PC3R
Cours 01 - Modèle Préemptif

Romain Demangeon

PC3R MU1IN507 - STL S2

15/01/2025

Plan du Cours 1

- Programmation Concurrente.
- Mémoire partagée vs. mémoire répartie.
- Sémantique d'entrelacement.
- Section critique, exclusion mutuelle:
  - Ecueils de la concurrence.
  - Algorithme de Dekker, Peterson.
  - Sémaphores.
- Présentations rapides:
  - Threads POSIX,
  - Threads Java,
  - Partage en Rust.

Concurrence

Séquentialité et Concurrence

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

Parallelisme et Concurrence

Des instructions sont exécutées par plusieurs unités de calcul.

- Parallélisme: 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.

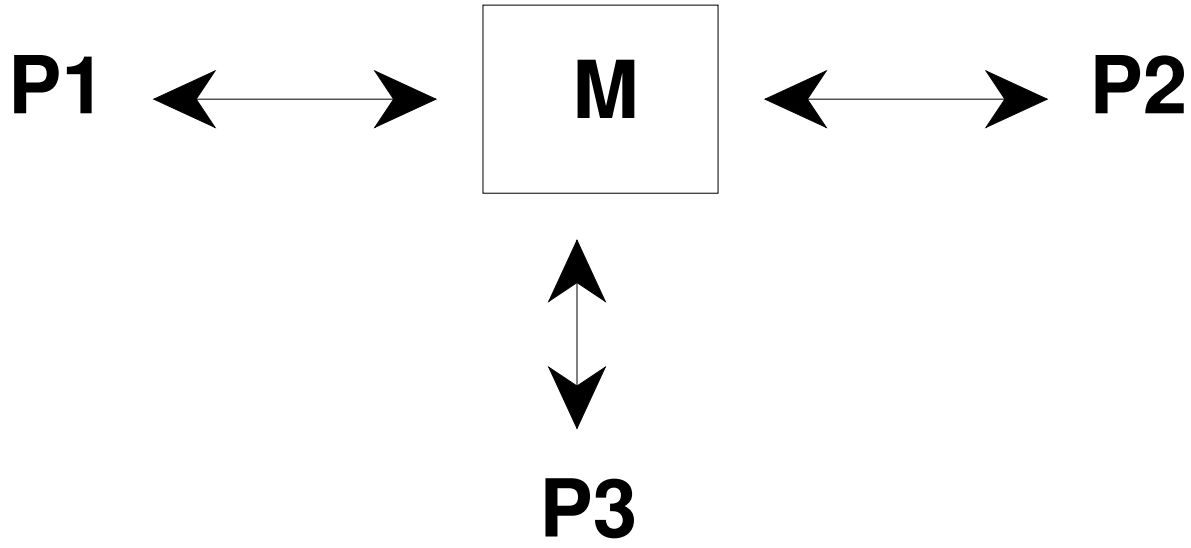
Programmation Concurrente (II)

- 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.
   - vision moderne: "les mémoires partagées ne devraient pas être utilisées pour communiquer."
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).

