

Les sémaphores de Dijkstra

Définition

- ❑ Le concept de sémaphore a été introduit en 1965 par Dijkstra.
- ❑ C'est un outil offert par le système d'exploitation pour résoudre les problèmes de **synchronisation** en l'occurrence le problème de l'**exclusion mutuelle**.
- ❑ On peut voir un sémaphore comme un distributeur de jetons.

Définition

□ Un sémaphore est une structure de données composée :

- D'une variable entière: **k** (nombre de jetons)
 - Représente le nombre de processus pouvant utiliser la (ou les) ressource(s) simultanément ,
- D'une file d'attente: **F**.
 - Utilisée pour faire attendre les processus non autorisés à utiliser la (ou les) ressource(s)
 - La file est généralement gérée en FIFO
 - A la création d'un sémaphore sa file doit être vide.

Opérations sur les sémaphores: Une variante possible

- On ne peut manipuler un sémaphore qu'à travers trois fonctions (**atomiques**):

a. La fonction init:

- Sert à initialiser le sémaphore en spécifiant la valeur de k et la file F.

```
init (semaphore sem, int val)
  début
    sem.k:=val ;
    sem.F:=null ;
  fin.
```

Opérations sur les sémaphores: Une variante possible

b. La fonction « P » ou « wait » (Proberen en hollandais: Essayer):

- C'est une fonction bloquante pour le processus qui l'appelle si le nombre k est inférieur à 0 (c'est-à-dire il n'y a plus de jetons)

```
P(semaphore sem)
```

```
  début
```

```
  sem.k := sem.k - 1;
```

```
  si sem.k < 0 alors
```

```
    Ranger le contexte du processus dans la file F
```

```
    Mettre le processus dans l'état bloqué
```

```
  fin si
```

```
  fin.
```

Opérations sur les sémaphores: Une variante possible

c. La fonction « V » ou « signal » (Verhogen: incrémenter):

- Elle permet de récupérer le jeton au sémaphore, de plus s'il y a au moins un processus bloqué dans la file d'attente F, ce dernier est réveillé.

V(semaphore sem)

début

sem.k := sem.k + 1;

si sem.k ≤ 0 alors

Retirer le contexte d'un processus de la file F

Mettre ce processus dans l'état prêt (réveiller le processus)

finsi

fin.

Implémentation des sémaphores

- ❑ Les primitives P et V doivent être exécutées en **exclusion mutuelle** car elles partagent :
 - La variable qui contient la valeur du sémaphore et
 - La file d'attente du sémaphore.
- ❑ Pour garantir l'exclusion mutuelle, on utilise 2 techniques:
 - ❑ Masquage/démasquage des interruptions
 - ❑ L'instruction TAS ou LOCK XCHG

Propriétés des sémaphores

- ❑ Un sémaphore **ne** peut être initialisé à une valeur **inférieure à 0**,
 - Mais sa valeur **peut devenir négative** après un certain nombre d'opérations P.
- ❑ La valeur initiale d'un sémaphore donne le nombre de processus pouvant exécuter simultanément une partie du code:

Propriétés des sémaphores

□ Si une ressource est **partageable** avec **k points d'accès**, la valeur initiale du sémaphore protégeant cette ressource est égale à **k**.

□ **Exemple:**

- Si un fichier peut être partagé simultanément, en lecture, entre **n** processus

⇒ La valeur initiale du sémaphore protégeant ce fichier est alors égale **n**.

Propriétés des sémaphores

- A tout instant la valeur $s.k$ d'un sémaphore s est donnée par la relation suivante : $s.k = s.k_0 - nP + nV$;
 - $s.k_0$: valeur initiale du sémaphore s ,
 - nP : nombre d'exécutions de P sur le sémaphore s ;
 - nV : nombre d'exécutions de V sur le sémaphore s ;
- Si la valeur d'un sémaphore est inférieure à zéro, sa valeur absolue est égale au nombre de processus bloqués dans sa file $s.F$.
 - $s.k < 0 \rightarrow |s.k|$ processus bloqués dans la file F .

