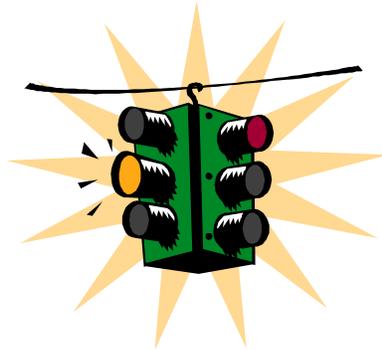


# Module 5 - Synchronisation de Processus

(ou threads, ou fils ou tâches)

---

## Chapitre 6 (Silberchatz)



# Problèmes avec concurrence = parallélisme

- Les threads concurrents doivent parfois partager des données (fichiers ou mémoire commune) et des ressources
  - ◆ On parle donc de tâches *coopératives*
- Si l'accès n'est pas contrôlé, le résultat de l'exécution du programme peut **dépendre de l'ordre d'entrelacement** de l'exécution des instructions (*non-déterministe*).
- Un programme peut fournir différents résultats à partir d'une exécution qui peuvent être parfois indésirables.

## Exemple

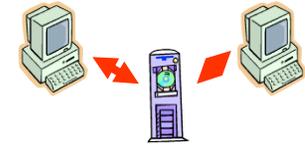
- Deux threads exécutent cette même procédure et partagent la même base de données
- Ils peuvent être interrompus n'importe quand
- Le résultat de l'exécution concurrente de P1 et P2 dépend de l'ordre de leur *entrelacement*

X demande une réservation d'avion

Base de données dit que le fauteuil A est disponible

Fauteuil A est assigné à X et marqué occupé

# Vue globale d'une exécution possible



P1

M. Leblanc demande une réservation d'avion

Base de données dit que fauteuil 30A est disponible

Fauteuil 30A est assigné à Leblanc et marqué occupé

Interruption  
ou retard

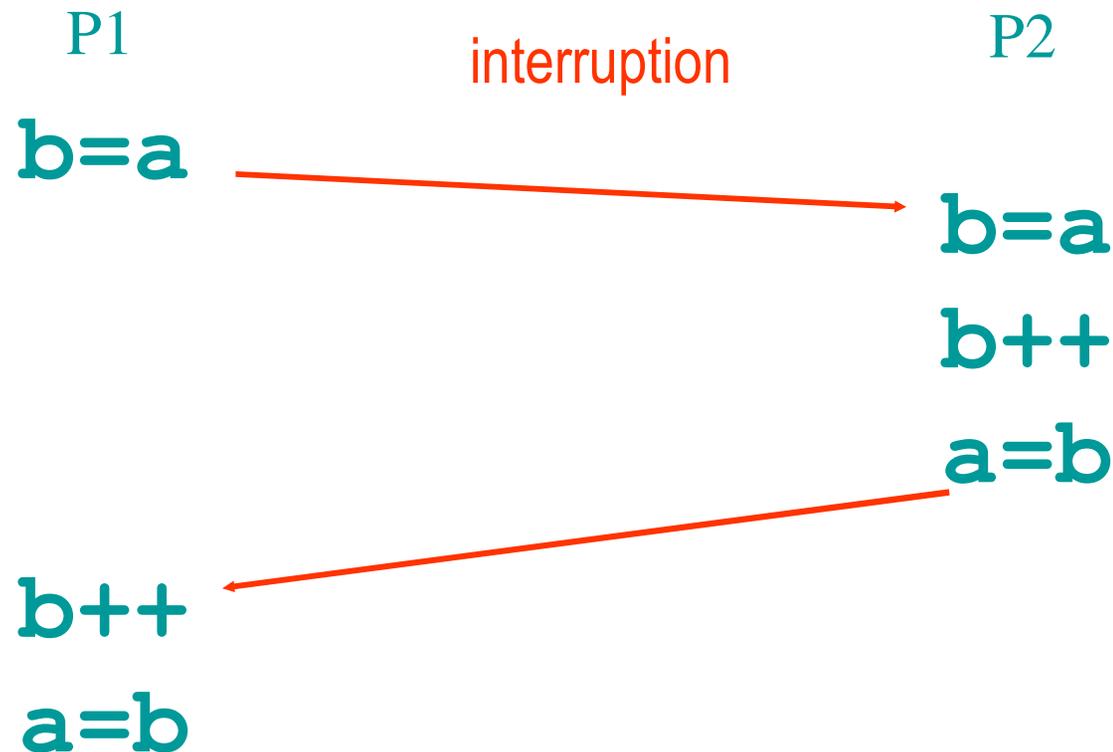
P2

M. Guy demande une réservation d'avion

Base de données dit que fauteuil 30A est disponible

Fauteuil 30A est assigné à Guy et marqué occupé

## Deux opérations en parallèle sur une variable a partagée (b est privé à chaque processus)



Supposons que a soit 0 au début

P1 travaille sur le plus vieux a donc le résultat final sera a=1.

Sera a=2 si les deux tâches sont exécutées l'une après l'autre

Si a était sauvegardé quand P1 est interrompu, il ne pourrait pas être partagé avec P2 (il y aurait deux a tandis que nous en voulons une seule)

## 3ème exemple

*Thread P1*

```
static char a;
```

```
void echo ()
```

```
{
```

```
    cin >> a;
```

```
    cout << a;
```

```
}
```

*Thread P2*

```
static char a;
```

```
void echo ()
```

```
{
```

```
    cin >> a;
```

```
    cout << a;
```

```
}
```

Si la var a est partagée, le premier a est effacé/remplacé  
Si elle est privée, l'ordre d'affichage est renversé

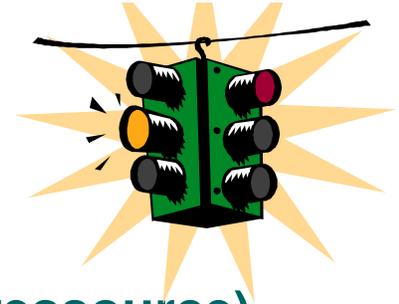
## Autres exemples

- **Des threads qui travaillent en simultanément sur une matrice, un pour la mettre à jour et l'autre pour en extraire des statistiques**
- **Problème qui affecte le programme du *tampon borné*, v. manuel**
- **Quand plusieurs threads exécutent en parallèle, nous ne pouvons pas faire d'hypothèse sur la vitesse d'exécution des threads, ni leur entrelacement**
  - ◆ **Peuvent être différents à chaque exécution du programme**

## Section Critique

- **Partie d'un programme dont l'exécution ne doit pas être *entrelacé* avec autres programmes**
- **Une fois qu'une tâche y entre, il faut lui permettre de terminer cette section sans permettre aux autres tâches de jouer sur les mêmes données**

# Le problème de la section critique



- Lorsqu'un thread manipule une donnée (ou ressource) partagée, nous disons qu'il se trouve dans une **section critique** (SC) (associée à cette donnée)
- Le problème de la section critique est de trouver un algorithme d'**exclusion mutuelle** de threads dans l'exécution de leur SCs afin que **le résultat de leurs actions ne dépendent pas de l'ordre d'entrelacement** de leur exécution (avec un ou plusieurs processeurs)
- L'exécution des sections critiques doit être **mutuellement exclusive**: à tout instant, **un seul** thread peut exécuter une SC pour une variable donnée (même lorsqu'il y a plusieurs processeurs)
- Ceci peut être obtenu en plaçant des **instructions spéciales** dans les sections d'entrée et sortie
- Pour simplifier les choses, assumons qu'il n'y a qu'une seule SC dans un programme.

# Structure du programme

- Chaque thread doit donc demander une permission avant d'entrer dans une section critique (SC)
- La section de code qui effectue cette requête est la **section d'entrée**
- La section critique est normalement suivie d'une **section de sortie**
- Le code qui reste est la **section restante (SR)**: non-critique

```
repeat
    section d'entrée
    section critique
    section de sortie
    section restante
forever
```

# Application

X demande une  
réservation d'avion

Section d'entrée

Base de données dit que  
fauteuil A est disponible

Fauteuil A est assigné à X et  
marqué occupé

Section de sortie

Section  
critique

# Critères nécessaires pour solutions valides

- **Exclusion Mutuelle:**
  - ◆ À tout instant, au plus un thread peut être dans une section critique (SC) pour une variable donnée
- **Progrès:**
  - ◆ absence d'interblocage (Chap 7)
  - ◆ si un thread demande d'entrer dans une section critique à un moment où aucun autre thread en fait requête, il doit être en mesure d'y entrer
  - ◆ Non interférence:
    - ☞ Si un thread s'arrête dans sa **section restante**, ceci ne doit pas affecter les autres threads
  - ◆ Mais on fait l'hypothèse qu'un thread qui entre dans une section critique, en sortira.
- **Attente limitée (bounded waiting):**
  - ◆ Aucun thread n'est empêché éternellement d'atteindre sa SC (pas de famine)

# Types de solutions

- **Solutions par logiciel**
  - ◆ des algorithmes dont la validité ne s'appuie pas sur l'existence d'instructions spéciales
- **Solutions fournies par le matériel**
  - ◆ s'appuient sur l'existence de certaines instructions (du processeur) spéciales
- **Solutions fournies par le SE**
  - ◆ procure certains appels du système au programmeur
- **Toutes les solutions se basent sur l'atomicité de l'accès à la mémoire centrale: une adresse de mémoire ne peut être affectée que par une instruction à la fois, donc par un thread à la fois.**
- *Plus en général, toutes les solutions se basent sur l'existence d'instructions atomiques, qui fonctionnent comme des SCs de base*

