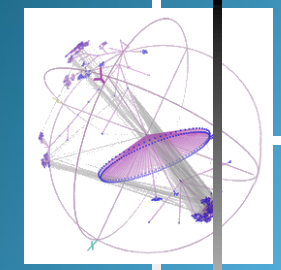
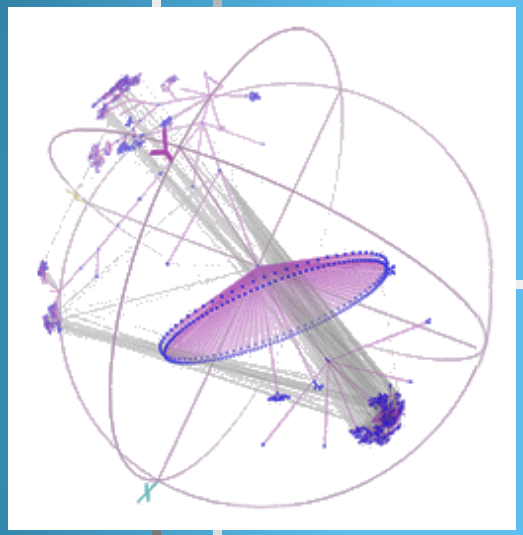
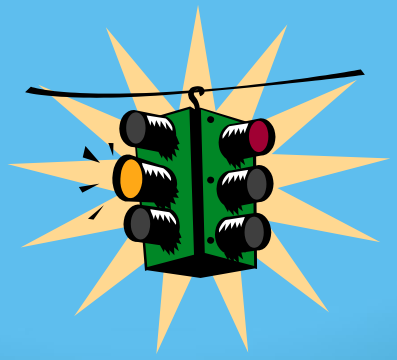
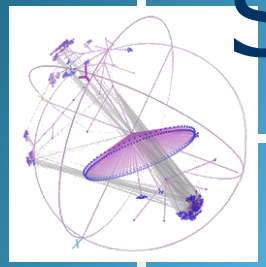




Synchronisation de Processus

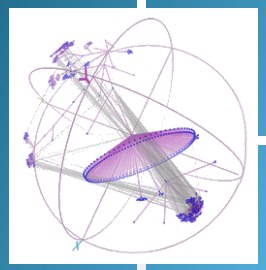
Chapitre 5





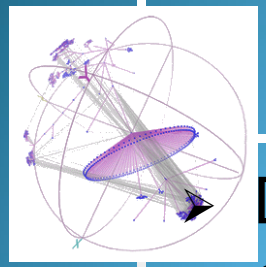
Synchronisation de Processus

1. **Conditions de Concurrence**
2. **Sections Critiques**
3. **Exclusion Mutuelle**
4. **Sommeil & Activation**
5. **Sémaphores**
6. **Mutex**



Concurrence

- Les processus concurrents doivent parfois partager données (fichiers ou mémoire commune) et ressources
  - On parle donc de tâches *coopératives*
- Si l'accès n'est pas contrôlé, le résultat de l'exécution du programme pourra dépendre de l'ordre d'entrelacement de l'exécution des instructions (*non-déterminisme*).
- Un programme pourra donner des résultats différents et parfois indésirables



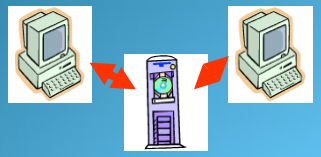
Un exemple

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

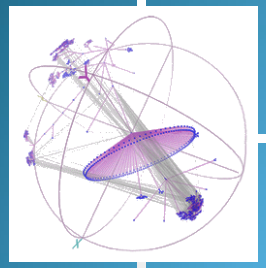
M. X demande une réservation d'avion

Base de données dit que fauteuil A est disponible

Fauteuil A est assigné à X et marqué occupé



Exemple d'exécution possible



P1

P2

Interruption
ou retard

M. Leblanc demande une réservation d'avion

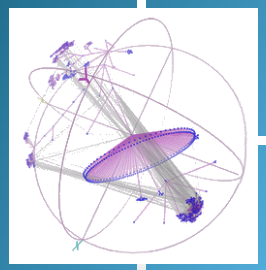
M. Guy demande une réservation d'avion

Base de données dit que fauteuil 30A est disponible

Base de données dit que fauteuil 30A est disponible

Fauteuil 30A est assigné à Leblanc et marqué occupé

Fauteuil 30A est assigné à Guy et marqué occupé



Un autre exemple

P1 interruption P2

```
b=a
```

```
b=a
b++
a=b
```

```
b++
a=b
```

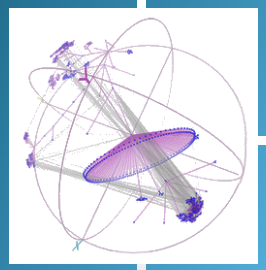
Deux opérations en parallèle sur une variable a partagée (b est privé à chaque processus)

Supposons que a soit 0 au début

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

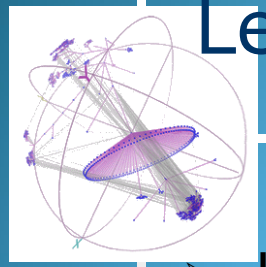
Il serait a=2 si les deux tâches sont exécutées l'une après l'autre

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)

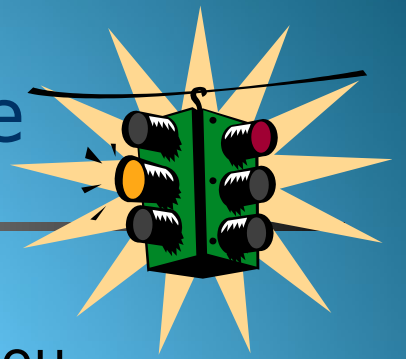


Section Critique

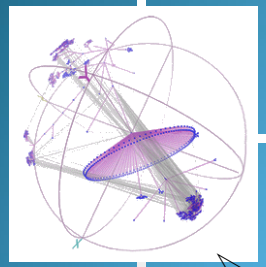
- Partie d'un programme dont l'exécution ne doit pas *entrelacer* avec autres programmes
- Une fois qu'un tâche y entre, il faut lui permettre de terminer cette section sans permettre à autres tâches de jouer sur les mêmes données



Le problème de la section critique



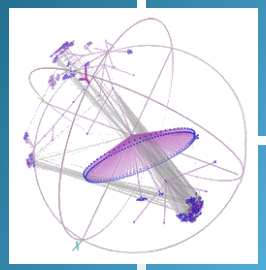
- Lorsqu'un processus manipule une donnée (ou ressource) partagée, nous disons qu'il se trouve dans une section critique (SC) (associée à cette donnée)
- Le problème de la section critique est de trouver un algorithme d`exclusion mutuelle de processus dans l'exécution de leur Section Critiques afin que 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 processus peut exécuter une SC pour une var 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, dorénavant nous faisons l'hypothèse qu'il n'y a qu'une seule SC dans un programme.