Réalisation de l'exclusion mutuelle avec les sémaphores

- ❑ Un sémaphore d'exclusion mutuelle est toujours initialisé à 1 (un).
- ❑ Soit mutex le sémaphore d'exclusion mutuelle initialisé à 1.

```
mutex: Sémaphore;
```

```
init (mutex,1); //initialisation avec 1
```

```
Processus i (i=1, ...N)
```

```
P(mutex);      //protocole d'entrée en SC
```

```
<Section critique>
```

```
V(mutex);      //protocole de sortie de la SC
```

Sections critiques imbriquées

❑ Soit à gérer **deux** ressources critiques R1 et R2:

- Pour cela il faut utiliser deux sémaphores SR1 et SR2

Solution proposée:

SR1, SR2: sémaphore; Init(SR1, 1); Init(SR2, 1);	
Processus A	Processus B
P(SR1) P(SR2) <Utilisation de R1 et R2> V(SR2) V(SR1)	P(SR2) P(SR1) <Utilisation de R1 et R2> V(SR1) V(SR2)

Sections critiques imbriquées

- ❑ La solution précédente peut mener à une situation d'*interblocage*.
- ❑ Pour éviter cette situation:
 - Les processus utilisant les mêmes ressources demandent celles-ci dans le *même ordre*

SR1, SR2: sémaphore; Init(SR1, 1); Init(SR2, 1);	
Processus A	Processus B
P(SR1) P(SR2) <Utilisation de R1 et R2> V(SR2) V(SR1)	P(SR1) P(SR2) <Utilisation de R1 et R2> V(SR1) V(SR2)

Synchronisation des processus

Introduction: Exemple d'illustration

sem: sémaphore; Init(sem, 0);	
Processus A	Processus B
Travail A	P(sem);
V (sem);	Travail B

- ❑ Quels sont les résultats possibles d'exécution des deux processus?
- ❑ **Principe de synchronisation** : Un processus doit attendre un autre pour continuer (ou commencer) son exécution

Définition

- ❑ La synchronisation consiste à implémenter des mécanismes assurant le respect des contraintes liées à la progression des processus.
- ❑ Un mécanisme de synchronisation doit permet à un processus actif:
 - De se bloquer (attendre une ressource ou un signal d'un autre processus);
 - D'activer un autre processus en lui envoyant un signal.

Synchronisation directe et indirecte

- ❑ **Synchronisation directe:** Le processus envoie un signal à un autre processus en le désignant par son identificateur (ou par son nom):
 - **Exemple:** les signaux: SIGSTOP, SIGCONT, SIGSHLD,...
- ❑ **Synchronisation indirecte:** Le processus envoie un signal(sans désigner un processus)
 - Le processus ne connaît pas l'identité du ou des processus réveillés
 - **Exemple:** Opération V sur un sémaphore.

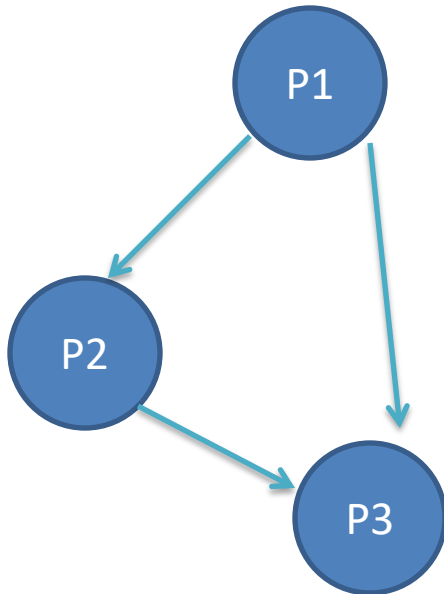
Spécification de la synchronisation

- ❑ La spécification d'un problème de synchronisation consiste à:
 - Définir pour chaque processus ses **points de synchronisation**.
 - Associer à chaque point de synchronisation une **condition de franchissement**.
- ❑ Les conditions sont exprimées au moyen des variables d'état du système:
 - Etat d'une ressource : libre ou occupée;
 - Nombre de ressources libres;
 - Nombre de ressources occupées;
 - Nombre de processus en attente d'une ressource donnée..

