Université Mohammed V- Rabat
École Mohammadia d'Ingénieurs
Département Génie Informatique

Filière Génie Informatique et Digitalisation

SYSTÈMES D'EXPLOITATION 2

Mr N. EL Faddouli

elfaddouli@emi.ac.ma, nfaddouli@gmail.com

2024-2025

1

Plan du cours

- Introduction et Rappels
- Gestion des processus: Synchronisation avec attente active
  - Définitions et rappels
  - Algorithmes avec attente active
- Gestion des processus: Synchronisation sans attente active
  - Les sémaphores
  - Les moniteurs
  - L'interblocage (Dead-lock)
- Gestion de la mémoire
  - Définitions et rappels
  - La mémoire virtuelle
- Les entrées/Sorties

2

Gestion des processus

Synchronisation sans attente active (Avec Arbitrage)

Introduction (1/2)

- Dans les solutions avec attente active (*sans arbitrage*), le code du contrôle de la concurrence est intégré dans le programme.
- Ce code englobe deux parties pour:
  - vérifie si le processus pourra entrer en section critique ou s'auto-bloquer (*dans une boucle*) jusqu'à ce qu'une condition soit satisfaite.
  - quitter la section critique en modifiant des variables pour débloquer un processus parmi ceux bloqués.
- Limites en termes de:
  - Consommation de CPU: chaque processus effectue beaucoup de vérifications inutiles et surcharge le CPU.
  - Complexité de programmation.
  - Scalabilité: Lorsque le nombre de processus augmente, les algorithmes avec attente active deviennent de moins en moins efficaces

Introduction (2/2)

- Solutions **avec arbitrage** (*avec blocage sans attente active*) : **Appel implicite à un arbitre**.

  1. Est-ce que je peux entrer en SC? (***demande à l'arbitre***)
  2. Si NON alors je me bloque
  3. <*section critique*>
  4. Je relâche la SC (***en le disant à l'arbitre***)

     S'il y a un processus bloqué alors le réveiller

- Au moment d'exécuter 2, le résultat de 1 n'est peut-être plus valide

  → **Réaliser 1 et 2 de manière indivisible**

- Le protocole de sortie doit aussi être réalisé de manière indivisible: *Modification de l'état d'occupation de la SC + réveil d'un processus bloqué*

Les Sémaphores (1)

- Introduit par Dijkstra en 1965 pour résoudre le problème d'exclusion mutuelle.
- Permet l'utilisation de **m** ressources identiques par **n** processus (**n>m**).
- Un sémaphore (*type abstrait de structure de données*) possède une **valeur entière** et une **file de processus en attente** de la ressource.
- Un sémaphore **S** est une variable **globale** protégée, **accessible** au moyen d'une **interface d'accès** ayant **deux opérations atomiques** :

  - **P(S)** (passer =“**P**asseren” en néerlandais):

Si la valeur de S est **<=0** Bloquer le processus appelant Sinon Décrémenter la valeur de S de 1

  - **V(S)** (sortir= “**V**rijgeven” en néerlandais):

Si la file de S est non vide Réveiller un processus bloqué cette la file Sinon Incrémenter S de 1

Les Sémaphores (2)

- Un sémaphore **binaire** (**booléen** ou d'**exclusion mutuelle - MUTEX**) est un sémaphore qui ne peut prendre que deux valeurs positives possibles: **1** et **0**.

- **P(S)**:

```
Si (S==1)  S=0;
Sinon  Bloquer le processus appelant dans la file d'attente de S
```

- **V(S)**:

```
Si la file de S est non vide
        Réveiller un processus en attente  dans la file de S
Sinon   S = 1;
```

- **Remarque:** S=1 ➔ accès à la SC

  S=0 ➔ pas d'accès à la SC

Les Sémaphores (3)

Exemple:

Pour le problème de **Lecteur-Rédacteur** avec **une** seule case:
on utilise deux sémaphores binaires **R** et **W**

Initialisation: W=1 et R=0 *// le rédacteur commence en premier*

```
P1:  (Rédacteur)
While (True) Do
      B=Créer_Msg();
      P(W);
          Write(B);
      V(R);
End
```

```
P2:  (Lecteur)
While (True) Do
      P(R);
          Read(B);
      V(W);
       Utiliser_Msg(B)
End
```

Exercices

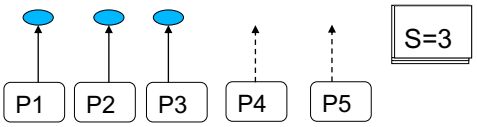
Les Sémaphores (4)

- Un sémaphore de **comptage** (ou **entier**) est un sémaphore pouvant prendre plus de deux valeurs positives possibles.
- Il est utile pour allouer une ressource parmi **plusieurs** exemplaires identiques : la valeur du sémaphore est initialisée avec le nombre de ressources disponibles.
- Il est associé à une ressource accessible par plusieurs processus.