Structure du programme

- Chaque processus 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

M. X demande une réservation d'avion

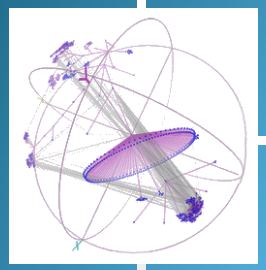
Section critique

Section d'entrée

Base de données dit que fauteuil A est disponible

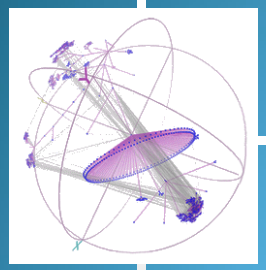
Fauteuil A est assigné à X et marqué occupé

Section de sortie



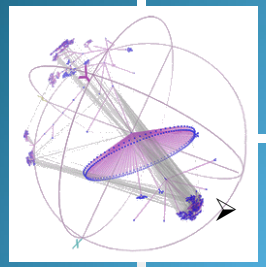
Critères nécessaires pour solutions valides

- Exclusion Mutuelle
  - À tout instant, au plus un processus peut être dans une section critique (SC) pour une variable donnée
- Non interférence:
  - Si un processus s'arrête dans sa section restante, ceci ne devrait pas affecter les autres processus
- Mais on fait l'hypothèse qu'un processus qui entre dans une section critique, en sortira.



Critères nécessaires pour solutions valides

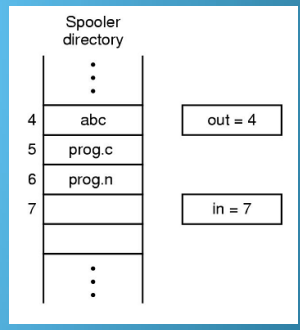
- Progrès:
  - absence d'interblocage
  - si un processus demande d`entrer dans une section critique à un moment où aucun autre processus en fait requête, il devrait être en mesure d'y entrer
- Absence de famine: aucun processus ne sera éternellement empêché d'atteindre sa Section Critique

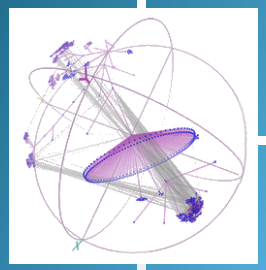


Conditions de Concurrence

Conditions de concurrence (*race conditions*): situation où 2 processus ou plus effectuent des lectures et des écritures conflictuelles.

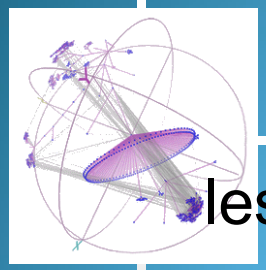
- Exemple du Spouler d'impression
  - Un processus qui veut imprimer un fichier, entre son nom dans un **répertoire de spoule**
  - Le processus **démon d'impression** regarde périodiquement s'il y a des fichiers à imprimer. Il a 2 variables:
    - **in**: pointe vers la prochaine entrée libre.
    - **out**: pointe vers le prochain fichier à imprimer
  - *in* = 7, *out* = 4
  - A et B deux processus qui veulent imprimer un fichier
  - A >> lire *in*, *next_free_slot* = 7
  - Interruption: la CPU bascule vers le processus B
  - B >> lire *in*, *next_free_slot* = 7, entrée7 = fichierB, *in* = 8
  - A >> entrée7 = fichierA, *in* = 8
  - Problème: le fichierB ne sera pas imprimé