Exercice

- ❑ Soient trois processus P1, P2, P3 tel que :
 - P1 calcule la somme $S1 = A + B$,
 - P2 calcule $S2 = 2 * S1 + 4$
 - P3 calcule $S3 = S1 + S2$;
- ❑ Représenter par un graphe de précédence les relations entre les 3 processus.
- ❑ Ecrire les codes des trois processus en utilisant les sémaphores

Solution



Sémaphores: 1vers2, 1vers3, 2vers3
init(1vers2, 0), init(1vers3, 0), init(2vers3, 0),

P1:

S1= A+B;

V(1vers2);

V(1vers3);

P2:

P(1vers2);

S2= 2*S1+4;

V(2vers3);

P3:

P(1vers3)

P(2vers3)

S3=S1+S2

Les problèmes types

- ❑ Les problèmes types peuvent être regroupés en trois classes:
 - Exclusion Mutuelle
 - Partage de ressources communes par plusieurs processus
 - Allocateur de ressources banalisées,
 - Modèle des lecteurs-rédacteurs.
 - Communication entre processus
 - Rendez-vous,
 - Modèle des producteurs-consommateurs.

1. Allocateur de ressources

a) Allocation d'une (seule) ressource critique (Ex: imprimante)

❑ m processus se partagent une **seule** imprimante.

- L'imprimante est une ressource critique : elle doit être utilisée en exclusion mutuelle;

❑ Schéma général d'un processus i :

Processus i

Début

Demander(imprimante);

<Utiliser l'imprimante>

Restituer(imprimante);

Fin;

Allocateur de ressources

Solution avec les sémaphores:

❑ Soit **mutex** le sémaphore d'exclusion mutuelle protégeant cette imprimante :

```
mutex: sémaphore;  
init (mutex, 1);
```

Processus i

Début

```
P(mutex);
```

<Utiliser l'imprimante>

```
V(mutex);
```

Fin;

Allocateur de ressources

b) Allocation de plusieurs ressources simultanément:

- Exemple: **m** processus se partagent **n** imprimantes
- Schéma général d'un processus i :

Processus i

Début

Demander(imprimante);

<Utiliser une imprimante parmi n >

Restituer(imprimante);

Fin;

Allocateur de ressources

- ❑ La ressource « imprimantes » est une ressource **banalisée** composée de **n exemplaires**.
 - Chaque unité ou exemplaire est une ressource critique.
 - **n** processus peuvent utiliser ces **n** imprimantes en même temps.
- ❑ Condition de franchissement de points de synchronisation:
 - Allocation d'une imprimante si $\text{nbre_imp_libre} > 0$

Allocateur de ressources

❑ Solution à l'aide des sémaphores:

- On utilise un sémaphore représentant le nombre de ressources libres (imprimantes libres) : `nblibres`.

```
nblibres: sémaphore;
```

```
init (nblibres, n);
```

```
Processus i
```

```
Début
```

```
P(nblibres); //Demander(imprimante)
```

```
<Utiliser une imprimante>
```

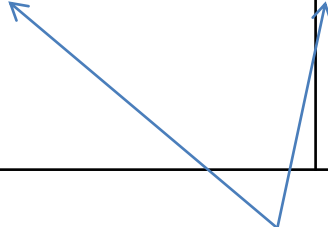
```
V(nblibres); //Restituer(imprimante)
```

```
Fin;
```

2. Rendez-vous de processus

a) Rendez-vous de deux processus : P1 et P2

Processus P1	Processus P2
Début	Début
...	...
Signaler son arrivée à P2;	Signaler son arrivée à P1;
Attendre P2;	Attendre P1;
...	...
Fin	Fin



Points de rendez-vous

Rendez-vous de processus

❑ Solution avec les sémaphores

- On utilise deux sémaphores de synchronisation initialisés à 0.

s1, s2: sémaphore; init(s1, 0); init (s2,0);	
Processus P1	Processus P2
Début	Début
...	...
V(s2);	V(s1);
P(s1);	P(s2);
...	...
Fin	Fin

Rendez-vous de processus

b) Rendez-vous des trois processus : P1, P2 et P3

Processus P1	Processus P2	Processus P3
Début	Début	Début
...
Signaler son arrivée à P2 et P3;	Signaler son arrivée à P1 et P3;	Signaler son arrivée à P1 et P2;
Attendre P2 et P3;	Attendre P1 et P3;	Attendre P1 et P2;
...
Fin	Fin	Fin

Rendez-vous de processus

❑ Solution avec les sémaphores

- On utilise trois sémaphores de synchronisation initialisés à 0

s1, s2, s3: sémaphore; init(s1, 0); init (s2,0); init(s3,0);		
Processus P1	Processus P2	Processus P3
Début	Début	Début
...
V(s2); V(s3);	V(s1); V(s3);	V(s1); V(s2);
P(s1); P(s1);	P(s2); P(s2);	P(s3); P(s3);
...
Fin	Fin	Fin

