Dans ce modèles l'implantation du medium est cruciale. Les programmes s'en chargeant s'appellent des protocoles.



Protocoles et communication

- protocole : organisés en couches. Les protocoles de haut niveau, implantant des services élaborés, utilisent les couches de plus bas niveaux (7 couches du modèle OSI).
- parallélisme : modèle valable dans le cas de parallélisme physique (réseau d'ordinateurs) ou logique (processus Unix communiquant par "pipes" ou threads OCaml communiquant par canaux). Il n'y a pas de valeurs globales connues par tous les processus (comme un temps global). La seule contrainte sur le temps est l'impossibilité de recevoir un message avant son émission.

Modèles de communication

- un-à-un (point-à-point) : communication d'un processus à un autre; les autres processus ignorent cette communication. Les deux primitives sont "envoi d'une valeur sur un canal" et "réception d'une valeur d'un canal".
- un-à-tous (diffusion) : communication d'un processus à tous les processus. Les primitives de communication sont : "envoi d'une valeur à tous" et "réception d'une valeur".
- tous-à-tous (diffusion) : communication de tous les processus à tous l es processus. La réception tient compte alors des différentes valeurs envoyées.

Types de communication

- synchrone : le transfert d'informations n'est possible que lors d'une synchronisation simultanée des processeurs émetteur et récepteur. L'émission et la réception sont bloquantes.
- asynchrone : le medium peut stocker des messages en vue de leur acheminement futur. Il faut donc spécifier la capacité de stockage, l'ordre d'acheminement, les délais de transmissions et la fiabilité de transmission. L'émission est non bloquante.
- évanescent : l'émission est non bloquante et le medium ne peut pas stocker de messages. Le message émis est reçu par les processus prêt à le recevoir et perdu pour les autres.

Section critique et exclusion mutuelle

Définition

Une section critique est 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

Des algorithmes (utilisant des instructions élémentaires) permettent de garantir l'exclusion mutuelle de tous les processus d'une section critique.

- Dekker, Peterson, Lamport (Bakery).

Algorithme de Dekker (1)

Difficulté du problème de l'exclusion mutuelle dès le cas simple de deux processus et d'une section critique.

Algorithme de Dekker

- on 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 consulte la variable globale (*turn*) qui indique qui a la priorité. Cet arbitre ne peut être modifié que dans la section critique. Ainsi, le processus étant entré dans la section critique modifie la variable globale à la fin de son travail en lui indiquant l'autre processus.

Algorithme de Dekker (2)

```
let turn = ref 1 and  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
    c.(i)<-0; (* desire entrer dans la section critique *)
    (* tant que l'autre processus  desire aussi *)
    while c.((i+1) mod 2) = 0 do
      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 au 2eme proc*)
    c.(i)<-1;      (* remise a 1 : sortie de la SC *)
    suite i
  done ;;
```

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Algorithme de Dekker (3)

Lancement:

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

(* lancement des processus *)
Thread.create p 0;;
Thread.create p 1;;
```

Algorithme de Peterson (1)

- On 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.
- On 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)

```
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
    c.(i) <- 0; (*desire entrer dans la section critique*)
    turn := (i + 1) mod 2; (*   donne le tour a l'autre *)
    (* tant que l'autre processus  desire entrer et que c'est son tour *)
    while ( c.(i+1 mod 2) = 0 && !turn == (i+1) mod 2 ) do
    done;
    crit i;
    c.(i) <- 1;
    suite i
  done ;;
```

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Attentes actives

- les algorithmes précédents comportent des boucles qui attendent la mise à jour d'une variable partagée.
- en pratique, les processus vont utiliser des cycles de calcul pour vérifier une condition de boucle.
- cette caractéristique, appelée attente active est indésirable (gachis d'occupation d'unité). Produire du code parallèle contenant des attentes actives est une faute.
- la plupart des architectures et des langages disposent de mécanismes adéquats (mise en sommeil, signal de réveil) pour traiter ce problème.

Sémaphores

Dans un cadre avec un nombre quelconque de processus, 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 $V(s)$:

- *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 processus pouvant partager une ressource 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.

Les sémaphores constituent un mécanisme permettant d'éviter les attentes active.

Exemple

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 crit () = ...
let suite () = ...
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;;
```

Progrès

Définition

Pour un processus P, le progrès est l'absence d'exécution du système dans laquelle P désire effectuer une action (progresser) mais ne l'effectue jamais.

- Dans le cas de l'exclusion mutuelle, le progrès caractérise le fait de finir par entrer en section critique.
- Dans l'exemple précedent, si un processus veut entrer en section critique, il finira par y entrer si :
  - il n'y a que 2 processus (si P_1 est suspendu alors P_2 est en section critique);
  - et si aucun processus ne s'arrête en section critique (si P_2 finit `crit` alors il exécute $signal(s)$).
- Cet argument 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", dû à Dijkstra, illustre les différents pièges du modèle à mémoire partagée.

L'histoire se passe dans un monastère reculé où n moines se consacrent exclusivement à la philosophie. 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 qui est toujours rempli.
Il y a 5 assiettes et 5 baguettes. Le philosophe qui veut manger sort de sa cellule, s'assoit à table, prend les deux baguettes autour de son assiette, mange et retourne ensuite à ses pensées.

Le Dîner des philosophes (2)

Types de problèmes

- sûreté: "peut-on arriver dans un mauvais état ?"
- vivacité: "arrive t-on forcément dans un bon état ?"

Les problèmes posés sont :

- interblocage (sûreté): peut-on arriver à une situation ou plus personne n'effectue d'action ?
- famine (vivacité): tout philosophe mange t-il infiniment souvent ?

Le Dîner

- Cas abstrait modélisant des cas concrets très fréquents.
- Illustre la difficulté de la programmation concurrente.
- Revient en CPS et PPC.