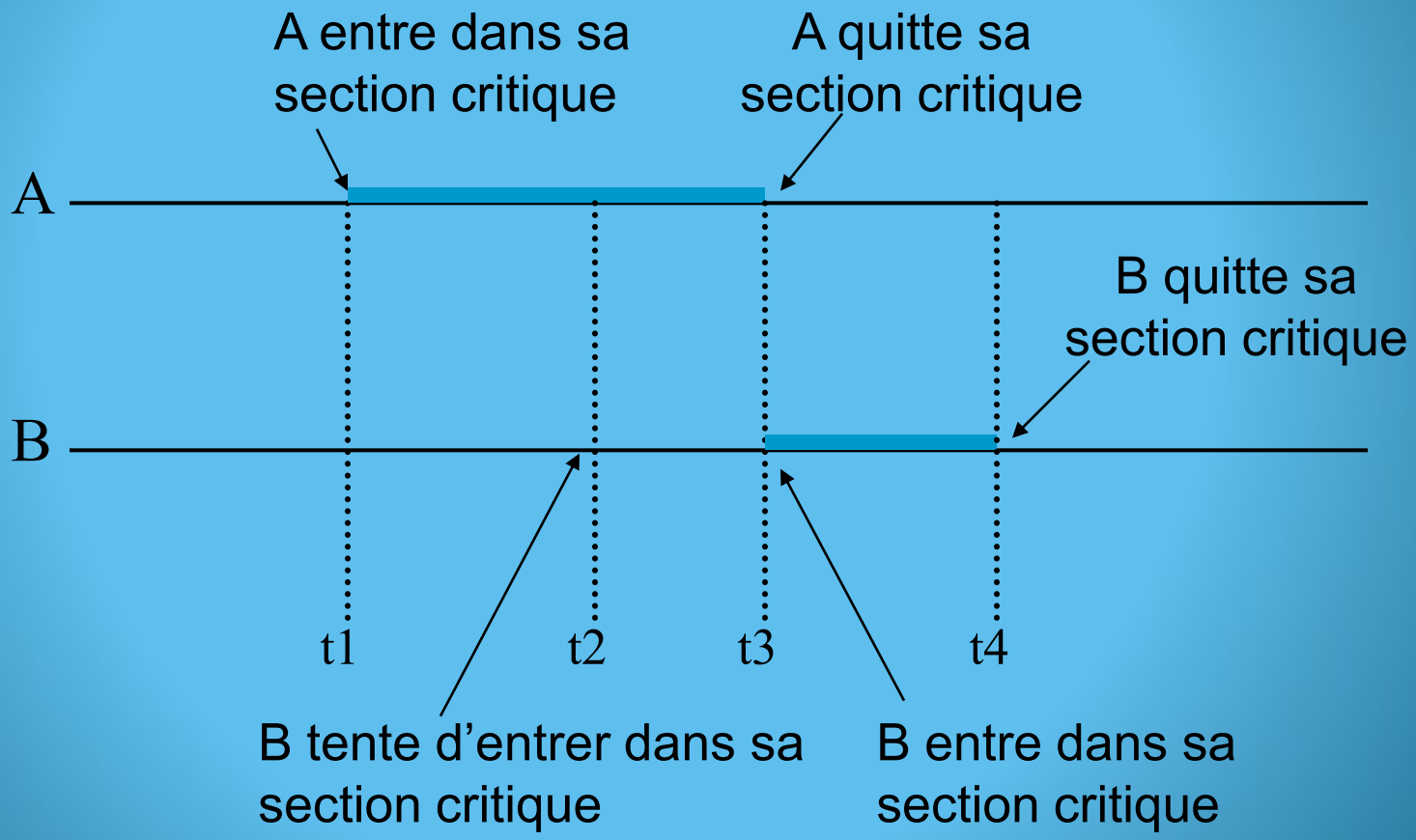
… Conditions de Concurrence

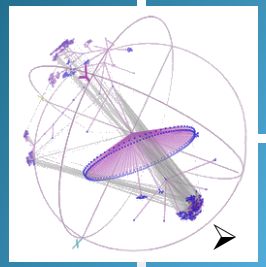
- Comment éviter les conditions de concurrence?
- Solution: Interdire que plusieurs processus lisent et écrivent des données partagées simultanément.
- Exclusion Mutuelle: permet d'assurer que si un processus utilise une variable ou fichier partagés, les autres processus seront exclus de la même activité



Les Sections Critiques

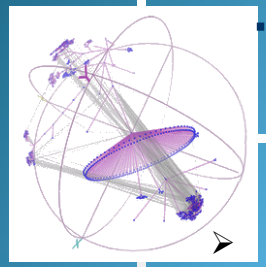
les Sections Critiques, méthode d'exclusion mutuelle





L'Exclusion Mutuelle avec Attente Active (*busy waiting*)

- Désactivation des interruptions
  - Après son entrée dans une SC, un processus désactive les interruptions, puis les réactive
  - Il empêche ainsi l'horloge d'envoyer des interruptions et le processeur de basculer
  - Il est imprudent de permettre à des processus user de désactiver les interruptions
- Variables de verrou *(lock)*
  - Avant d'entrer en SC, tester la valeur de verrou, si verrou = 0, verrou ← 1, entrer en SC
  - Défaillance: 2 processus peuvent entrer simultanément dans leurs sections critiques comme le spouler d'impression
- Alternance Stricte
  - la variable *turn* porte le numéro du processus dont c'est le tour d'entrer en SC. Chaque processus inspecte la valeur de la variable, avant d'entrer en SC.
  - Inconvénient: consomme bcp de temps CPU



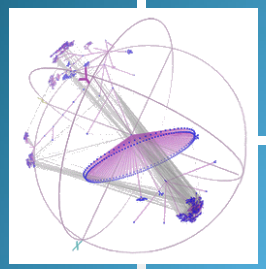
… Exclusion Mutuelle avec Attente Active (*busy waiting*)

… Alternance Stricte

```
while (TRUE) {
    while (turn != 0);
    critical_region();
    turn = 1;
    non_critical_region();
}
```

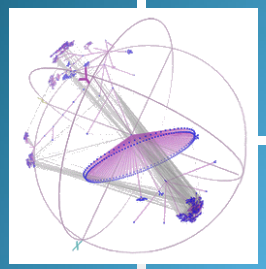
```
while (TRUE) {
    while (turn != 1);
    critical_region();
    turn = 0;
    non_critical_region();
}
```

- Les attentes actives sont performantes dans le cas où elles sont brèves. En effet, il y' a risque d'attente
  - P0 quitte la CS, turn = 1
  - P1 termine sa CS, turn = 0
  - Les 2 processus sont en section non critique
  - P0 exécute sa boucle, quitte la SC et turn = 1
  - Les 2 processus sont en section non critique
  - P0 quoiqu'il a terminé, il ne peut pas entrer en SC, il est bloqué



Une leçon à retenir…

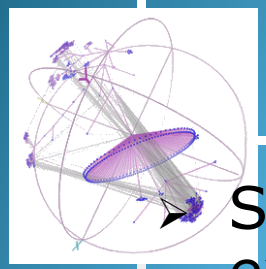
- À fin que des processus avec des variables partagées puissent réussir, il est nécessaire que tous les processus impliqués utilisent le même algorithme de coordination
  - Un protocole commun



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.

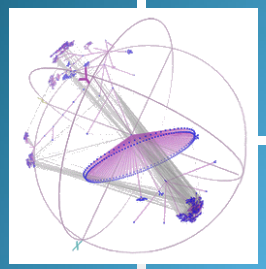
- Les processus qui requièrent l'entrée dans leur SC sont occupés à attendre (busy waiting); consommant ainsi du temps de processeur
  - Pour de longues sections critiques, il serait préférable de bloquer les processus qui doivent attendre...