Atomicité = indivisibilité

# Solutions par logiciel

(pas pratiques, mais intéressantes pour comprendre le problème)

## ■ Considérons d'abord 2 threads

- ◆ Algorithmes 1 et 2 ne sont pas valides

  - ☞ Montrent la difficulté du problème

- ◆ Algorithme 3 est valide (algorithme de Peterson)

## ■ Notation

- ◆ Débutons avec 2 threads: T0 et T1

- ◆ Lorsque nous discutons de la tâche  $T_i$ ,  $T_j$  dénotera toujours une autre tâche différente ( $i \neq j$ )

# Algorithme 1: threads se donnent mutuellement le tour

- La variable partagée **turn** est initialisée à 0 ou 1
- La SC de  $T_i$  est exécutée si  $turn = i$
- $T_i$  est occupé à attendre si  $T_j$  est dans SC.
- Fonctionne pour l'exclusion mutuelle!
- Pas de famine (seulement 1 thread à son tour selon **turn**).
- Mais le critère de progrès n'est pas satisfait car l'exécution des SCs doit strictement alterner

Thread  $T_i$ :

```
repeat
```

```
    while (turn != i) {};
```

```
        SC
```

```
        turn = j;
```

```
        SR
```

```
forever
```

Rien  
faire



Ex 1:  $T_0$  possède une longue SR et  $T_1$  possède une courte SR. Si  $turn == 0$ ,  $T_0$  entre dans sa SC et puis sa SR ( $turn == 1$ ).  $T_1$  entre dans sa SC et puis sa SR ( $turn == 0$ ), et tente d'entrer dans sa SC: refusée! il doit attendre que  $T_0$  lui donne le tour.

## initialisation de turn à 0 ou 1

Thread T0:  
repeat

while (turn != 0) {};

SC

turn = 1;

SR

forever

Thread T1:  
repeat

while (turn != 1) {};

SC

turn = 0;

SR

forever

### Algorithme 1 vue globale

Ex 2: Généralisation à n threads: chaque fois, avant qu'un thread puisse entrer dans sa section critique, il lui faut attendre que tous les autres aient eu cette chance!

# Algorithme 2 ou l'excès de courtoisie...

- Une variable Booléenne par Thread: flag[0] et flag[1]
  - Ti signale qu'il désire exécuter sa SC par: flag[i] =vrai
  - Mais il n'entre pas si l'autre est aussi intéressé!
  - Exclusion mutuelle ok
  - Progrés ok
  - Absence de famine pas satisfait:
  - Considérez la séquence:
    - ◆ T0: flag[0] = vrai
    - ◆ T1: flag[1] = vrai
- ☞ Chaque thread attendra indéfiniment pour exécuter sa SC: on a une *famine*

Thread Ti:

repeat

flag[i] = vrai;

while (flag[j]==vrai) {};

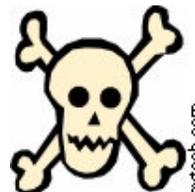
SC

flag[i] = faux;

SR

forever

rien faire



Thread T0:

repeat

flag[0] = vrai;

while (flag[1]==vrai) {};

SC

flag[0] = faux;

SR

forever

Après vous,  
monsieur

Thread T1:

repeat

flag[1] = vrai;

while (flag[0]==vrai) {};

SC

flag[1] = faux;

SR

forever

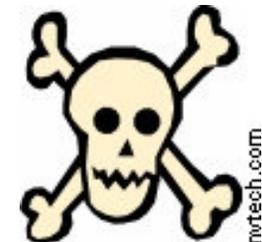
Après vous,  
monsieur

## Algorithme 2 vue globale

T0: flag[0] = vrai

T1: flag[1] = vrai

interblocage!



# Algorithme 3 (dit de Peterson): bon!

combine les deux idées: **flag[i]=intention d'entrer**; **turn=à qui le tour**

- **Initialisation:**
  - ◆  $\text{flag}[0] = \text{flag}[1] = \text{faux}$
  - ◆  $\text{turn} = i \text{ ou } j$
- **Désire d'exécuter SC est indiqué par  $\text{flag}[i] = \text{vrai}$**
- **$\text{flag}[i] = \text{faux}$  à la section de sortie**

```
Thread Ti:
repeat
    flag[i] = vrai;
    // je veux entrer
    turn = j;
    // je donne une chance à l'autre
do while
    (flag[j]==vrai && turn==j) {};
    SC
    flag[i] = faux;
    SR
forever
```

# Entrer ou attendre?

- **Thread  $T_i$  attend si:**
  - ◆  $T_j$  veut entrer est c'est la chance de  $T_j$ 
    - ☞ `flag[j]==vrai et turn==j`
- **Un thread  $T_i$  entre si:**
  - ◆  $T_j$  ne veut pas entrer ou c'est la chance de  $T_i$ 
    - ☞ `flag[j]==faux ou turn==i`
- **Pour entrer, un thread dépend de la bonne volonté de l'autre qu'il lui donne la chance!**

Thread T0:

repeat

```
flag[0] = vrai;  
// T0 veut entrer
```

```
turn = 1;
```

```
// T0 donne une chance à T1
```

while

```
(flag[1]==vrai&&turn=1) {};
```

SC

```
flag[0] = faux;
```

```
// T0 ne veut plus entrer
```

SR

forever

Thread T1:

repeat

```
flag[1] = vrai;
```

```
// T1 veut entrer
```

```
turn = 0;
```

```
// T1 donne une chance à 0
```

while

```
(flag[0]==vrai&&turn=0) {};
```

SC

```
flag[1] = faux;
```

```
// T1 ne veut plus entrer
```

SR

forever

## Algorithme de Peterson vue globale

# Scénario pour le changement de contrôle

Thread T0:

```
...  
SC  
flag[0] = faux;  
// T0 ne veut plus entrer  
SR  
...
```

Thread T1:

```
...  
flag[1] = vrai;  
// T1 veut entrer  
turn = 0;  
// T1 donne une chance à T0  
while  
(flag[0]==vrai&&turn=0) {};  
//test faux, entre  
...
```

T1 prend la relève, donne une chance à T0 mais T0 a dit qu'il ne veut pas entrer.  
T1 entre donc dans la SC

# Autre scénario de changement de contrôle

Thread T0:

Thread T1:

```
SC
flag[0] = faux;
// T0 ne veut plus entrer
SR
flag[0] = vrai;
// T0 veut entrer
turn = 1;
// T0 donne une chance à T1
while
(flag[1]==vrai&&turn=1) {};
// test vrai, n'entre pas

flag[1] = vrai;
// T1 veut entrer
turn = 0;
// T1 donne une chance à T0
// mais T0 annule cette action
while
(flag[0]==vrai&&turn=0) {};
//test faux, entre
```

T0 veut rentrer mais est obligé de donner une chance à T1, qui entre

# Mais avec un petit décalage, c'est encore T0!

Thread T0:

```
SC
flag[0] = faux;
// 0 ne veut plus entrer
RS
flag[0] = vrai;
// 0 veut entrer
turn = 1;
// 0 donne une chance à 1
// mais T1 annule cette action
while
(flag[1]==vrai&&turn=1) {};
// test faux, entre
```

Thread T1:

```
flag[1] = vrai;
// 1 veut entrer
turn = 0;
// 1 donne une chance à 0
while
(flag[0]==vrai&&turn=0) {};
// test vrai, n'entre pas
```

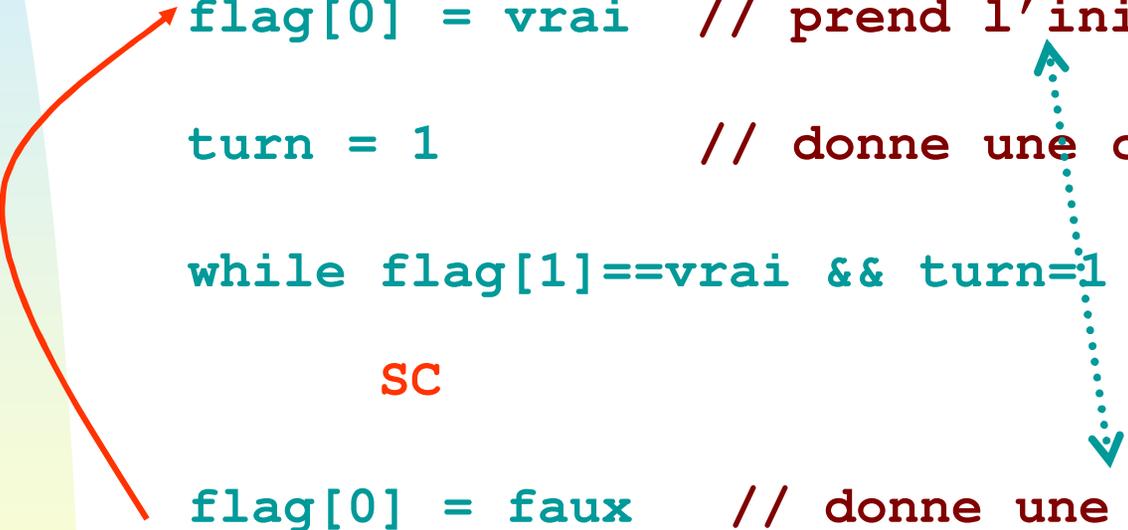
Si T0 et T1 tentent simultanément d'entrer dans SC, seule une valeur de turn survivra:

non-déterministe (on ne sait pas qui gagnera), mais l'exclusion fonctionne

# Donc cet algorithme n'oblige pas une tâche d'attendre d'autres qui pourraient ne pas avoir besoin de la SC

Supposons que T0 soit le seul à avoir besoin de la SC, ou que T1 soit lent à agir: T0 peut rentrer de suite (`flag[1]==faux` la dernière fois que T1 est sorti)

```
flag[0] = vrai // prend l'initiative
turn = 1 // donne une chance à l'autre
while flag[1]==vrai && turn==1 {} //test faux, entre
SC
flag[0] = faux // donne une chance à l'autre
```