Rendez-vous de processus

c) Rendez-vous de n processus : P_0 à P_{n-1}

□ Solution avec une variable partagée (par les n processus) et deux sémaphores :

- La variable c'est un **compteur** qui permet de compter les **arrivées** des processus (nombre de processus arrivés) .
- Deux sémaphores:
 - Le compteur est partagé donc il doit être protégé par un sémaphore d'exclusion mutuelle initialisé à un.
 - Un sémaphore de synchronisation permettant de mettre en attente les processus; ce sémaphore est initialisé à zéro

Solution1: Le dernier processus réveil les (N-1) processus précédents

```
Init(mutex,1) ; Init(s,0) ;
```

```
Int cpt=0 ;
```

Processus RDV

Début

```
P(mutex) ;
```

```
cpt++ ;
```

```
si(cpt< N) alors
```

```
/* Non tous arrivés */
```

```
    V(mutex) ;
```

```
/* on libère mutex et */
```

```
    P(s) ;
```

```
/* on se bloque */
```

```
sinon
```

```
    V(mutex) ;
```

```
/* le dernier arrivé libère mutex et */
```

```
    Pour i=1 à N-1 faire V(s) ; /* réveille les N-1 bloqués, dans l'ordre d'arrivée */
```

```
Fin.
```


Solution2: Réveil en “cascade ”

```
Init(mutex,1) ; Init(s,0) ;
```

```
Int cpt=0 ;
```

Processus RDV2

Début

```
P(mutex) ;
```

```
cpt++ ;
```

```
si(cpt < N) alors
```

```
    V(mutex) ;
```

```
    P(s) ;
```

```
    V(s);
```

```
/* Non tous arrivés */
```

```
/* on libère mutex et */
```

```
/* on se bloque */
```

```
/* Réveiller le processus suivant */
```

```
Sinon
```

```
    cpt=0;
```

```
    V(mutex) ;
```

```
    V(s) ;
```

```
    P(s);
```

```
/* le dernier arrivé libère mutex et */
```

```
/* réveille le premier processus puis */
```

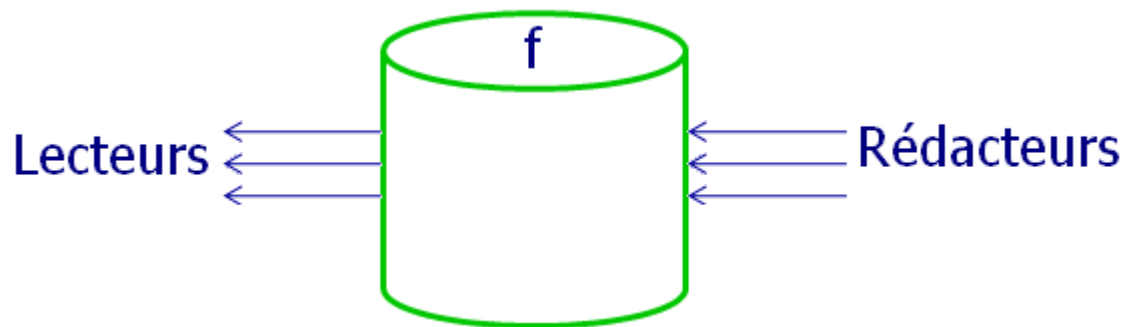
```
/* se met en attente */
```

```
/* le dernier processus est réveillé par l'avant dernier*/
```

```
Fin.
```

3. Problème des lecteurs/rédacteurs

- ❑ Considérons deux classes de processus appelés : **Lecteurs et Rédacteurs**.
- ❑ Ces processus se partagent un fichier f.
 - Les lecteurs peuvent seulement consulter le fichier,
 - les rédacteurs peuvent seulement écrire sur le fichier.



3. Problème des lecteurs/rédacteurs

- ❑ Les processus de ces deux classes doivent respecter les **contraintes suivantes**:
 - Plusieurs lecteurs peuvent lire simultanément le fichier.
 - Un seul rédacteur à la fois peut écrire sur le fichier.
 - Un lecteur et un rédacteur ne peuvent pas utiliser en même temps le fichier.