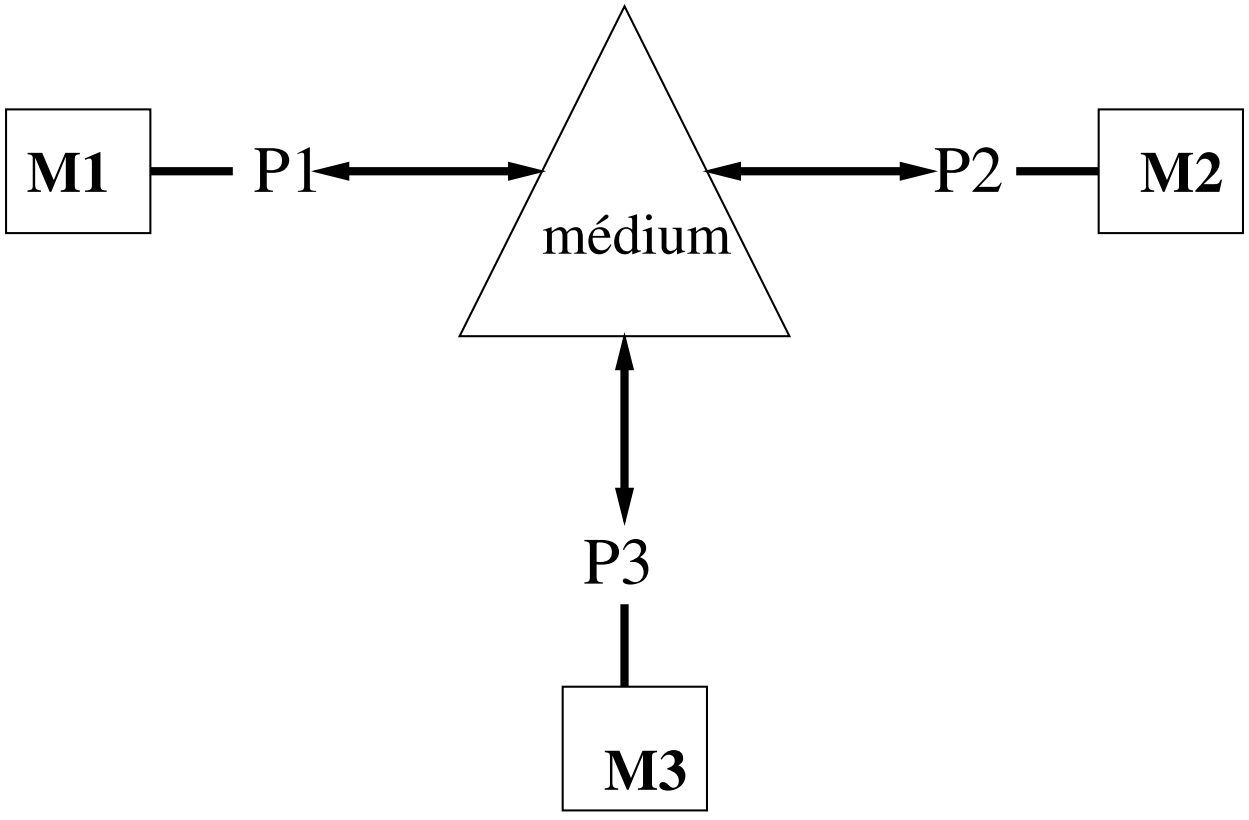
Mémoire Partagée

- Les processus agissent sur une mémoire commune (partagée).
- Les processus peuvent lire et écrire les mêmes cases mémoires.



Mémoire répartie

- Les processus ont chacun des mémoires indépendantes (réparties).
- Des communication sont possibles à travers un medium.



- protocole: implémentation du medium.

Types de communication (mémoire répartie)

- synchrone: la communication n'est possible que lorsque deux processus sont simultanément prêts à envoyer et recevoir.
  - émission et réception bloquantes.
- asynchrone : le medium peut stocker des messages, la réception d'un message se fait après (causalité) son émission.
  - émission non bloquante.
- envoi/réception nominatif: un message est envoyé à un processus donné.
- envoi/réception par canal: un message est envoyé dans un endroit abstrait (canal).
- abonnement/publication : un message est publié à un endroit abstrait, et transmis à tous les abonnés à cet endroit.

Modèle Préemptif (mémoire partagée)

- Le modèle préemptif est la sémantique standard des systèmes à mémoire partagée, basé sur la réalité des systèmes d'exploitation.
  - plusieurs processus (tâches séquentielles, suites d'instructions).
  - un seul processeur physique.
  - une mémoire commune à tous les processus.
- L'exécution concurrente des processus est obtenue en exécutant successivement sur le processeur quelques instructions de chaque processus.
- L'ordonnanceur est l'algorithme qui s'occupe de choisir (élection) le processus physiquement exécuté (les autres sont en attente), de l'arrêter (préemption) au bout d'un certain temps, et de choisir un nouveau processus
  - le pseudo-hasard est souvent utilisé dans ces choix
- L'entrelacement d'instructions de processus différents donne l'illusion de la simultanéité.
- Le changement de contexte (*context switch*) opéré à chaque élection (variables locales, environnement, ...) est coûteux.