Solutions matérielles: désactivation des interruptions

- Sur un uniprocesseur: exclusion mutuelle est préservée mais l'efficacité se détériore: lorsque dans SC il est impossible d'entrelacer l'exécution avec d'autres processus dans une SR
- Perte d'interruptions
- Sur un multiprocesseur: exclusion mutuelle n'est pas préservée
- Une solution qui n'est généralement pas acceptable

```
Process Pi:
repeat
   inhiber interrupt
    section critique
   rétablir interrupt
    section restante
forever
```



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

- Les solutions vues jusqu'à présent sont difficiles à programmer et conduisent à du mauvais code.
- On voudrait aussi qu`il soit plus facile d'éviter des erreurs communes, comme interblocages, famine, etc.
  - Besoin d'instruction à plus haut niveau
- Les méthodes que nous verrons dorénavant utilisent des instructions puissantes, qui sont implantées par des appels au SE (system calls)

Sémaphores

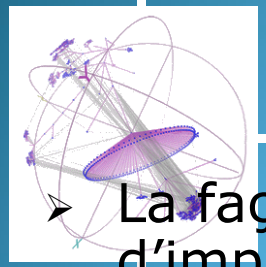
Un sémaphore S est un entier qui, sauf pour l'Initialisation, est accessible seulement par ces 2 opérations atomiques et mutuellement exclusives:

- wait(S)
- signal(S)

- Il est partagé entre tous les procs qui s`intéressent à la même section critique
- Les sémaphores seront présentés en deux étapes:
  - sémaphores qui sont occupés à attendre (busy waiting)
  - sémaphores qui utilisent des files d ’attente
- On fait distinction aussi entre sémaphores compteurs et sémaphores binaires, mais ce derniers sont moins puissants.

Sémaphores occupés à attendre

(busy waiting)



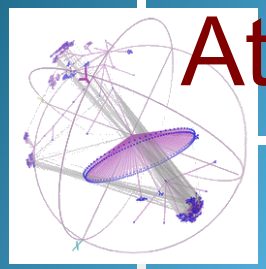
- La façon la plus simple d'implanter les sémaphores.
- Utiles pour des situations où l'attente est brève, ou il y a beaucoup d'UCTs
- S est un entier initialisé à une valeur positive, de façon que un premier processus puisse entrer dans la SC
- Quand S>0, jusqu'à n processus peuvent entrer
- Quand S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