Les Sémaphores (5)

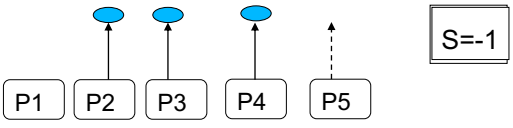
Soit **Q** le nombre de détenteurs potentiels (*nombre d'exemplaires*)

Initialisation: S=Q



P(S):

S ← S-1
Si (S<0) Alors Bloquer le processus appelant dans la file d'attente de S



V(S):

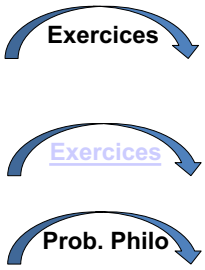
S ← S+1
Si (S<=0) Alors Réveiller un processus en attente dans la file de S

Remarque:

Si S<0 alors |S| = nombre de processus **bloqués** dans la file d'attente de S

Les Moniteurs: Exemple en Java

```
public synchronized  void  append(Object data)
{  if (count == N) plein.qWait();
buffer[in] = data;
in = (in + 1) % N; count++;
vide.qSignal();
}
public synchronized  Object  take()
{
Object  data;
    if (count == 0) vide.qWait();
data = buffer[out];
out = (out + 1) % N;
count--; plein.qSignal(); return data;
}}
```



Exercice: Problème des 5 philosophes

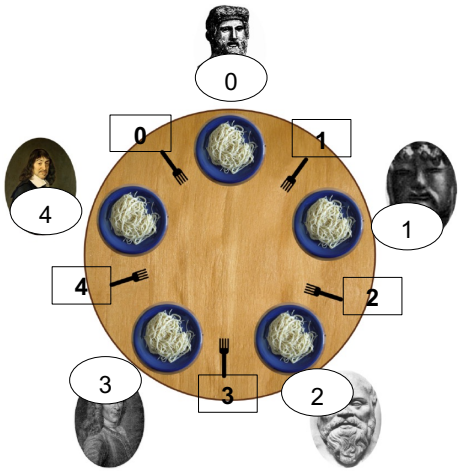
Chaque philosophe répète:

-Penser

-Prendre les fourchettes

-Manger

-Poser les fourchettes



Exercices: Problème des 5 philosophes *(suite)*

Solution 1: Prévoir un sémaphore d'exclusion mutuelle pour chaque fourchette.

Solution 2:

- Utiliser un sémaphore entier initialisé avec une valeur pouvant éviter l'attente circulaire.

Indication: C'est le nombre maximal de philosophes pouvant prendre des fourchettes simultanément sans provoquer un interblocage total.

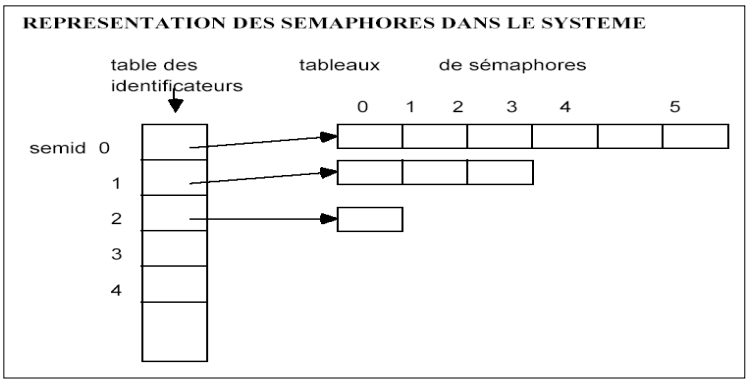
Exercices: Problème des 5 philosophes *(suite)*

Solution 3:

Utiliser un tableau de sémaphores binaire pour les philosophes ainsi qu'un tableau qui représente l'état des philosophes (pense, mange et affamé).