Sémantique d'entrelacement

Définition (tautologique)

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

- instruction atomique: instruction qu'on ne peut pas diviser (étymologie).
  - le langage utilisé doit préciser ce qui est atomique,
  - en réalité, l'architecture décrit quand une préemption est possible (et donc ce qui est atomique)
- entrelacement (*interleaving*) de séquences s_i: ensemble de toutes les séquences contenant toutes les instructions des séquences s_i en préservant l'ordre entre instructions d'une même séquence.
- exemple avec $P_1 = a; b$, $P_2 = c; d$ et $P_3 = e$
  - $a; c; b; e; d$ est dans l'entrelacement de $[P_1||P_2||P_3]$
  - $a; b; c; d; e$ est dedans (pas de préemption)
  - $a; d; b; e; c$ n'est pas dedans (ordre de P_2)
- la taille de l'entrelacement peut être infinie (indénombrable) si un processus est non-terminant.

Exemple et Espace d'état

(on utilise un langage impératif simple pour décrire les processus.)

- l'effet effectif d'un programme concurrent dans le modèle préemptif est une des séquences de sa sémantique d'entrelacement
  - en l'absence d'information exacte sur l'ordonnanceur (qui utilise le pseudo-aléatoire), il est imprévisible.
- Soit $S = [x := x + 1; x := x + 1 || x := 2 * x]$.
  - x initialisée à 0 est partagée entre les deux processus de S
  - après l'exécution de S , x peut valoir

Exemple et Espace d'état

(on utilise un langage impératif simple pour décrire les processus.)

- l'effet effectif d'un programme concurrent dans le modèle préemptif est une des séquences de sa sémantique d'entrelacement
  - en l'absence d'information exacte sur l'ordonnanceur (qui utilise le pseudo-aléatoire), il est imprévisible.
- Soit $S = [x := x + 1; x := x + 1||x := 2 * x]$.
  - x initialisée à 0 est partagée entre les deux processus de S
  - après l'exécution de S , x peut valoir
    - 2,3, ou 4 (affectation atomique)

Exemple et Espace d'état

(on utilise un langage impératif simple pour décrire les processus.)

- l'effet effectif d'un programme concurrent dans le modèle préemptif est une des séquences de sa sémantique d'entrelacement
  - en l'absence d'information exacte sur l'ordonnanceur (qui utilise le pseudo-aléatoire), il est imprévisible.
- Soit $S = [x := x + 1; x := x + 1||x := 2 * x]$.
  - x initialisée à 0 est partagée entre les deux processus de S
  - après l'exécution de S , x peut valoir
    - 2,3, ou 4 (affectation atomique)
    - ou 0,1,2,3 ou 4 (division lecture/écriture).

Exemple et Espace d'état

(on utilise un langage impératif simple pour décrire les processus.)

- l'effet effectif d'un programme concurrent dans le modèle préemptif est une des séquences de sa sémantique d'entrelacement
  - en l'absence d'information exacte sur l'ordonnanceur (qui utilise le pseudo-aléatoire), il est imprévisible.
- Soit $S = [x := x + 1; x := x + 1||x := 2 * x]$.
  - x initialisée à 0 est partagée entre les deux processus de S
  - après l'exécution de S , x peut valoir
    - 2,3, ou 4 (affectation atomique)
    - ou 0,1,2,3 ou 4 (division lecture/écriture).
- Point crucial: la taille de l'espace d'état de l'entrelacement (ensemble des états des séquences de l'entrelacement) est exponentielle par rapport aux tailles des espaces d'état des processus.
  - c'est la difficulté de la programmation concurrente:
    - tests,
    - vérification,
    - possibilité d'états indésirables,