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

Attend si no. de processus qui peuvent entrer = 0 ou négatif

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

Augmente de 1 le no des processus qui peuvent entrer



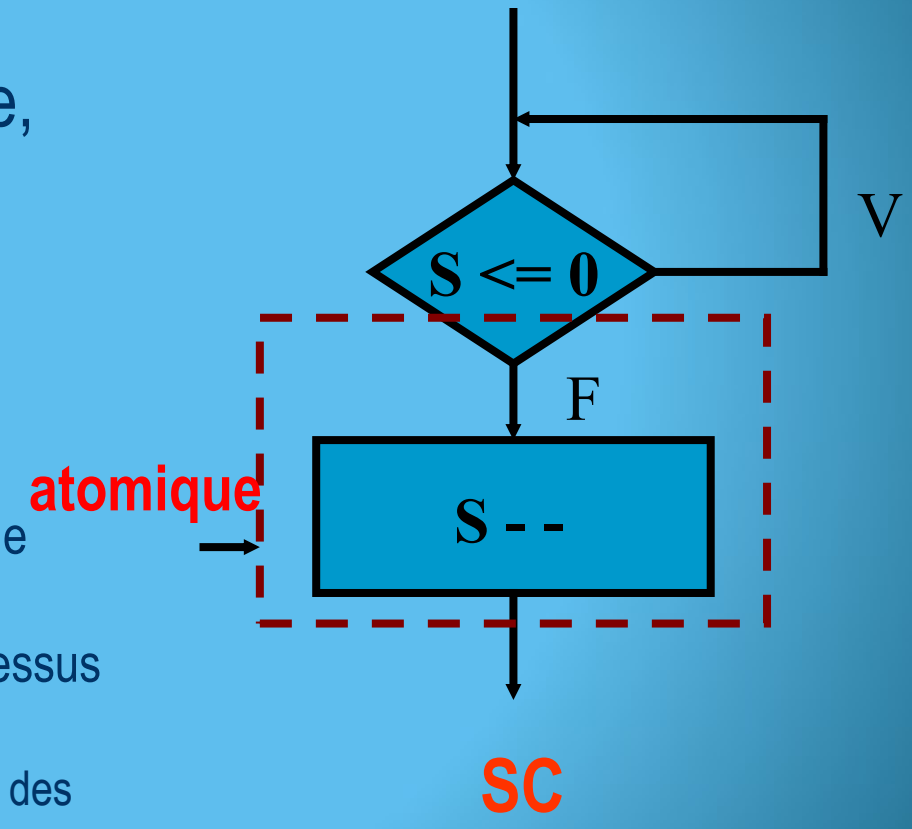
Atomicité

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

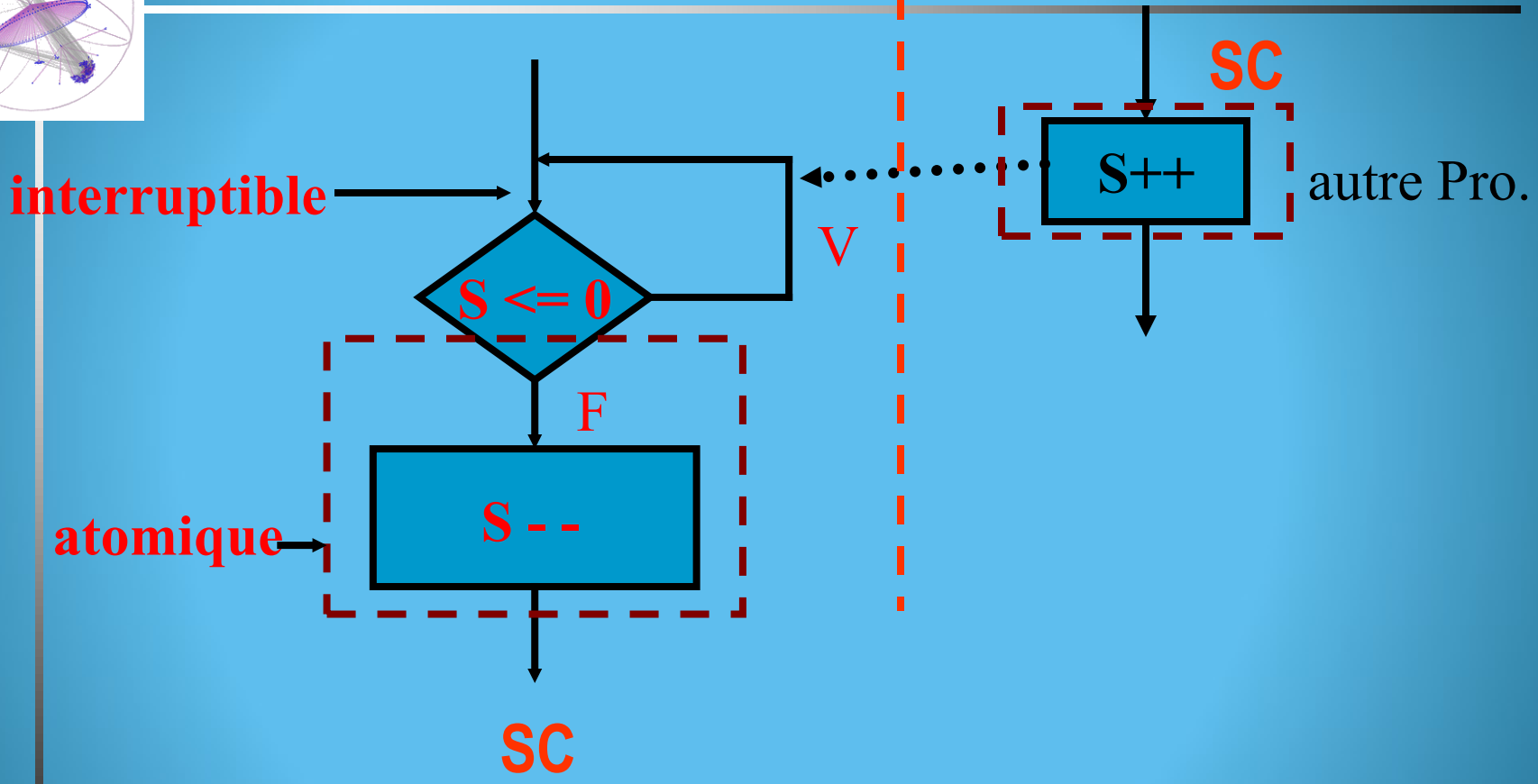
Signal est atomique.

Rappel: les sections atomiques ne peuvent pas être exécutées simultanément par différent processus

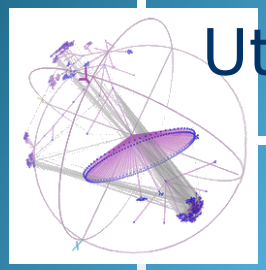
(ceci peut être obtenu en utilisant un des mécanismes précédents)



Atomicité et interruptibilité



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



Utilisation des sémaphores pour sections critiques

- Pour n processus
- Initialiser S à 1
- Alors 1 seul processus peut être dans sa SC
- Pour permettre à k processus d'exécuter SC, initialiser S à k

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

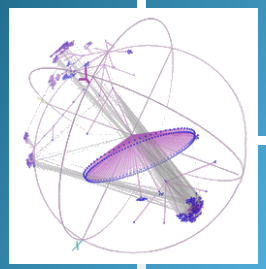
Initialise S à >=1

```
processus T1:
 repeat
 wait(S);
     SC
 signal(S);
     SR
forever
```

```
processus T2:
repeat
  wait(S);
   SC
  signal(S);
   SR
forever
```

Semaphores: vue globale

Peut être facilement généralisé à plus. processus



Utilisation des sémaphores pour synchronisation de processus

- On a 2 processus : T1 et T2
- Énoncé S1 dans T1 doit être exécuté avant énoncé S2 dans T2
- Définissons un sémaphore S
- Initialiser S à 0

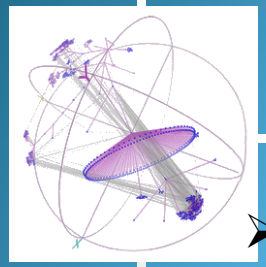
- Synchronisation correcte lorsque T1 contient:

```
S1;
signal(S);
```

- et que T2 contient:

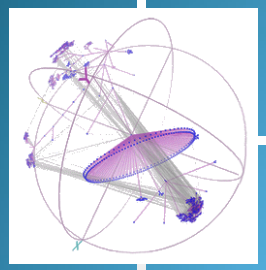
```
wait(S);
S2;
```

Interblocage et famine avec les sémaphores



- Famine: un processus peut ne jamais arriver à s'exécuter car il ne teste jamais le sémaphore au bon moment
- Interblocage: Supposons S et Q initialisés à 1

T0	T1
wait(S)	
	wait(Q)
wait(Q)	wait(S)

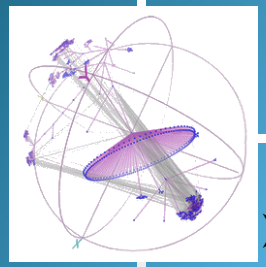


Sémaphores: observations

- Quand S >= 0:

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

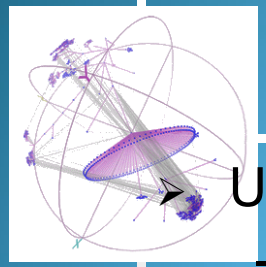
  - Le nombre de processus qui peuvent exécuter wait(S) sans devenir bloqués= S
    - S processus peuvent entrer dans la SC
    - noter puissance par rapport à mécanismes déjà vus
    - dans les solutions où S peut être >1il faudra avoir un 2ème sém. pour les faire entrer un à la fois (excl. mutuelle)
- Quand S devient > 1, le processus qui entre le premier dans la SC est le premier à tester S (choix aléatoire)
  - ceci ne sera plus vrai dans la solution suivante
- Quand S < 0: le nombre de processus qui attendent sur S est = |S|