Cette propriété est désirable

## Algorithme 3: preuve de validité

- **Exclusion mutuelle est assurée car:**
  - ◆ T0 et T1 ne peuvent être tous deux dans SC seulement si **turn** est simultanément égal à 0 et 1 (ce qui est impossible)
- **Démontrons que progrès et attente limitée sont satisfaits:**
  - ◆ Ti ne peut pas entrer dans SC seulement si en attente dans la boucle while() avec condition:  $\text{flag}[j] == \text{vrai}$  et  $\text{turn} = j$ .
  - ◆ Si Tj ne veut pas entrer dans SC alors  $\text{flag}[j] = \text{faux}$  et Ti peut alors entrer dans SC

## Algorithme 3: preuve de validité (cont.)

- ◆ Si  $T_j$  a effectué  $\text{flag}[j]=\text{vrai}$  et se trouve dans le  $\text{while}()$ , alors  $\text{turn}==i$  ou  $\text{turn}==j$
- ◆ Si
  - ☞  $\text{turn}==i$ , alors  $T_i$  entre dans SC.
  - ☞  $\text{turn}==j$  alors  $T_j$  entre dans SC mais il mettra  $\text{flag}[j]=\text{false}$  à la sortie: permettant à  $T_i$  d'entrer dans SC
- ◆ mais si  $T_j$  a le temps de faire  $\text{flag}[j]=\text{true}$ , il devra aussi faire  $\text{turn}=i$
- ◆ Puisque  $T_i$  ne peut modifier  $\text{turn}$  dans le  $\text{while}()$ ,  $T_i$  entrera dans SC après au plus une entrée dans SC par  $T_j$  (attente limitée)

## A propos d'échecs des threads

- **La solution qui satisfait les 3 critères (EM, progrès et attente limitée), procure une robustesse face à l'échec d'un thread dans sa section restante (SR)**
  - ◆ un thread qui échoue dans sa SR est équivalent à un thread ayant une SR infiniment longue...
- **Par contre, aucune solution valide ne procure pas de robustesse face à l'échec d'un thread dans sa section critique (SC)**
  - ◆ un thread  $T_i$  qui échoue dans sa SC n'envoie pas de signal aux autres threads: donc pour eux  $T_i$  est encore dans sa SC...

## Extension à $>2$ threads

- **L 'algorithme de Peterson peut être généralisé à plus de 2 threads**
- **Mais dans ce cas il existe des algorithmes plus élégants, comme l'algorithme du boulanger, basée sur l'idée de 'prendre un numéro'...**
  - ◆ Pas le temps d'en parler...

## Une leçon à retenir...

- **À fin que des threads avec des variables partagées puissent réussir, il est nécessaire que tous les threads impliqués utilisent le même algorithme de coordination**
  - ◆ Un protocole commun

# Critique des solutions par logiciel

- **Difficiles à programmer! Et à comprendre!**
  - ◆ Les solutions que nous verrons dorénavant sont toutes basées sur l'existence d'instructions spécialisées, qui facilitent le travail.
- **Les threads qui requièrent l'entrée dans leur SC sont occupés à attendre (busy waiting); consommant ainsi du temps de processeur**
  - ◆ Pour de longues sections critiques, il serait préférable de bloquer les threads qui doivent attendre...

# Solutions matérielles: désactivation des interruptions

- Sur un monoprocesseur: l'exclusion mutuelle est préservée mais l'efficacité se détériore: Lorsqu'un processus est dans sa SC il est impossible d'entrelacer l'exécution d'autres processus dans leur SR
- Perte d'interruptions
- Dans un multiprocesseur: l'exclusion mutuelle n'est pas garantie (long délais et différentes horloges)
- Une solution qui n'est généralement pas acceptable

Process  $P_i$ :

repeat

  inhiber interrupt

  section critique

  rétablir interrupt

  section restante

forever

# Solutions matérielles: Instructions spécialisées

- **Normal:** pendant qu'un thread ou processus accède une adresse de mémoire, aucun autre ne peut accéder cette adresse en même temps
- **Extension:** instructions de machine exécutant plusieurs actions (ex: lecture et écriture) sur la même location de mémoire de manière **atomique (indivisible)**
- **Une instruction atomique ne peut être exécutée que par un thread/processus à la fois (même en présence de plusieurs processeurs)**

# L'instruction test-and-set

- Une version C++ de test-and-set:

```
bool testset(int& i)
{
    if (i==0) {
        i=1;
        return true;
    } else {
        return false;
    }
}
```

↑  
**Instruction atomique!**

- Un algorithme utilisant testset pour l'exclusion Mutuelle:
- La variable partagée b est **initialisée à 0**
- Le 1er Ti/Pi qui met b à 1 qui entre dans SC

Tâche Ti/Pi:

```
while testset(b)==false {};  
    SC //entre quand vrai  
b=0;  
    SR
```

## L'instruction test-and-set (cont.)

- **L'exclusion mutuelle est assurée: si  $T_i$  entre dans SC, l'autre  $T_j$  est occupé à attendre**
  - ◆ Problème: utilise encore *occupé à attendre*
- **Peut procurer facilement l'exclusion mutuelle mais nécessite des algorithmes plus complexes pour satisfaire les autres exigences du problème de la section critique**
- **Lorsque  $T_i$  sort de sa SC, la sélection du prochain  $T_j$  qui entrera dans SC est arbitraire: pas de limite d'attente et donc possibilité de famine**

## L'instruction 'Échange'

- Certains UCTs (ex: Pentium) offrent une instruction `xchg(a,b)` qui interchange *atomiquement* le contenu de a et b.
- Mais `xchg(a,b)` souffre des même lacunes que `test-and-set`

# Utilisation de xchg pour l'exclusion mutuelle (Stallings)

- La variable *partagée* **b** est **initialisée à 0**
- Chaque  $T_i$  possède une variable *locale* **k**
- Le  $T_i$  pouvant entrer dans sa SC est celui qui trouve **b=0**
- Ce  $T_i$  exclue tous les autres en attribuant **b à 1**
  - ◆ Quand la SC est occupée, **k** et **b** seront à 1 pour toutes tâches cherchant à entrer dans leur SC
  - ◆ Mais **k** est 0 pour le thread qui est dans la SC

usage:

```
Thread  $T_i$ :  
repeat  
    k = 1  
    while (k != 0) xchg (k, b) ;  
    SC  
    xchg (k, b) ;  
    SR  
forever
```

## Solutions basées sur des instructions fournies par le SE (appels du système)

- **Les solutions vues jusqu'à présent sont difficiles à programmer et mal codées.**
- **On a besoin de méthodes qui nous évitent les erreurs communes, comme la famine, l'impasse, etc.**
  - ◆ Besoin d'instructions à plus haut niveau
- **Les méthodes que nous verrons dorénavant utilisent des instructions puissantes, qui sont réalisées par des appels de SE (system calls)**

# Sémaphores

- **Un sémaphore S est un entier qui, sauf durant l'initialisation, est accessible seulement par ces 2 opérations atomiques et mutuellement exclusives:**
  - ◆ wait(S)
  - ◆ signal(S)
- **Il est partagé entre tous les processus qui s'intéressent à la même section critique**
- **Les sémaphores seront catégorisés en deux types:**
  - ◆ Les sémaphores qui sont occupés à attendre (busy waiting)
  - ◆ Les sémaphores qui utilisent des files d'attente
- **On distingue aussi les sémaphores compteurs et les sémaphores binaires, ces derniers sont moins puissants (v. livre).**

## Spinlocks d'Unix: Sémaphores occupés à attendre

(busy waiting)

- La façon la plus simple d'implanter les sémaphores.
- Utiles pour des situations où l'attente est brève, où il y a beaucoup d'UCTs
- **S** est un entier **initialisé à une valeur positive**, de façon qu'un premier thread puisse entrer dans la SC
- Quand  $S > 0$ , jusqu'à  $n$  threads peuvent entrer
- Quand  $S \leq 0$ , il faut attendre  $S+1$  signals (d'autres threads) pour entrer

```
wait (S) :  
while S <= 0 {};  
S--;
```

*Attend si le nb de threads qui peuvent entrer = 0 ou négatif*

```
signal (S) :  
S++;
```