Atomicité et Limite

- Il peut être utile de manipuler l'atomicité de manière explicite.
  - i.e.: signaler expliciement qu'une suite d'instructions est atomique: la préemption ne peut avoir lieu entre le début et la fin de la suite.
- Soit $S = [ATOM(x := x + 1; x := x + 1)||x := 2 * x]$.
  - les valeurs de x possibles sont

Atomicité et Limite

- Il peut être utile de manipuler l'atomicité de manière explicite.
  - i.e.: signaler expliciement qu'une suite d'instructions est atomique: la préemption ne peut avoir lieu entre le début et la fin de la suite.
- Soit $S = [ATOM(x := x + 1; x := x + 1)||x := 2 * x]$.
  - les valeurs de x possibles sont $\{0, 4\}$.

Limite de la sémantique d'entrelacement

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

Atomicité et Limite

- Il peut être utile de manipuler l'atomicité de manière explicite.
  - i.e.: signaler expliciement qu'une suite d'instructions est atomique: la préemption ne peut avoir lieu entre le début et la fin de la suite.
- Soit $S = [ATOM(x := x + 1; x := x + 1)||x := 2 * x]$.
  - les valeurs de x possibles sont $\{0, 4\}$.

Limite de la sémantique d'entrelacement

- Considérer [`write Y 1`; `read X`||`write X 1`; `read Y`] dans une mémoire ou `X` et `Y` sont initialisées à 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.
  - i.e.: signaler expliciement qu'une suite d'instructions est atomique: la préemption ne peut avoir lieu entre le début et la fin de la suite.
- Soit $S = [ATOM(x := x + 1; x := x + 1)||x := 2 * x]$.
  - les valeurs de x possibles sont $\{0, 4\}$.

Limite de la sémantique d'entrelacement

- Considérer [`write Y 1; read X`||`write X 1; read Y`] dans une mémoire ou `X` et `Y` sont initialisées à 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.
  - i.e.: signaler expliciement qu'une suite d'instructions est atomique: la préemption ne peut avoir lieu entre le début et la fin de la suite.
- Soit $S = [ATOM(x := x + 1; x := x + 1)||x := 2 * x]$.
  - les valeurs de x possibles sont $\{0, 4\}$.

Limite de la sémantique d'entrelacement

- Considérer [`write Y 1; read X`||`write X 1; read Y`] dans une mémoire ou `X` et `Y` sont initialisées à 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).

Ecueils de la Concurrence

Compétition (*race*):

- résultat indésirable obtenu par l'accès "simultané" à une même ressource par plusieurs processus.
- $$\left[\begin{array}{l} \texttt{u := x} \\ \texttt{u := u + 1} \\ \texttt{x := u} \end{array} \middle\| \begin{array}{l} \texttt{v := x} \\ \texttt{v := v + 1} \\ \texttt{x := v} \end{array}\right]$$
- intention: chaque processus incrémente un même compteur `x`
- compétition: si les deux processus incrémentent `x` "en même temps", c'est-à-dire si la préemption a lieu apres la 1ere ou deuxieme instruction, le compteur n'atteint pas 2.
- Solution: utiliser des outils du langage (du système) pour:
  - expliciter l'atomicité d'une série d'instructions (mutex).
  - protéger une ressource partagée.

Ecueils de la Concurrence (II)

Interbloquage (*deadlock*)

- situaton indésirable dans laquelle au moins un processus non-terminé ne peut plus effectuer d'instructions
-

```
┌ while(true)      ‖ while(true)      ┐
│ ....wait(b)      ‖ ....wait(c)      │
│ ....b := false   ‖ ....c := false   │
│ ....wait(c)      ‖ ....wait(b)      │
│ ....c := false   ‖ ....b := false   │
│ ....x := x + 1   ‖ ....x := x + 1   │
│ ....b := true    ‖ ....b := true    │
└ ....c := true    ‖ ....c := true    ┘
```

  avec `b` et `c` initialement à `true`.
- `wait(b)` est une instruction qui ne s'exécute pas tant que `b` n'est pas vraie.
- dans la sémantique d'entrelacement, il existe une séquence où les deux processus restent bloqués sur des `wait` (aucune instruction ne peut être exécutée).
- Solutions:
  - modification des processus (leur code) pour éviter le deadlock
  - détection du deadlock:
    - tests exhaustif (limites),
    - étude de l'ordonnanceur.