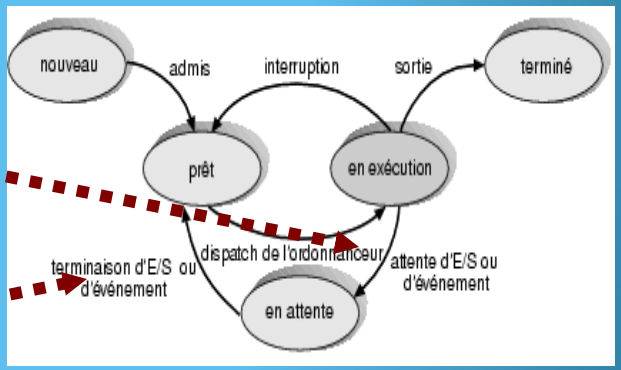
Comment éviter l'attente occupée et le choix aléatoire dans les sémaphores

- Quand un processus doit attendre qu'un sémaphore devienne plus grand que 0, il est mis dans une file d'attente de processus qui attendent sur le même sémaphore.
- Les files peuvent être PAPS (FIFO), avec priorités, etc. Le SE contrôle l`ordre dans lequel les processus entrent dans leur SC.
- *wait* et *signal* sont des appels au SE 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.

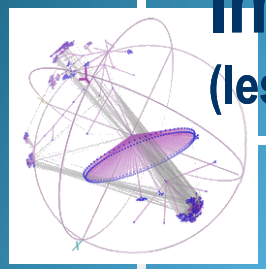
Sémaphores sans attente occupée



- Un sémaphore S devient une structure de données:
  - Une valeur
  - Une liste d'attente L
- Un processus devant attendre un sémaphore S, est bloqué et ajouté la file d'attente S.L du sémaphore (v. état bloqué = attente chap 4).



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



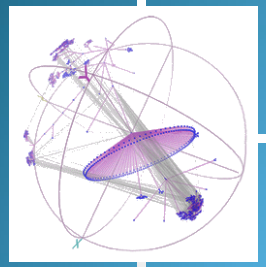
Implementation

(les boîtes réprésentent des séquences non-interruptibles)

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

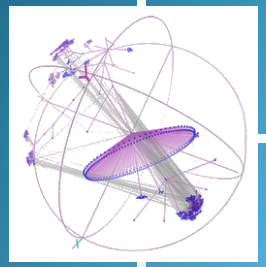
```
signal(S): S.value ++;
           if S.value ≤ 0 {         // des processus attendent
                remove a process P from S.L;
                wakeup(P)           // processus choisi devient prêt
                }
```

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



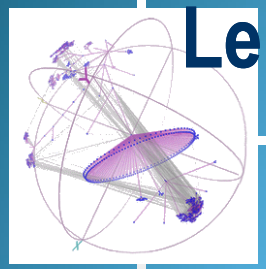
Wait et signal contiennent elles mêmes des SC!

- Les opérations *wait* et *signal* doivent être exécutées atomiquement
- Dans un système avec 1 seule UCT, ceci peut être obtenu en inhibant les interruptions quand un processus exécute ces opérations
- L'attente occupée dans ce cas ne sera pas trop onéreuse car *wait* et *signal* sont brefs



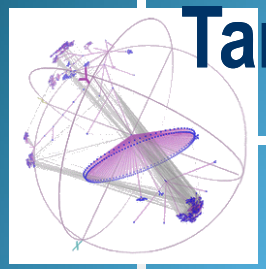
Problèmes classiques de synchronisation

- Tampon borné (producteur-consommateur)
- Écrivains - Lecteurs
- Les philosophes mangeant