Problème des lecteurs/rédacteurs

□ Protocole utilisé par les processus:

Lecteurs	Rédacteurs
Demande de lecture <Lecture>	Demande d'écriture <Ecriture>
Fin de lecture (libérer le fichier)	Fin d'écriture (libérer le fichier)

□ Les conditions de franchissement

- **Accès en lecture:** si nombre d'écritures en cours (**ne**)=0;
- **Accès en écriture:** si nombre de lectures en cours (**nl**)=0 et **ne**=0;

Problème des lecteurs/rédacteurs

□ Un rédacteur :

- Exclut les autres rédacteurs ainsi que tous les lecteurs.
- L'accès en écriture se fait en exclusion mutuelle:
 - Un sémaphore appelé *accés* initialisé à 1.

□ Un lecteur :

- Exclut les rédacteurs **mais pas les autres lecteurs**.
- On utilise deux sémaphores :
 - Un pour l'accès à la variable *nl* (qui sert à savoir ou est le premier et le dernier lecteur),
 - Et un autre qui sert à l'accès en lecture (évidemment le sémaphore *accés*).

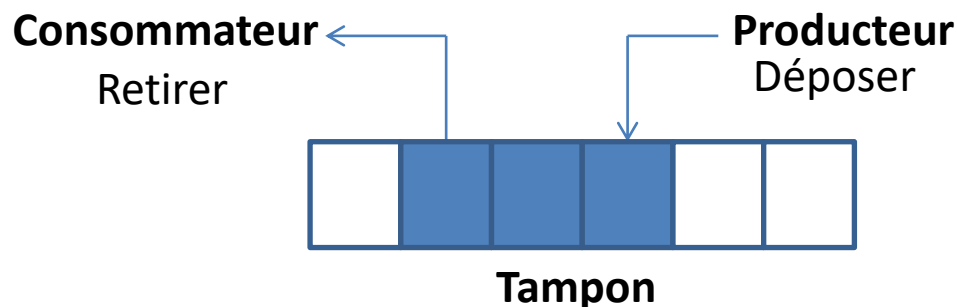
Problème des lecteurs/rédacteurs

```
acces, mutex: sémaphore;  
nl: entier;  
init(acces,1); init (mutex,1); nl=0;
```

Lecteurs	Rédacteurs
<pre>Début /* demande de lecture */ P(mutex); nl=nl+1; si nl=1 alors P(acces); V(mutex); < lire > /* fin de lecture */ P(mutex); nl=nl-1; si nl=0 alors V(acces); V(mutex); Fin</pre>	<pre>Début P(acces); /* demande d'écriture */ < écrire > V(acces); /* fin d'écriture */ Fin</pre>

4. Problème du producteur et du consommateur

- ❑ Soit T un tampon(buffer) accessible à 2 processus voulant communiquer.
- ❑ On distingue deux processus :
 - 1.**Producteur**: processus désirant déposer de l'information dans un tampon T
 - 2.**Consommateur** : un processus désirant retirer de l'information d'un tampon T.



Problème du producteur et du consommateur

❑ Contraintes de synchronisation

- Le producteur **ne peut déposer** un message que s'il y a de **la place disponible** dans le tampon.
- Le consommateur **ne peut retirer** un message que **s'il y en a de disponible**.
- Le producteur et le consommateur ne peuvent accéder **simultanément** à la même case.
- La politique de gestion du tampon est **FIFO**

Problème du producteur et du consommateur

Gestion du tampon:

- ❑ Le tampon contient **N** cases (**éléments**)
- ❑ Le tampon sera géré de manière **circulaire**
 - Les éléments du tampon sont numérotés de 0 à N-1.
 - On considère l'élément 0 comme le successeur de l'élément N-1.



Tampon circulaire

Problème du producteur et du consommateur

- ❑ Il y a deux types de ressources critiques: les cases **vides** et les cases **pleines**.
- ❑ On associe un sémaphore à chacune des ressources critiques :
 - Le sémaphore **Vide** initialisé au nombre de cases vides soit N , et
 - Le sémaphore **Plein** initialisé à 0 .

Problème du producteur et du consommateur

1. Le producteur s'alloue une case vide par une opération $P(\text{Vide})$,
2. Il remplit cette case vide et de ce fait génère une case pleine qu'il signale par une opération $V(\text{Plein})$.
3. Cette opération réveille éventuellement le consommateur en attente d'une case pleine.