*Augmente de 1 le nb des threads qui peuvent entrer*

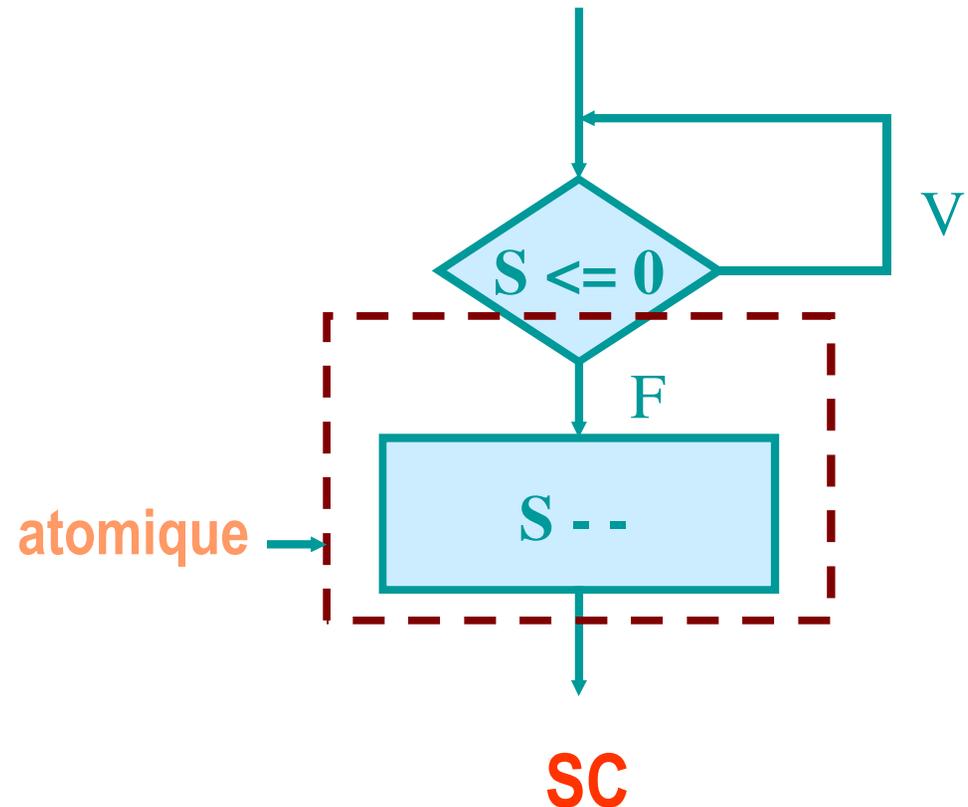
# Atomicité

*Wait:* La séquence test-décroissement est atomique, mais pas la boucle!

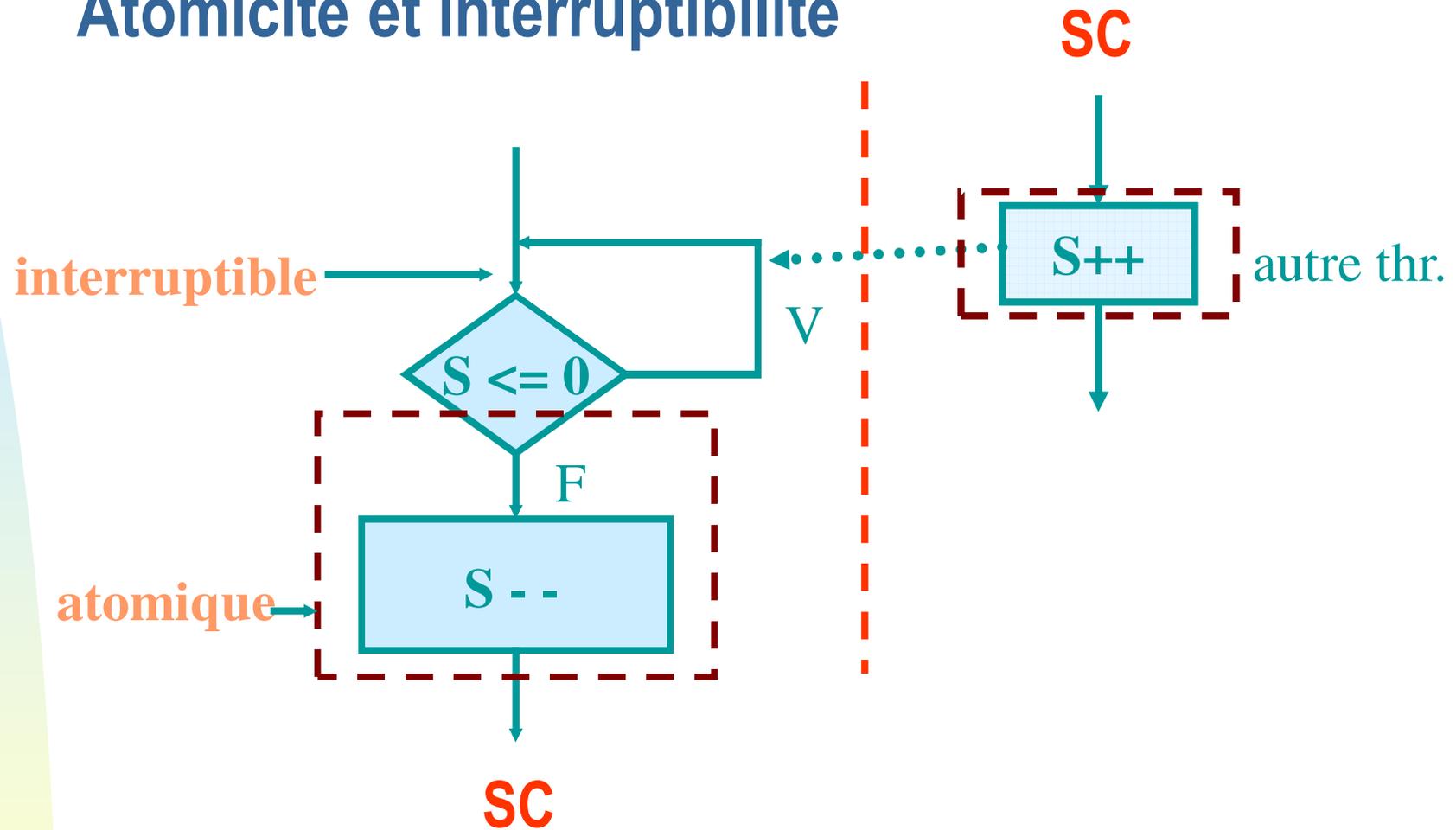
*Signal* est atomique.

Rappel: les sections atomiques ne peuvent pas être exécutées simultanément par différents threads

(ceci peut être obtenu en utilisant un des mécanismes précédents)



# Atomicité et interruptibilité



La boucle n'est pas atomique pour permettre à un autre thread d'interrompre l'attente sortant de la SC

# Utilisation des sémaphores pour les sections critiques

- **Pour n threads**
- **Initialiser S à 1**
  - ◆ Alors 1 seul thread peut être dans sa SC
- **Pour permettre à k threads d'entrer dans leur SC, initialiser S à k**

```
Thread Ti:  
repeat  
    wait (S) ;  
    SC  
    signal (S) ;  
    SR  
forever
```

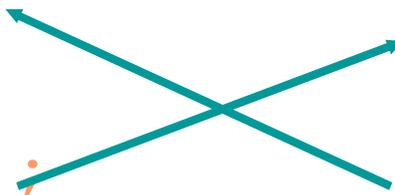
Initialise S à  $\geq 1$

Thread T1:

```
repeat  
wait (S) ;  
  SC  
signal (S) ;  
  SR  
forever
```

Thread T2:

```
repeat  
wait (S) ;  
  SC  
signal (S) ;  
  SR  
forever
```



Semaphores: vue globale

*Peut être facilement généralisé à plusieurs threads*

# Utilisation des sémaphores pour la synchronisation de threads

- On a 2 threads: T1 et T2
- La commande S1 dans T1 doit être exécuté avant celle de S2 dans T2
- Définissons un sémaphore S
- Initialisons S à 0
- Synchronisation correcte lorsque T1/P1 contient:  
S1;  
signal(S);
- et que T2/P2 contient:  
wait(S);  
S2;

# Interblocage et famine avec les sémaphores

- **Famine:** un thread peut ne jamais arriver à exécuter s'il ne teste jamais le sémaphore au bon moment
- **Inter-blocage:** Supposons S et Q initialisés à 1

T0

wait(S)

wait(Q)

T1

wait(Q)

wait(S)



# Sémaphores: observations

```
wait (S) :  
while S<=0 {};  
S--;
```

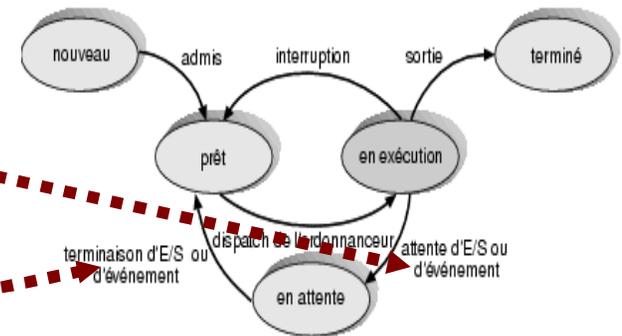
- **Quand  $S \geq 0$ :**
  - ◆ Le nombre de threads qui peuvent exécuter `wait(S)` sans devenir bloqués =  $S$ 
    - ☞  $S$  threads peuvent entrer dans la SC
    - ☞ plus puissant que certains mécanismes déjà vus
    - ☞ dans les solutions où  $S$  peut être  $>1$  il faudra avoir un 2ème sém. pour les faire entrer un à la fois (exclusion mutuelle ou mutex)
- **Quand  $S$  est  $> 1$ , le thread qui entre le premier dans la SC est le premier à tester  $S$  (choix aléatoire)**
  - ◆ ceci n'est plus vrai dans la solution suivante
- **Quand  $S < 0$ : le nombre de threads qui attendent sur  $S$  est  $= |S|$  - Ne s'applique pas pour sémaphores occupés à attendre**

# Comment éviter l'attente occupée et le choix aléatoire dans les sémaphores

- Quand un thread doit attendre qu'un sémaphore soit plus grand que 0, il est mis dans une file d'attente de threads qui postulent sur le même sémaphore.
- Les files peuvent être PAPS (FIFO), avec priorités, etc. Le SE contrôle l'ordre dans lequel les threads sont choisis de la queue.
- *wait* et *signal* deviennent donc des appels de système **comme les appels à des opérations d'E/S.**
- Il y a une file d'attente pour chaque sémaphore comme il y a une file d'attente pour chaque unité d'E/S.