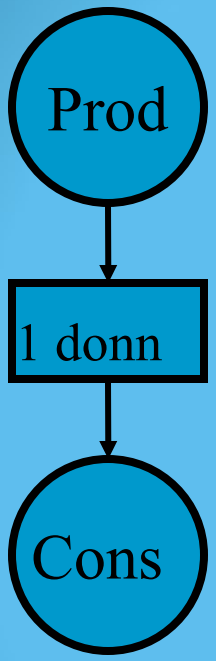


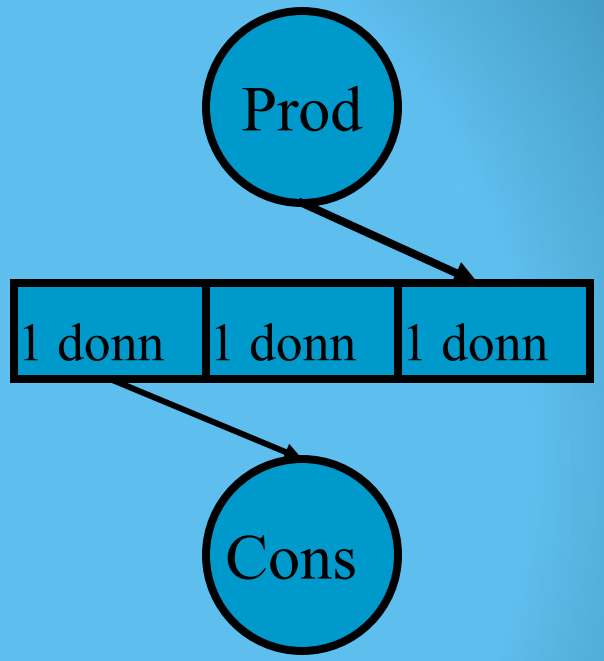
Le pb du producteur - consommateur

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



Tampons de communication

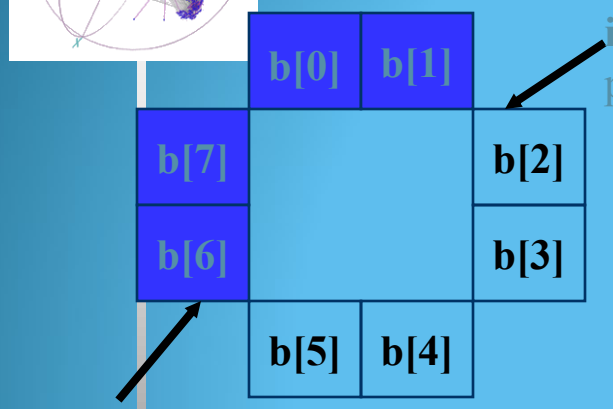




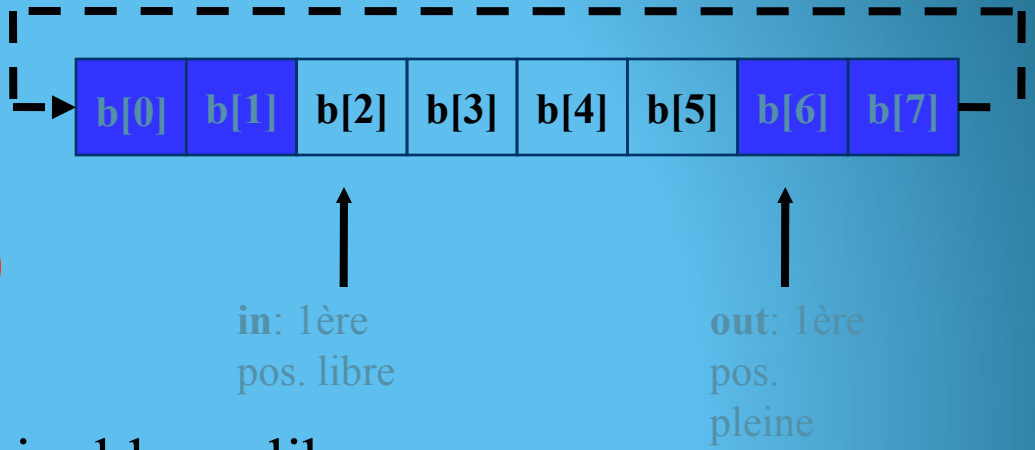
Si le tampon est de longueur 1, le producteur et consommateur doivent forcement aller à la même vitesse

Des tampons de longueur plus grandes 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

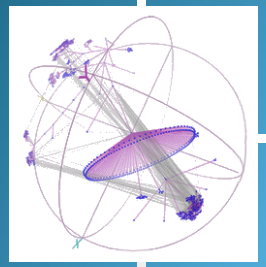


ou



bleu: plein, blanc: libre

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

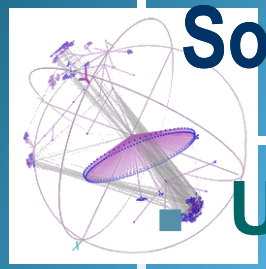


Problème de synchronisation entre processus pour le tampon borné

- Étant donné que le producteur et le consommateur sont des processus indépendants, des problèmes pourraient se produire en permettant accès simultané au tampon
- Les sémaphores peuvent résoudre ce problème

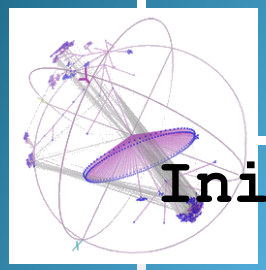
Sémaphores: rappel.

- **Soit S un sémaphore sur une SC**
  - il est associé à une file d ’attente
  - S positif: S processus peuvent entrer dans SC
  - S zéro: aucun processus ne peut entrer, aucun processus en attente
  - S négatif: |S| processus dans file d ’attente
- **Wait(S): S - -**
  - si après S >= 0, processus peut entrer dans SC
  - si S < 0, processus est mis dans file d ’attente
- **Signal(S): S++**
  - si après S<= 0, il y avait des processus en attente, et un processus est réveillé
- **Indivisibilité = atomicité de ces ops**



Solution avec sémaphores

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



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

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

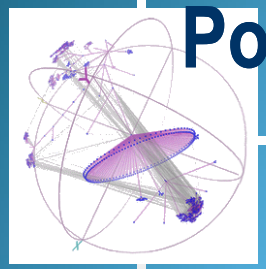
```
append(v):
  b[in]=v;
  In ++ mod k;
```

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

```
Producer:
repeat
   produce v;
   wait(E);
   wait(S);
   append(v);
   signal(S);
   signal(N);
forever
```

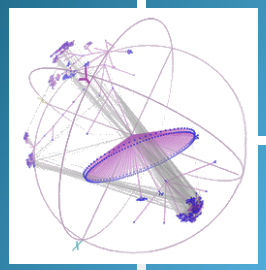
```
Consumer:
repeat
   wait(N);
   wait(S);
   w=take();
   signal(S);
   signal(E);
   consume(w);
forever
```

- Sections critiques



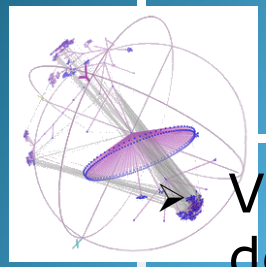
Points importants à étudier

- **dégâts possibles en interchangeant les instructions sur les sémaphores**
  - ou en changeant leur initialisation
- **Généralisation au cas de plus. Producteurs et consommateur**



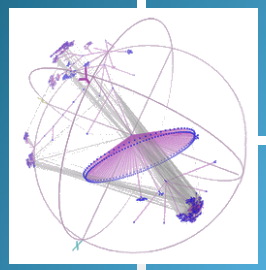
Problème des lecteurs - rédacteurs

- Plusieurs processus 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 processus 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 processus 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