Problème du producteur et du consommateur

tampon: tableau[0..N-1]de cases;
Plain, Vide: sémaphore;
init(Plein, 0); init(Vide, N);

Producteur

```
int i=0;  
Début  
  Répéter  
    Produire(message);  
    P(Vide);  
    tampon[i]=message ;  
    i=i+1 mod N ;  
    V(Plein);  
  Jusqu'à faux;  
Fin
```

Consommateur

```
init j=0;  
Début  
  Répéter  
    P(Plein);  
    message=tampon[j];  
    j=j+1 mod N ;  
    V(Vide);  
    Consommer(message);  
  Jusqu'à faux;  
Fin
```

Extension: Problème des producteurs consommateurs

- ❑ p producteurs et c consommateurs
- ❑ Ce cas pose deux problèmes d'exclusion mutuelle :
 - **Exclusion mutuelle entre producteurs** concurrence pour l'accès à l'index i
 - **Solution:** sémaphore d'exclusion mutuelle: $Mutex_i$
 - **Exclusion mutuelle entre consommateurs** concurrence pour l'accès à l'index j
 - **Solution:** sémaphore d'exclusion mutuelle: $Mutex_j$

Problème des producteurs consommateurs

tampon: tableau[0..N-1]de cases;

Plain, Vide, Mutexi, Mutexj: sémaphore;

init(Pain, 0); init(Vide, N); inti(Mutexi,1); inti(Mutexj,1);

Producteur k

int i=0;

Début

Répéter

Produire(message);

P(Vide);

P(Mutexi);

tampon[i]=message ;

i=i+1 mod N ;

V(Mutexi);

V(Plein);

Jusqu'à faux;

Fin

Consommateur l

init j=0;

Début

Répéter

P(Plein);

P(Mutexj);

message=tampon[j];

j=j+1 mod N ;

V(Mutexj);

V(Vide);

Consommer(Message);

Jusqu'à faux;

Fin

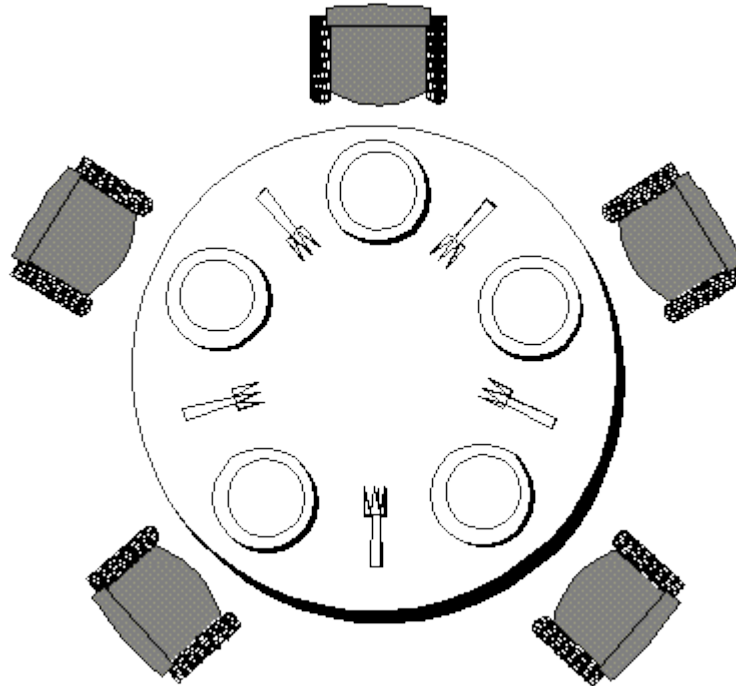
Problème des philosophes

(Dîner des philosophes)

- ❑ Un problème théorique proposé par Dijkstra.
- ❑ Cinq (5) philosophes sont assis autour d'une **table circulaire**.
 - Le problème peut être généralisé à **N** philosophes (tel que $N > 5$)
- ❑ Un philosophe passe son temps à **manger** et à **penser**.
- ❑ Sur la table, il y a alternativement cinq plats de spaghettis et cinq fourchettes

Problème des philosophes

- ❑ Pour manger, un philosophe a besoin de **deux fourchettes** qui sont de part et d'autre de son plat.