# Sémaphores sans attente occupée

- Un sémaphore **S** devient une structure de données:
  - ◆ Une valeur (**S.V**)
  - ◆ Une liste d'attente **L** (**S.L**)
- Un thread devant attendre un sémaphore **S**, est bloqué et **ajouté** dans la file d'attente **S.L** du sémaphore (v. état bloqué = attente chap 3).



- **signal(S) enlève** (selon une politique choisie, ex: PAPS/FIFO) un thread de **S.L** et le place sur la liste des threads prêts/ready.

# Implementation

(les boîtes représentent des séquences non-interruptibles)

```
wait(S):      S.value --;  
    if S.value < 0 {           // SC occupée  
        add this thread to S.L;  
        block // thread mis en état attente (wait)  
    }
```

```
signal(S): S.value ++;  
    if S.value ≤ 0 {           // des threads attendent  
        remove a process P from S.L;  
        wakeup(P) // thread choisi devient prêt  
    }
```

**S.value** doit être initialisé à une valeur non-négative (dépendant de l'application, v. exemples)

## Wait et signal contiennent sont aussi des SC!

- Les opérations *wait* et *signal* doivent être exécutées atomiquement (un seul thread à la fois)
- Dans un système à 1 seule UCT, on peut neutraliser les interruptions et les opérations sont courtes (moins de 10 instructions)
- Normalement, nous devons utiliser un des mécanismes déjà vus (instructions spéciales, algorithme de Peterson, etc.)
- L'attente occupée dans ce cas ne sera pas trop couteuse car *wait* et *signal* sont brefs

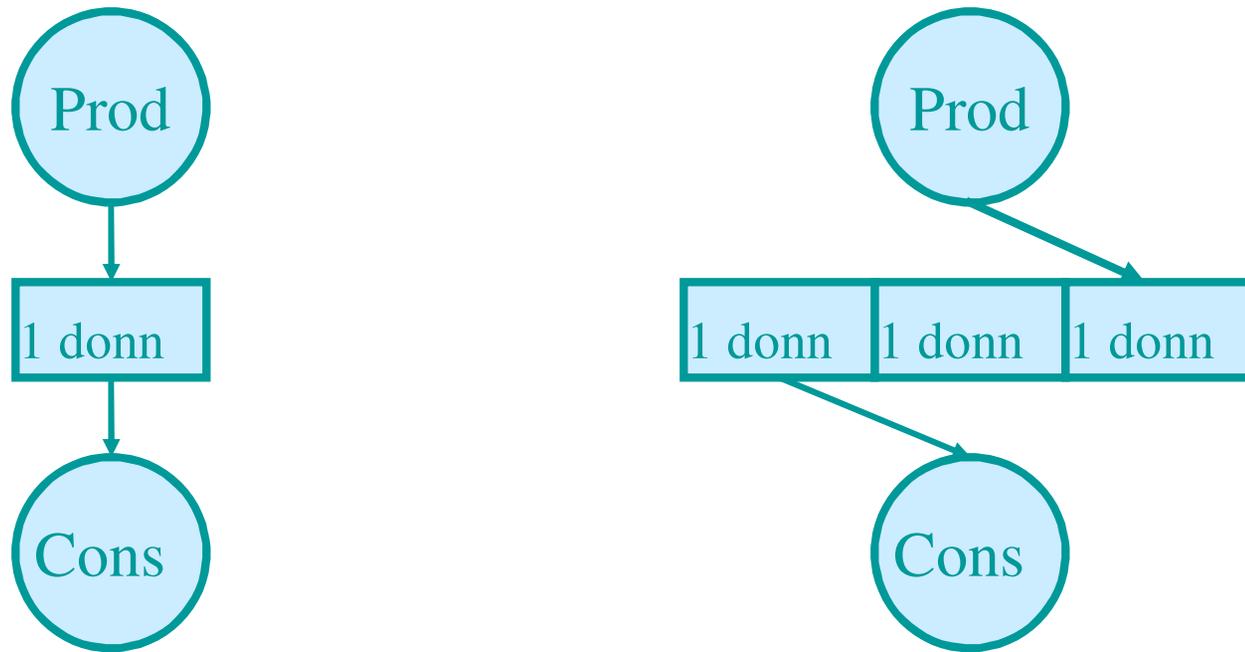
# Problèmes classiques de synchronisation

- **Tampon borné (producteur-consommateur)**
- **Rédacteurs - Lecteurs**
- **Les philosophes mangeant**

# Le problème du producteur - consommateur

- **Un problème classique dans l'étude des threads communicants**
  - ◆ un thread *producteur* produit des données (p.ex. des enregistrements d'un fichier) pour un thread *consommateur*

# Tampons de communication

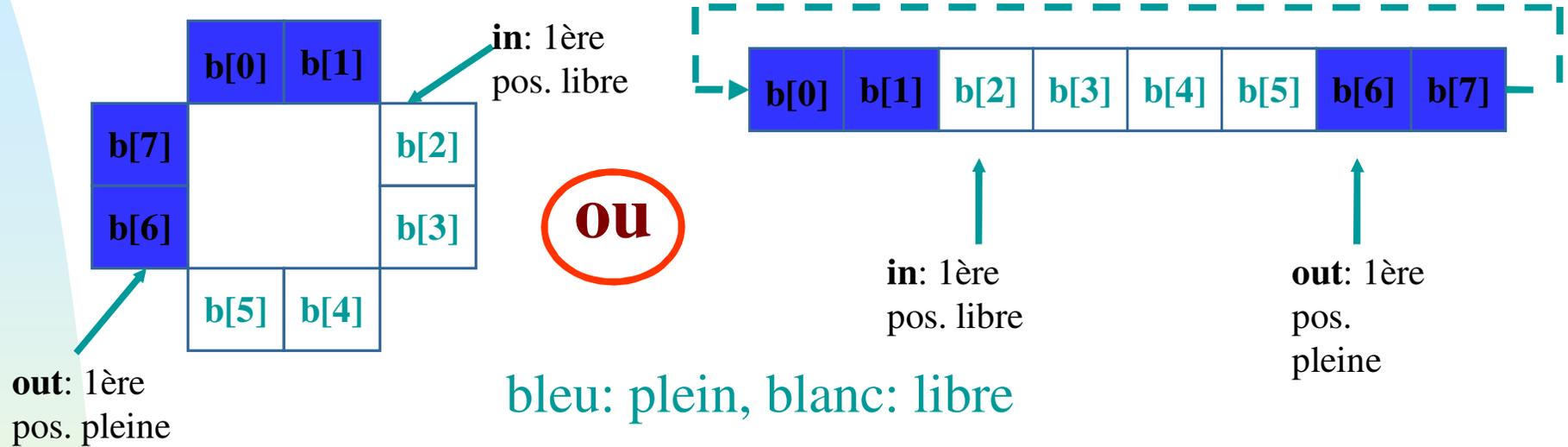


Si le tampon est de longueur 1, le producteur et consommateur doivent forcément aller à la même vitesse

Des tampons de plus grandes longueur permettent une certaine indépendance. P.ex. à droite le consommateur a été plus lent

# Le tampon borné (bounded buffer)

une structure de données fondamentale dans les SE



Le tampon borné se trouve dans la mémoire partagée entre consommateur et usager

## Pb de sync entre threads pour le tampon borné

- **Étant donné que le producteur et le consommateur sont des threads indépendants, des problèmes peuvent se produire dans le cas d'accès simultané du tampon**
- **Les sémaphores peuvent résoudre ce problème**

# Sémaphores: rappel.

- **Soit S un sémaphore sur une SC**
  - ◆ il est associé à une file d'attente
  - ◆ S positif: S threads peuvent entrer dans SC
  - ◆ S zéro: aucun thread ne peut entrer, pas de thread en attente
  - ◆ S négatif: |S| thread dans file d'attente
- **Wait(S): S - -**
  - ◆ si après  $S \geq 0$ , thread peut entrer dans SC
  - ◆ si  $S < 0$ , thread est mis dans file d'attente
- **Signal(S): S++**
  - ◆ si après  $S \leq 0$ , il y avait des threads en attente, et un thread est réveillé
- **Indivisibilité = atomicité de ces opérations**

## Solution avec sémaphores

- Un sémaphore **S** pour l'**exclusion mutuelle** de l'accès du tampon
  - ◆ Les sémaphores suivants ne font pas l'EM
- Un sémaphore **N** pour synchroniser le producteur et consommateur avec le **nombre d'éléments consommables** dans le tampon
- Un sémaphore **E** pour synchroniser le producteur et consommateur avec le **nombre d'espaces libres**

## Solution de P/C: tampon circulaire fini de dimension k

```
Initialization: S.count=1; //excl. mut.  
                N.count=0; //espaces pleins  
                E.count=k; //espaces vides
```

```
append(v) :  
    b[in]=v;  
    In++ mod k;
```

```
take() :  
    w=b[out];  
    Out++ mod k;  
    return w;
```

Producer:

repeat

produce v;

wait (E);

wait (S);

■ append (v);

signal (S);

signal (N);

forever

Consumer:

repeat

wait (N);

wait (S);