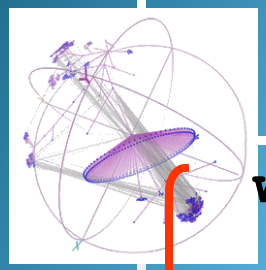
Les données et les rédacteurs

Données: deux sémaphores et une variable

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

Rédacteur

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



Les lecteurs

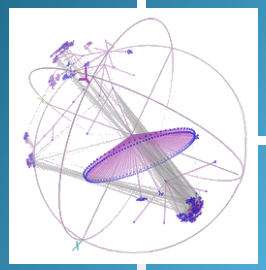
```
wait(mutex);
        readcount ++ ;
        if readcount == 1 then wait(wrt);
signal(mutex);

//SC: lecture

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

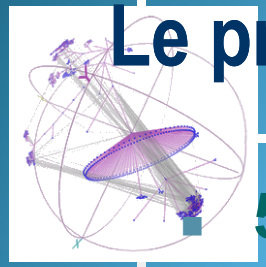
Le premier lecteur d 'un groupe pourrait devoir attendre sur wrt, il doit aussi bloquer les rédacteurs. Quand il sera entré, les suivants pourront entrer librement

Le dernier lecteur sortant doit permettre l`accès aux rédacteurs



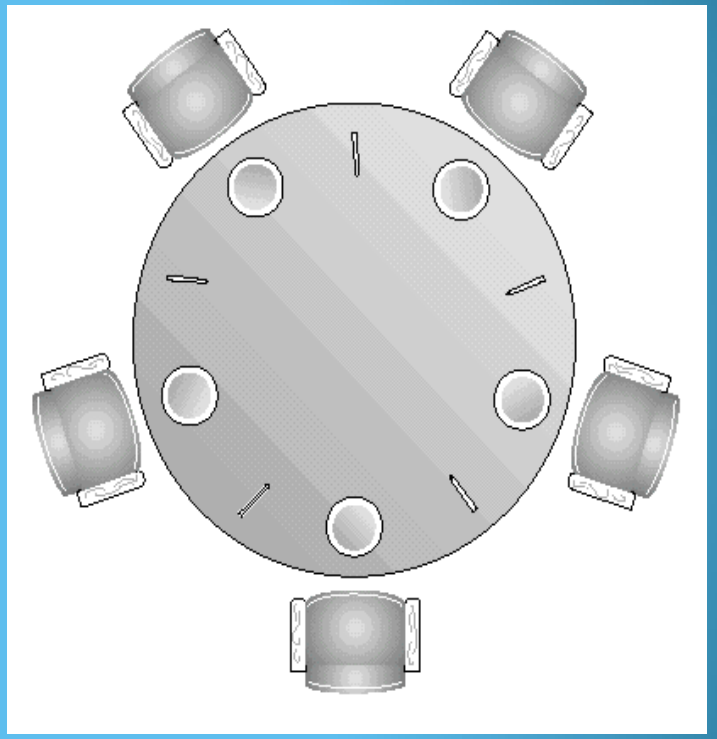
Observations

- 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

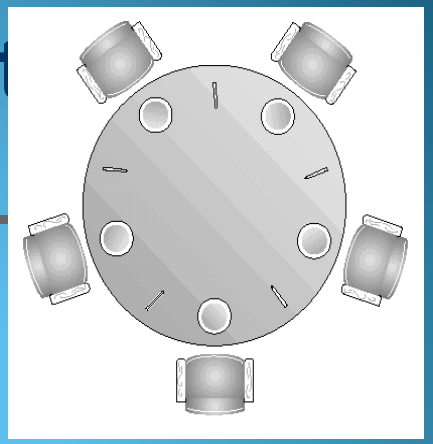


Le problème des philosophes mangeant

- 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 processus tout en évitant interblocage et famine



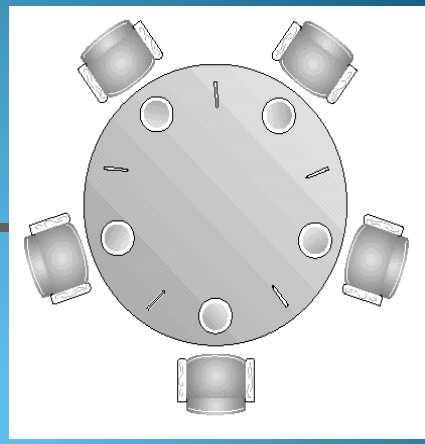
Le problème des philosophes mangeant



- **Un processus 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])`

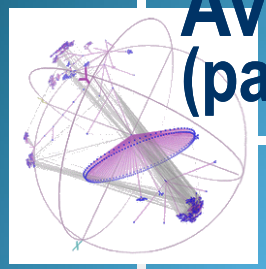
```
processus Pi:
repeat
 think;
 wait(fork[i]);
 wait(fork[i+1 mod 5]);
 eat;
 signal(fork[i+1 mod 5]);
 signal(fork[i]);
forever
```

Le problème des philosophes mangeant



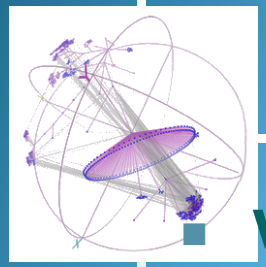
- **Une solution: admettre seulement 4 philosophes à la fois qui peuvent tenter de manger**
- **Il y aura toujours au moins 1 philosophe qui pourra manger**
  - même si tous prennent 1 fourchette
- **Ajout d'un sémaphore T qui limite à 4 le nombre de philosophes "assis à la table"**
  - initial. de T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

```
processus 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
```



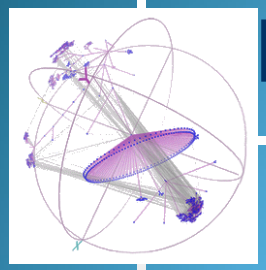
Avantage des sémaphores
(par rapport aux solutions précédentes)

- Une seule variable partagée par section critique
- deux seules opérations: wait, signal
- contrôle plus localisé (que avec les précédents)
- extension facile au cas de plus. processus
- possibilité de faire entrer plus. processus 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 processus, mais ils doivent se correspondre**
  - V. programme du tampon borné
- **Utilisation doit être correcte dans tous les processus**
- **Un seul "mauvais" processus peut faire échouer toute une collection de processus (p.ex. oublie de faire signal)**
- **Considérez le cas d`un processus qui a des waits et signals dans des boucles et des tests...**



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