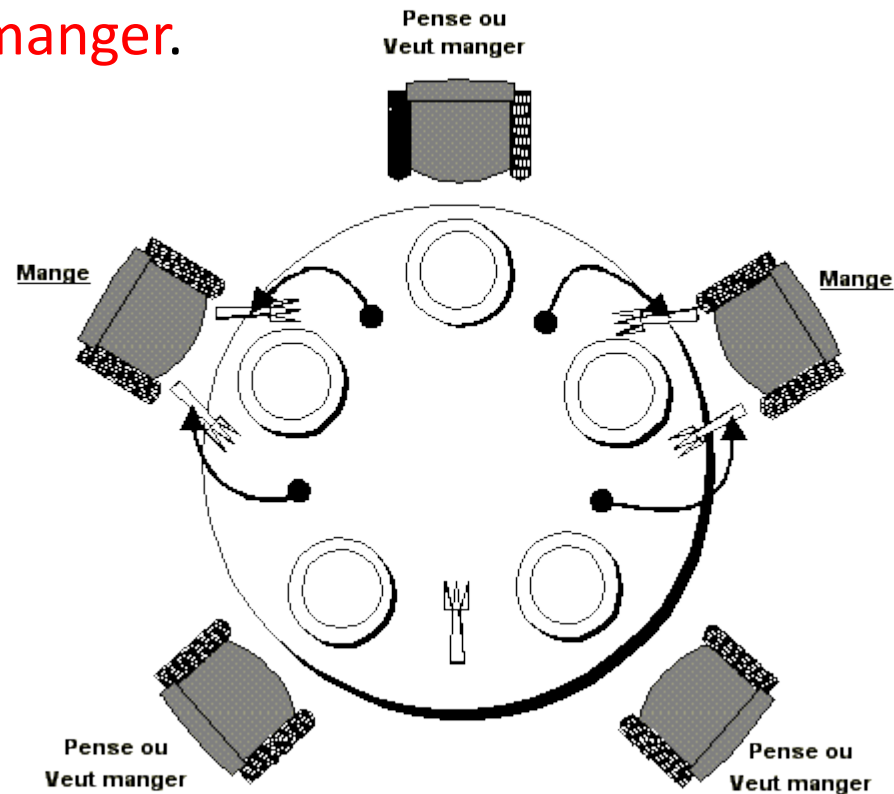


Problème des philosophes

- ❑ A tout instant, chaque philosophe est dans l'un des **états suivants** :
 - il mange avec deux fourchettes (de droite et de gauche);
 - il a faim, et attend la fourchette de droite, celle de gauche ou les deux ;
 - il pense, et n'utilise pas de fourchette.
- ❑ Initialement, tous les philosophes pensent.
- ❑ Un philosophe qui mange s'arrête en un **temps borné**.

Problème des philosophes

- Dîner des philosophes à un instant donné:
 - Si un philosophe mange, ces deux voisins immédiats ne peuvent pas manger.



Problème des philosophes

- ❑ Comment organiser la gestion des ressources (**fourchettes**) pour permettre aux philosophes de manger lorsqu'ils ont faim?
 - Les fourchettes sont des ressources **partagées** pour lesquelles les philosophes sont en concurrence.
- ❑ Comment éviter les situations de famine et d'interblocage?

Problème des philosophes

- ❑ On représente les philosophes par des processus.

Philosophe i

Début

tant que (vrai)

 début

 penser();

 prendre_fourchettes(de gauche et de droite);

 manger ();

 poser_fourchettes(de gauche et de droite)

 fin;

Fin.

Solution 1 (Idée)

- ❑ Les fourchettes sont des objets partagés (par deux philosophes) et doivent être protégées.
 - Chaque fourchette est représentée par un sémaphore initialisé à 1.
- ❑ Un philosophe appelle $P(\text{fourchette})$ avant de prendre une fourchette et appelle le $V(\text{fourchette})$ pour la libérer.

Solution 1 (naïve)

```
semémaphore fourch[ N ] = {1,.....1}  /* de 0 à N-1 */
```

```
Philosophe i
```

```
{  
  tant que (vrai)  
  {  
    penser();  
    P(fourch[i]); /* On attend la fourchette de gauche */  
    P(fourch[(i + 1) % N]); /* On attend la fourchette de droite */  
    manger();  
    V(fourch[i]); /* On libere la fourchette de gauche */  
    V(fourch[(i + 1) % N]); /* On libere la fourchette de droite N = 5 */  
  }  
}
```

Problème de la solution naïve

- ❑ **Interblocage** possible (si tous les philosophes prennent la fourchette de gauche, personne ne pourra prendre la fourchette à sa droite)
- ❑ **Causes du problème:** Chaque philosophe doit acquérir 2 ressources:
 - En 2 étapes
 - Dans un ordre qui peut mener à un blocage
 - Sans annulation possible
- ❑ Pour éviter l'interblocage, il faut éliminer un de ces trois éléments.