■ w=take ();

signal (S);

signal (E);

consume (w);

forever

■ Sections critiques

## Points importants à étudier

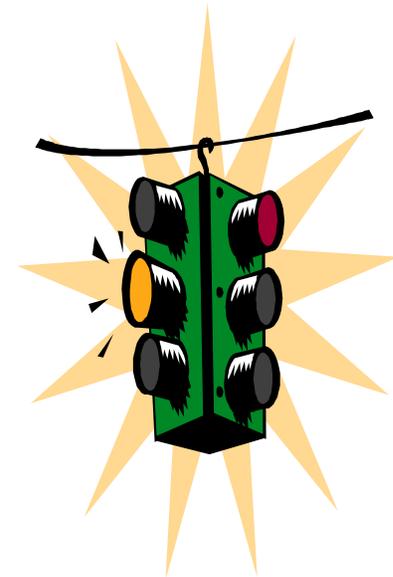
- **dégâts possibles en interchangeant les instructions sur les sémaphores**
  - ◆ ou en changeant leur initialisation
- **Généralisation au cas de plusieurs producteurs et consommateurs**

## Concepts importants de cette partie du Chap 6

- **Le problème de la section critique**
- **L'entrelacement et l'atomicité**
- **Problèmes de famine et interblocage**
- **Solutions logiciel**
- **Instructions matériel**
- **Sémaphores occupés ou avec files**
- **Fonctionnement des différentes solutions**
- **L'exemple du tampon borné**

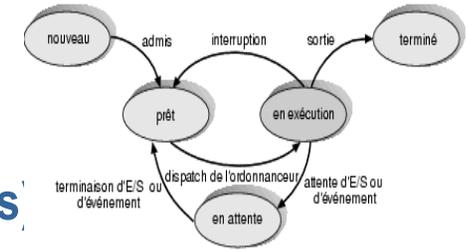
## Quelques exemples

- **Problèmes classiques de synchronisation**
- **Lecteurs - Rédacteurs**
- **Les philosophes mangeant**
- **Moniteurs**



# Sémaphores: rappel

(les boîtes représentent des séquences non-interruptibles)



```
wait(S): S.value --;  
    if S.value < 0 {           // SC occupée  
        ajouter ce thread à S.L;  
        block                 // thread mis en état attente (wait)  
    }
```

```
signal(S): S.value ++;  
    if S.value ≤ 0 {         // des threads attendent  
        enlever un thread P de S.L;  
        wakeup(P) // thread choisi devient prêt  
    }
```

S.value doit être initialisé à une valeur non-négative

dépendant de l'application, v. exemples

# Sémaphores: rappel.

- **Soit S un sémaphore sur une SC**
  - ◆ il est associé à une file d 'attente
  - ◆ S positif: S thread peuvent entrer dans SC
  - ◆ S zéro: aucun thread ne peut entrer, aucun thread en attente
  - ◆ S négatif: |S| thread dans file d 'attente
- **Wait(S): S - -**
  - ◆ si après  $S \geq 0$ , thread peut entrer dans SC
  - ◆ si  $S < 0$ , thread est mis dans file d 'attente
- **Signal(S): S++**
  - ◆ si après  $S \leq 0$ , il y avait des threads en attente, et un thread est transféré à la file prêt
- **Indivisibilité = atomicité de wait et signal**

## Problème des lecteurs - rédacteurs

- **Plusieurs threads peuvent accéder à une base de données**
  - ◆ Pour y lire ou pour y écrire
- **Les rédacteurs doivent être synchronisés entre eux et par rapport aux lecteurs**
  - ◆ il faut empêcher à un thread de lire pendant l'écriture
  - ◆ il faut empêcher à deux rédacteurs d'écrire simultanément
- **Les lecteurs peuvent y accéder simultanément**

# Une solution (n'exclut pas la famine)

- **Variable readcount:** nombre de threads lisant la base de données
- **Sémaphore mutex:** protège la SC où readcount est mis à jour
- **Sémaphore wrt:** exclusion mutuelle entre rédacteurs et lecteurs
- **Les rédacteurs doivent attendre sur wrt**
  - ◆ les uns pour les autres
  - ◆ et aussi la fin de toutes les lectures
- **Les lecteurs doivent**
  - ◆ attendre sur wrt quand il y a des rédacteurs qui écrivent
  - ◆ bloquer les rédacteurs sur wrt quand il y a des lecteurs qui lisent
  - ◆ redémarrer les rédacteurs quand personne ne lit

# Les données et les rédacteurs

Données: deux sémaphores et une variable

```
mutex, wrt: semaphore (init. 1);  
readcount : integer (init. 0);
```

Rédacteur

```
wait (wrt);  
    . . .  
    // écriture  
    . . .  
signal (wrt);
```

# Les lecteurs

```
wait(mutex);  
    readcount ++ ;  
    if readcount == 1 then wait(wrt);  
signal(mutex);
```

**//SC: lecture**

```
wait(mutex);  
    readcount -- ;  
    if readcount == 0 then signal(wrt);  
signal(mutex);
```

↑  
Le premier lecteur d'un groupe pourrait devoir attendre sur wrt, il doit aussi bloquer les rédacteurs. Quand il sera entré, les suivants pourront entrer librement

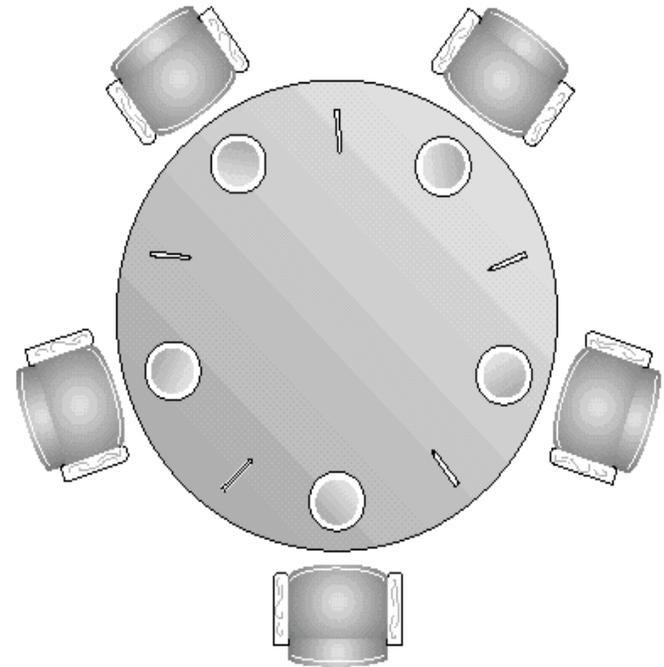
↑  
Le dernier lecteur sortant doit permettre l'accès aux rédacteurs

## Observations

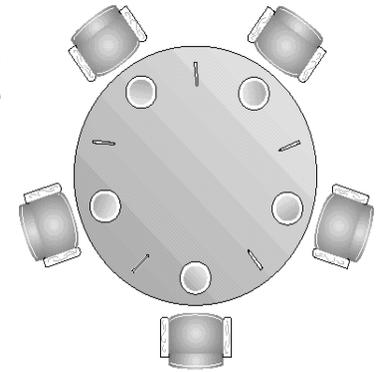
- **Le 1er lecteur qui entre dans la SC bloque les rédacteurs (wait (wrt)), le dernier les remet en marche (signal (wrt))**
- **Si 1 rédacteur est dans la SC, 1 lecteur attend sur wrt, les autres sur mutex**
- **un signal(wrt) peut faire exécuter un lecteur ou un rédacteur**

# Le problème des philosophes mangeant

- **5 philosophes qui mangent et pensent**
- **Pour manger il faut 2 fourchettes, droite et gauche**
- **On en a seulement 5!**
- **Un problème classique de synchronisation**
- **Illustre la difficulté d'allouer ressources aux threads tout en évitant interblocage et famine**



# Le problème des philosophes mangeant

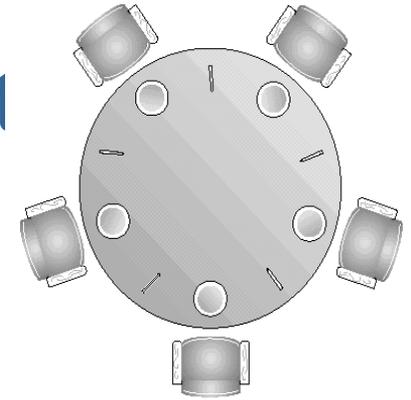


- **Un thread par philosophe**
- **Un sémaphore par fourchette:**
  - ◆ fork: array[0..4] of semaphores
  - ◆ Initialisation: fork[i] = 1 for i:=0..4
- **Première tentative:**
  - ◆ interblocage si chacun débute en prenant sa fourchette gauche!
    - ☞ `Wait (fork [i])`

```
Thread Pi:  
repeat  
  think;  
  wait (fork [i]);  
  wait (fork [i+1 mod 5]);  
  eat;  
  signal (fork [i+1 mod 5]);  
  signal (fork [i]);  
forever
```

# Le problème des philosophes mangeant

- Une solution: **admettre seulement 4 philosophes à la fois** qui peuvent tenter de manger
- Il y aura touj. au moins 1 philosophe qui pourra manger
  - ◆ même si tous prennent 1 fourchette
- Ajout d'un sémaphore T qui limite à 4 le nombre de philosophes "assis à la table"
  - ◆ initial. de T à 4



```
Thread Pi:  
repeat  
  think;  
  wait (T);  
  wait (fork[i]);  
  wait (fork[i+1 mod 5]);  
  eat;  
  signal (fork[i+1 mod 5]);  
  signal (fork[i]);  
  signal (T);  
forever
```

## Avantage des sémaphores (par rapport aux solutions précédentes)

- **Une seule variable partagée par section critique**
- **deux seules opérations: wait, signal**
- **contrôle plus localisé (que avec les précédés)**
- **extension facile au cas de plus. threads**
- **possibilité de faire entrer plus. threads à la fois dans une section critique**
- **gestion de files d`attente par le SE: famine évitée si le SE est équitable (p.ex. files FIFO)**

## Problème avec sémaphores: difficulté de programmation

- **wait et signal sont dispersés parmi plusieurs threads, mais ils doivent se correspondre**
  - ◆ V. programme du tampon borné
- **Utilisation doit être correcte dans tous les threads**
- **Un seul “mauvais” thread peut faire échouer toute une collection de threads (p.ex. oublie de faire signal)**
- **Considérez le cas d`un thread qui a des waits et signals dans des boucles et des tests...**

## Moniteurs: une autre solution

- **Constructions (en langage de haut-niveau) qui procurent une fonctionnalité équivalente aux sémaphores mais plus facile à contrôler**
- **Disponibles en:**
  - ☞ Concurrent Pascal, Modula-3...
    - *synchronized method* en Java (moniteurs simplifiés)

# Moniteur

- **Est un module contenant:**
  - ◆ une ou plusieurs procédures
  - ◆ une séquence d'initialisation
  - ◆ variables locales
- **Caractéristiques:**
  - ◆ variables locales accessibles seulement à l'aide d'une procédure du moniteur
  - ◆ un thread entre dans le moniteur en invoquant une de ses procédures
  - ◆ *un seul thread peut exécuter dans le moniteur à tout instant* (mais plus. threads peuvent être en attente dans le monit.)