Ecueils de la Concurrence (III)

Famine:

- Situation indésirable dans laquelle un processus prêt à effectuer une instruction ne l'exécute jamais.
- $\left[\begin{array}{l} \texttt{while(true)} \\ \texttt{....x := x + 1} \end{array} \right\| \left. \begin{array}{l} \texttt{while(true)} \\ \texttt{....y := x} \end{array} \right]$
- dans la sémantique d'entrelacement, il existe une séquence infinie dans laquelle le processus de gauche est le seul à exécuter ses instructions.
- l'ordonnanceur d'un système est (souvent) un algorithme (il n'est pas totalement aléatoire) utilisant des priorités
- un mauvais ordonnanceur peut conduire à des situation de famines.
- Solutions:
  - modification de l'ordonnanceur pour qu'il soit équitable (*fair*)
  - détection de la famine:
    - tests exhaustif,
    - vérification statique.

Ecueils de la Concurrence (IV)

Attente Active (*busy-waiting*)

- Situation indésirable dans laquelle un processus élu exécute la même instruction d'attente
- ```
  [ while(true)          ||  proc()
  ....if(b)              ||  b := true
  ........x := x + 1     ||
  ]
  ```
  avec `b` initialisé à `false`
- le processus de gauche attend que `b` soit mis à `true`
- si `proc()` est long, à chaque élection, le processus de gauche va répéter la même suite d'action (condition du while, condition du if) improductive.
- Solutions:
  - utilisation des outils dédiés du système pour endormir un processus, et le réveiller quand l'attente est finie.
  ```

Ecueils de la Concurrence (V)

Cycle non-productif (*livelock*)

- Situation indésirable dans laquelle plusieurs processus exécutent les mêmes cycles d'instructions sans effet productif (progrès du système)
-

```
while(true)                    ||  while(true)
....if(b)                      ||  ....if(c)
........b := false             ||  ........c := false
........if(c)                  ||  ........if(b)
............c := false         ||  ............b := false
............x := x + 1         ||  ............x := x + 1
............c := true          ||  ............b := true
........b := true              ||  ........c := true
```

  avec b et c initialisé à `true`
- Il existe une séquence infinie (jamais bloquée) de la sémantique d'entrelacement dans laquelle `x` n'est jamais incrémentée.
- Solution:
  - modification des processus,
  - détection difficile (plus compliqué que les *deadlocks*, notion de "non-productivité").

Section critique et exclusion mutuelle

Définition

Une section critique est une ressource qui ne doit être utilisée que par un processus au plus.

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 ;;
```

- ces deux algorithmes permettent d'éviter les compétitions.

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 ;;
```

- ces deux algorithmes permettent d'éviter les compétitions.
- ces deux algorithmes produisent des attentes actives.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

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):

- *wait*(s) : si $s > 0$ alors $s := s - 1$, 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.
- 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.

```
while true do
begin
Mutex.lock m;
while predicat do
    Condition.wait c m;
done;
crit ();
Condition.signal c;
Mutex.unlock m;
suite ()
end
```

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 famine 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.

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 mange prend les deux baguettes autour de son assiette (la gauche, puis la doite), mange, dépose ses baguettes (la droite, puis la gauche) et se remet à penser.

- il existe une séquence de la sémantique d'entrelacement ou les philosophe sont tous bloqués avec la baguette de gauche dans la main.
- au delà du conte, cas courant de partage de plusieurs ressources.

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.

Modèles Concurrents: Processus et Threads

- dans les machines moderne, le système d'exploitation gère la concurrence,
- les programmes s'executent simultanément avec un système de processus
  - avec une mémoire propre,
  - ordonnancés par le système,
  - avec un coût élevé de changement de contexte.
- un processus léger *(thread)* un processus:
  - avec une mémoire partagée,
  - avec un coût plus faible de changement de contexte.
- les SEs utilisent des threads (appelés threads système)
- les applications gèrent (ordonnancent) leurs propres threads.
  - les langages de programmation usuels proposent des modèles concurrents spécifiques: primitives pour manipuler des threads, ordonnanceurs, communication entre threads, gestion de la mémoire, ...

Modèles Concurrents: Threads POSIX

- l'API des threads POSIX est une bibliothèque simple d'utilisation pour la gestion de threads en C