Quelques solutions pour éviter l'interblocage:

1. **Altern**er les choix des premières fourchettes
2. Ne pas permettre à **tous les philosophes** de s'asseoir sur la table en même.
 - Permettez seulement à quatre philosophes (ou moins) de s'asseoir à la table
3. Prendre les deux fourchettes dans une seule **section critique**
 - Un philosophe doit être autorisé à prendre les fourchettes seulement si les fourchettes gauche et droite sont disponibles

Solution 2

- ❑ Dans cette solution, on **modifie l'ordre** dans lequel le philosophe N-1 prend ses fourchettes.

```
sémaphore fourch[ N ] = {1,.....1}  /* de 0 à N-1 */
```

```
/* Philosophes 0 a N-2 */
```

```
Philosophe i
```

```
{
  tant que (vrai)
  {
    penser();
    P(fourch[i]); /*gauche*/
    P(fourch[(i + 1) % N]);
    manger()
    V(fourch[i]);
    V(fourch[(i + 1) % N]);
  }
}
```

```
/* Philosophes N-1 */
```

```
Philosophe N-1
```

```
{
  tant que (vrai)
  {
    penser();
    P(fourch[0]); /*droite*/
    P(fourch[N-1]);
    manger()
    V(fourch[0]);
    V(fourch[N-1]);
  }
}
```

Solution 3

- Si **seulement** quatre philosophes sont autorisés à s'asseoir, l'interblocage ne peut pas se produire.

```
semaphore fourch[5]= {1,1,1,1,1};  
semaphore chaise = 4;
```

```
/* Philosophes 0 a N-1*/
```

```
Philosophe i
```

```
tant que (vrai) {
```

```
    penser();
```

```
    P(chaise);
```

```
    P(fourch[i]);
```

```
    P(fourch[(i+1)%5]);
```

```
    manger();
```

```
    V(fourch[(i+1)%5]);
```

```
    V(fourch[i]);
```

```
    V(chaise);
```

```
}
```

Solution 4

- ❑ Chaque philosophe a trois états :
 - « PENSE », « A FAIM », « MANGE »
- ❑ Par lesquels il passe toujours dans cet ordre
 - Lorsqu'il a faim, un philosophe ne peut manger que si ses deux voisins ne mangent pas, sinon attend
 - Lorsqu'il termine de manger, le philosophe réveille ses voisins et se remet à penser

Solution 4

- ❑ Un **sémaphore** est attribué à **chaque philosophe**.
- ❑ Un philosophe qui veut prendre les fourchettes (donc manger) déclare qu'il a faim.
 - Si **l'un de ses deux voisins** est en train de manger, il se met en **attente**.
 - Si **les deux philosophes a coté ne mangent pas** alors il **peut prendre les deux fourchettes** et déclarer qu'il mange.
 - Quand le philosophe a **fini de manger**, il déclare donc qu'il **pense** (et réveil ses 2 voisins s'ils sont bloqués)

Solution 4

```
/*variables partagés*/  
sémaphore philo[ N ]={0,0,0,0,0}, mutex=1;  
etat etat_philo[ N ] = { PENSE, ....., PENSE}
```

```
prendre_fourchette (int i){  
  P(mutex );  
  etat_philo[ i ] = A_FAIM;  
  test_mange( i );  
  V(mutex );  
  P(philo[ i ]);  
}
```

```
poser_fourchette (int i){  
  P(mutex );  
  etat_philo[ i ] = PENSE;  
  test_mange((i+1)%N );  
  test_mange((i-1+N)%N );  
  V(mutex );  
}
```

```
test_mange( int i ){  
  if (etat_philo[ i ] == A_FAIM  
    && etat_philo[ (i+1)%N ] != MANGE  
    && etat_philo[ (i-1+N)%N ] != MANGE ) {  
    etat_philo[ i ] = MANGE;  
    V(philo[ i ]);  
  }  
}
```