# Moniteur

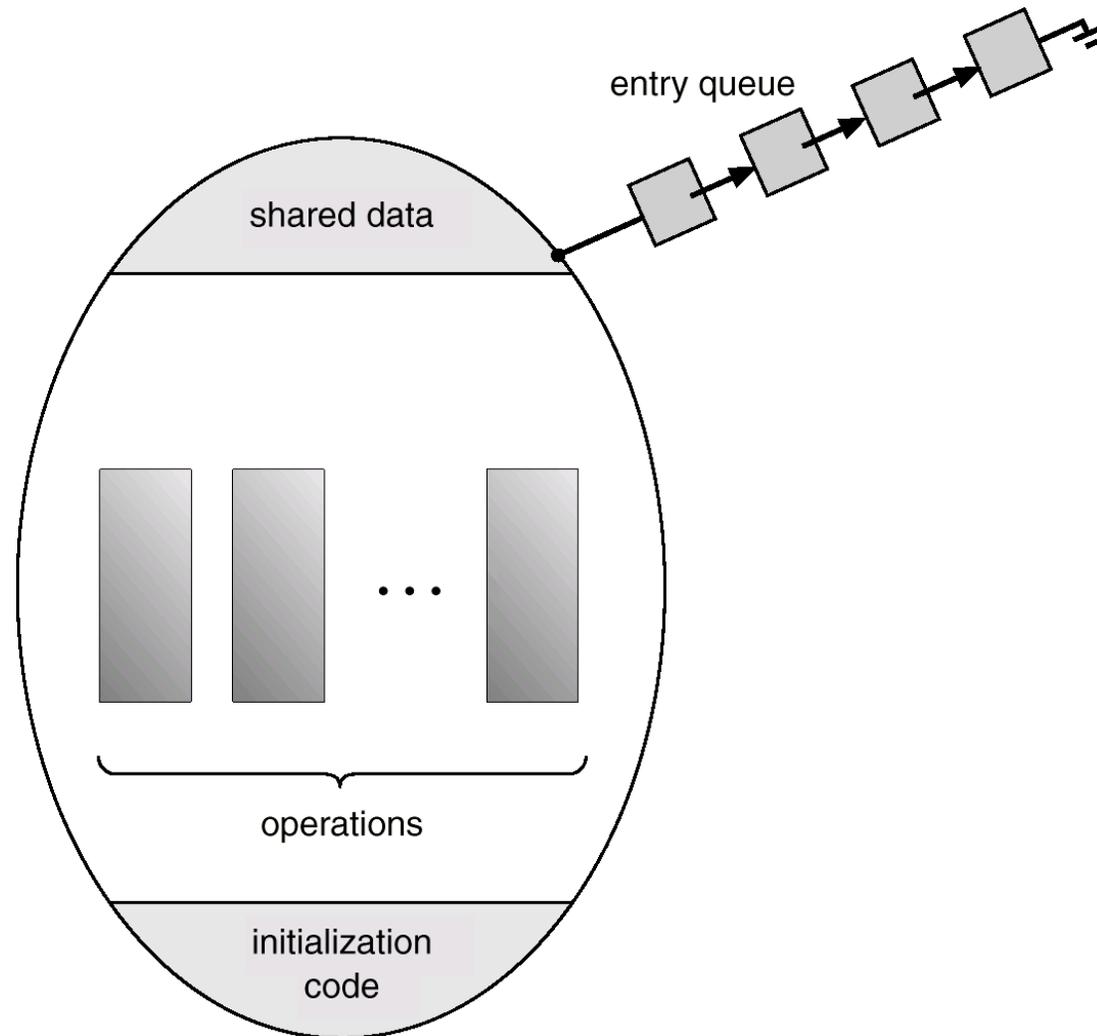
- **Il assure à lui seul l'exclusion mutuelle: pas besoins de le programmer explicitement**
- **On assure la protection des données partagées en les plaçant dans le moniteur**
  - ◆ Le moniteur verrouille les données partagées lorsqu'un thread y entre
- **Synchronisation de threads est effectuée en utilisant des **variables conditionnelles** qui représentent des conditions après lesquelles un thread pourrait attendre avant d'exécuter dans le moniteur**

# Structure générale du moniteur (style Java)

```
monitor nom-de-moniteur
{ // déclarations de vars
    public entry p1(. . .) {code de méthode p1}
    public entry p2(. . .) {code de méthode p2}
    . . .
}
```

La seule façon de manipuler les vars internes au moniteur est d'appeler une des méthodes d'entrée

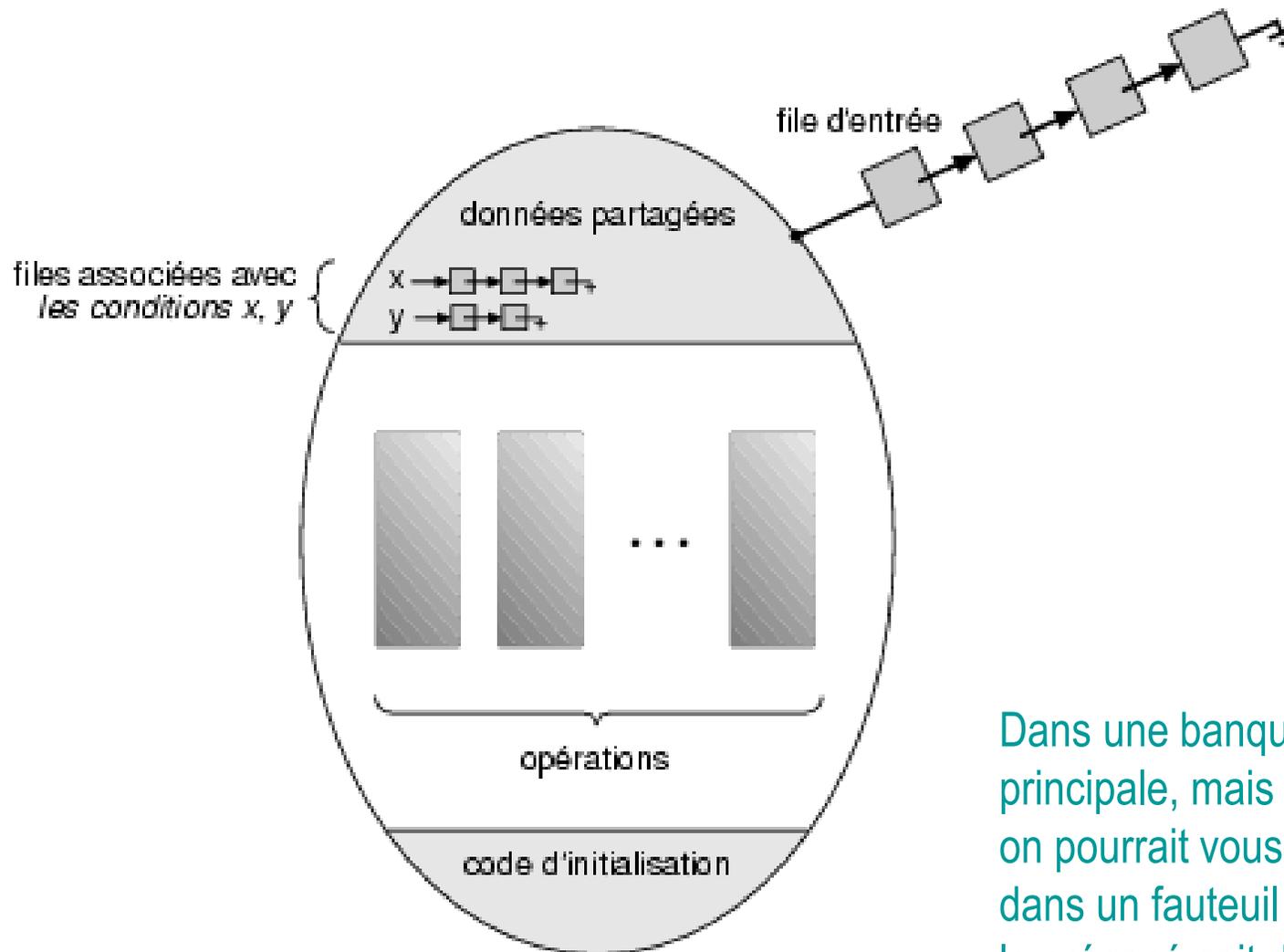
# Moniteur: Vue schématique simplifiée style Java