```
int compteur;
pthread_mutex_t mutc; // mutex de l'API
void* inc_compt(void *arg)
{
  pthread_mutex_lock(&mutc); // primitive de lock
  int temp = compteur;
  compteur = temp+1;
  pthread_mutex_unlock(&mutc); // primitive de relachement
}

int main(void)
{
  int i;
  pthread_t threads[NB_THREAD]; // threads de l'API
  void *status;

  pthread_mutex_init(&mutc,NULL); // primitive d'initialisation de mutex
  for(i=0;i<NB_THREAD;i++)
    pthread_create(&threads[i], NULL, inc_compt, NULL); // creation
  for(i=0;i<NB_THREAD;i++)
    pthread_join(threads[i],&status); // attente de terminaison

  pthread_mutex_destroy(&mutc); // liberation de mutex
}
```

Modèles Concurrents: Threads Java

- Chaque application a un thread principal qui peut créer d'autre threads à partir de *Runnable*.
- La synchronisation explicite sur un objet est possible, un seul thread peut opérer dans un bloc synchronisé sur un objet donné.
- Outil de haut-niveau pour éviter les écueils.

```
class incCompt implements Runnable {
    int v;
    Compteur compt,

    incCompt (Compteur c) {
          v = 0 ;
      compt = c;
    }

    public void run () {
          synchronized(compt){
            v = compt.get_valeur();
            v = v + 1;
            compt.set_valeur(v);
           }
        }
}
```

```
public static void main(String[] args) {
    Compteur c = new Compteur;
    incCompt t1 = new incCompt(c);
    incCompt t2 = new incCompt(c);
    t1.start();
    t2.start();
    }
```

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Modèles Concurrents: ARC en Rust

- En *Rust*, la propriété des ressources est vérifiée statiquement:
  - deux fonctions ne peuvent pas partager la propriété d'une même donnée.
- la concurrence est réalisée par des *atomically reference counted pointers* (ARC) (et des mutex) qui:
  - permettent de compter (RC) le nombre de fonctions qui accèdent à la donnée, et donc de la libérer quand c'est nécessaire.
  - garantissent l'atomicité de l'utilisation du pointeur.

Modèles Concurrents: ARC en Rust (II)

```
fn inc_compteur(c : Arc<Mutex<u32>>){ // c encapsule dans mutex + arc
    let mut c = c.lock().unwrap(); // ouverture de l'arc et lock du mutex
    *c += 1;
} // le mutex est signale a la fin du scope de c

fn main() {
    let compteur = Arc::new(Mutex::<u32>::new(0)); // encapsulation
    let mut poignees = Vec::new(); // handles des threads, utile pour join
    for _i in 1..11 {
        let compteur_clone = Arc::clone(&compteur); // clonage de l'arc
        poignees.push(std::thread::spawn(move || {
        inc_compteur(compteur_clone);
    })); // on passe une fermeture a spawn
    };

    for h in poignees {
        h.join().unwrap();
    }

}
```

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Conclusion

- Résumé:
  - l'exponentiation des états du a l'entrelacement, rend la programmation concurrente difficile.
  - techniques empiriques pour éviter les écueils.
- TD / TME:
  - TD: sémantique d'entrelacement, programmation de threads
  - TME: programmation de threads (C / Java / Rust)
  - travail personnel: consulter les APIs (POSIX / Thread / ARC)
- Séance prochaine:
  - modèles concurrents,
  - coopération.