# Variables conditionnelles (n'existent pas en Java)

- **sont accessibles seulement dans le moniteur**
- **accessibles et modifiables seulement à l'aide de 2 fonctions:**
  - ◆ **x: wait** bloque l'exécution du thread exécutant sur la condition x
    - ☞ le thread pourra reprendre l'exécution seulement si un autre thread exécute x: signal)
  - ◆ **x: signal** reprend l'exécution d'un thread bloqué sur la condition x
    - ☞ S'il en existe plusieurs: en choisir un (file?)
    - ☞ S'il n'en existe pas: ne rien faire

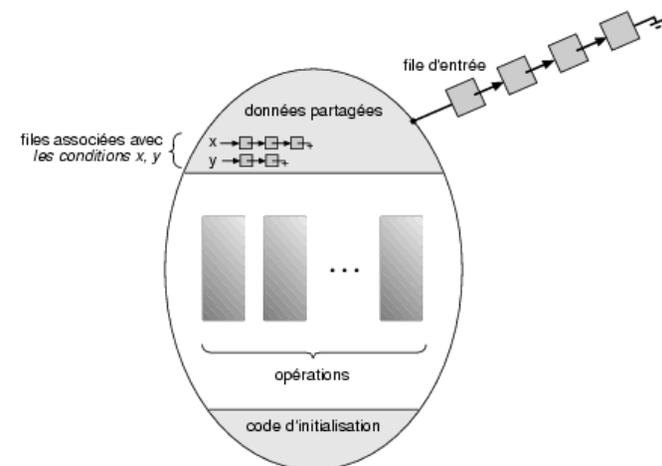
# Moniteur avec variables conditionnelles



Dans une banque, il y a une file principale, mais une fois entré on pourrait vous faire attendre dans un fauteuil jusqu'à ce que le préposé soit disponible

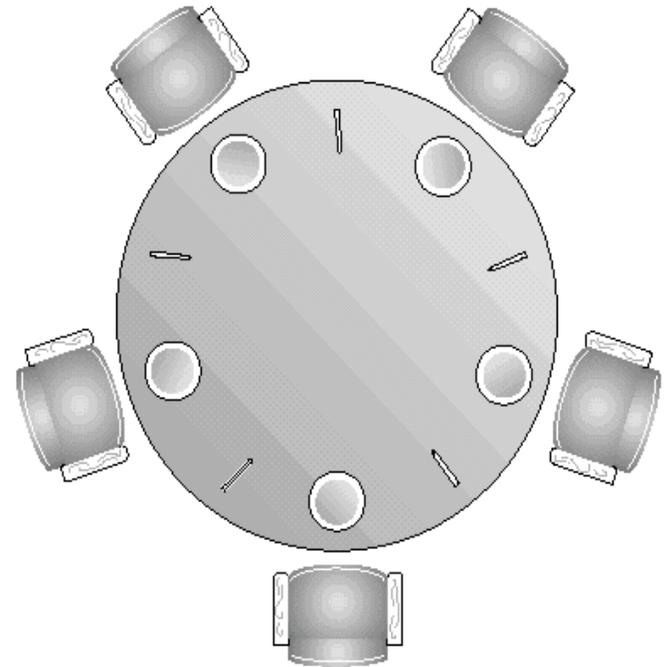
# Blocage dans les moniteurs

- threads attendent dans la file d'entrée ou dans une file de condition (ils n'exécutent pas)
- sur `x.wait`: le thread est placé dans la file de la condition (il n'exécute pas)
- `x.signal` amène dans le moniteur 1 thread de la file `x` (si `x` vide, aucun effet)



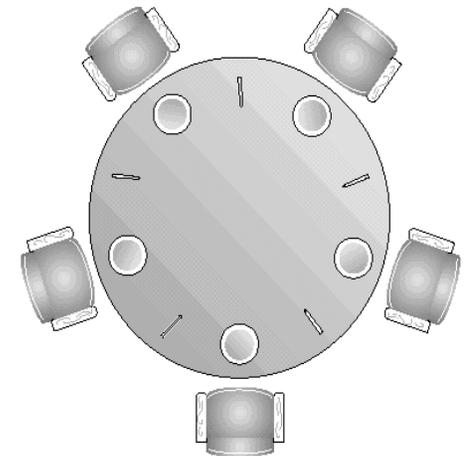
# Retour au problème des philosophes mangeant

- **5 philosophes qui mangent et pensent**
- **Pour manger il faut 2 baguettes, droite et gauche**
- **On en a seulement 5!**
- **Un problème classique de synchronisation**
- **Illustre la difficulté d'allouer ressources aux threads tout en évitant interblocage et famine**



# Philosophes mangeant structures de données

- **Chaque philos. a son propre **state** qui peut être (thinking, hungry, eating)**
  - ◆ philosophe  $i$  peut faire  $state[i] = \text{eating}$  ssi les voisins ne mangent pas
- **Chaque condition a sa propre condition **self****
  - ◆ le philosophe  $i$  peut attendre sur  $self [ i ]$  si veut manger, mais ne peut pas obtenir les 2 baguettes



# Chaque philosophe exécute à jamais:

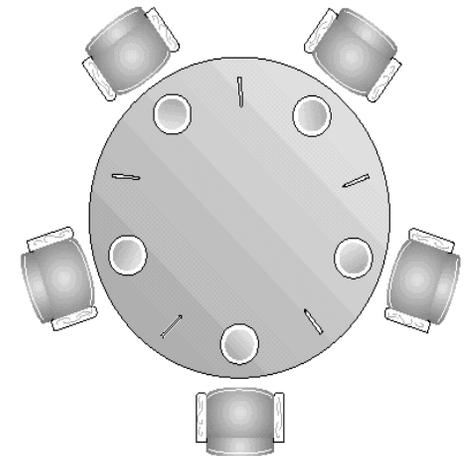
**repeat**

*pickup*

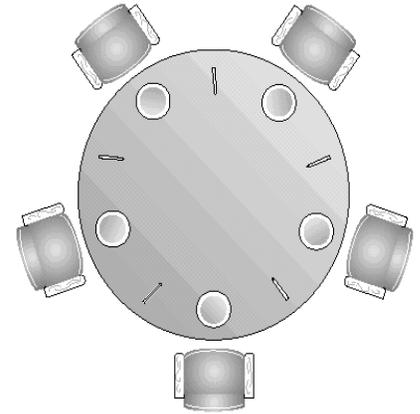
**eat**

*putdown*

**forever**



# Un philosophe mange



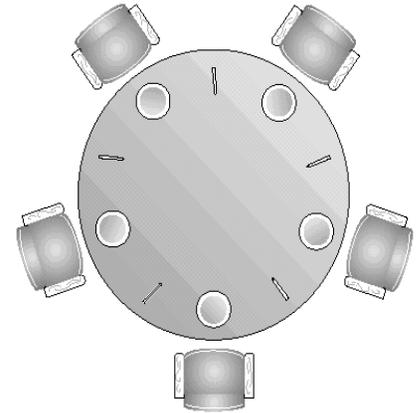
```
private test(int i) {  
    if ( (state[(i + 4) % 5] != EATING) &&  
        (state[i] == HUNGRY) &&  
        (state[(i + 1) % 5] != EATING) ) {  
        state[i] = EATING;  
        self[i].signal;  
    }  
}
```

Un philosophe mange si ses voisins ne mangent pas et s'il a faim.

Une fois mangé, il signale de façon qu'un autre pickup soit possible, si pickup s'était arrêté sur wait

Il peut aussi sortir sans avoir mangé si le test est faux

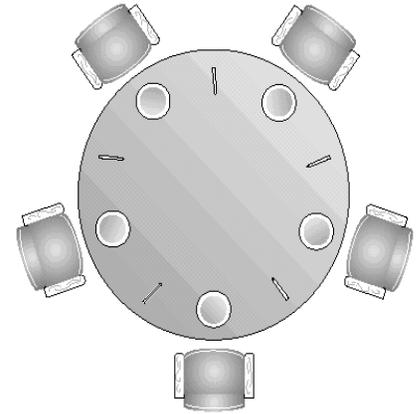
## Chercher de prendre les baguettes



```
public entry pickUp(int i) {  
    state[i] = HUNGRY;  
    test(i);  
    if (state[i] != EATING)  
        self[i].wait;  
}
```

Phil. cherche à manger en testant, s'il sort de test qu'il n'est pas mangeant il attend – un autre pickup n'est pas possible avant un `self[i]` signal

## Déposer les baguettes



```
public entry putDown(int i) {  
    state[i] = THINKING;  
    // tester les deux voisins  
    test((i + 4) % 5);  
    test((i + 1) % 5);  
}
```

Une fois fini de manger, un philosophe se préoccupe de faire manger ses voisins en les testant

## Relation entre moniteurs et autre mécanismes

- **Les moniteurs sont implantés utilisant les sémaphores ou les autres mécanismes déjà vus**
- **Il est aussi possible d`implanter les sémaphores en utilisant les moniteurs!**
  - ◆ Voir le texte

## Le problème de la SC en pratique...

- **Les systèmes réels rendent disponibles plusieurs mécanismes qui peuvent être utilisés pour obtenir la solution la plus efficace dans différentes situations**

## Concepts importants du Chapitre 6

- **Sections critiques: pourquoi**
- **Difficulté du problème de la synch sur SC**
  - ◆ Bonnes et mauvaises solutions
- **Accès atomique à la mémoire**
- **Solutions logiciel `pures`**
- **Solution matériel: test-and-set**
- **Solutions par appels du système:**
  - ◆ Sémaphores, moniteurs, fonctionnement
- **Problèmes typiques: tampon borné, lecteurs-écrivains, philosophes**