Création de sémaphore (1)

Dans Linux, chaque sémaphore (ou **tableau de sémaphores**) est identifié par un **identificateur**.



On peut ainsi créer un seul sémaphore ou un tableau de sémaphores selon les besoin en utilisant la fonction **semget.**

Création de sémaphore (1)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

int semget (key_t **Clé**, int **N**, int **Options**)

Retourne **l'identificateur** de l'ensemble de **N** sémaphores associés à une **Clé** (**entier** ou **IPC_PRIVATE**). En cas d'échec, la fonction retourne **-1**.

Options indique: Droits d'accès + Conditions de création (*des constantes*)

IPC_CREAT: Créer les N sémaphores et retourner l'identificateur. Si les sémaphores sont déjà crées par un autre processus (**Clé** déjà utilisée), la fonction retourne **l'identificateur** sans recréer les sémaphores.

IPC_ALLOC: Retourne l'identificateur des sémaphores s'ils sont déjà crées ou **-1** dans le cas contraire.

IPC_EXCL avec **IPC_CREAT:** Créer les sémaphores s'ils n'existent pas déjà, retourne **-1** dans le cas contraire.

Création de sémaphore (2)

Exemple1:

```
int  id1, id2;
// Créer un nouveau sémaphore dont la clé est 12.
id1= semget(12, 1, IPC_CREAT|IPC_EXCL|0666);

// Si le sémaphore est déjà crée,  on récupère son identifiant
if (id1== -1)    id1= semget (12, 1, IPC_ALLOC|0666);
 …….
// Créer un nouveau tableau de sémaphore dont la clé est donné par le système.
id2= semget (IPC_PRIVATE, 3, IPC_CREAT|0666);
```


Initialisation de sémaphore

int **semctl** (int ***semid***, int ***semnum***, int ***cmd***, *args_cmd*)

Effectuer l'opération ***cmd*** sur l'ensemble des sémaphores identifié par ***semid*** (ou sur le sémaphore d'indice ***semnum***, selon l'opération). Le premier sémaphore a l'indice **0**.

Les commandes possibles sont:

SETVAL: initialise le sémaphore d'indice ***semnum*** avec la valeur ***args_cmd***.

SETALL: initialise tous les sémaphores à l'aide des valeurs ***args_cmd***.

IPC_RMID: supprime le tableau de sémaphore ***semid.***

Exemple:

```
id2= semget (IPC_PRIVATE, 3, IPC_CREAT|0666);
id1= semget(12, 1, IPC_CREAT|IPC_EXCL|0666);
if (id1== -1)    id1= semget (12, 1, IPC_ALLOC|0666);
semctl (id1, 0, SETVAL, 1); // Initialisation à 1 du sémaphore d'indice 0
int  T[3] = {2, 0, 1}
semctl (id2, 3, SETALL, T); // initialiser les 3 sémaphores id2 par les valeurs  de T
```

Les opérations P et V (1)

int **semop**(int **semid**, **struct sembuf** ***Op**, unsigned int **N**)

Réaliser **atomiquement** un tableau **Op** de **N** opérations (P ou V) sur un ensemble de sémaphores indiqué par **semid**. **Op** est un tableau de **N** structures **sembuf**.

```
struct sembuf
  {  unsigned short int  sem_num;  /* numéro du sémaphore*/
     short    sem_op;     /* opération du sémaphore:
                    -1 (ou < 0) → l'opération P: décrémenter le sémaphore de |sem_op|
                    1 (ou > 0) → l'opération V:incrémenter le sémaphore de sem_op */

     short  sem_flg;  /*options */
  }
```

sem_flg : **- IPC_NOWAIT**: permet au processus de ne pas se bloquer sur une "ressource" indisponible, **semop** retourne l'erreur **EAGAIN** si le processus doit se bloquer. Utile si on veut juste vérifier la disponibilité d'une ressource.

- **SEM_UNDO** : si un processus est tué dans sa SC le système libère la "ressource" (annule l'opération effectuée ayant modifié la valeur du sémaphore)

Les opérations P et V (1)

```
void   P (int   id, int ind)      // id est l'identifiant d'un tableau de sémaphores
{                                 // ind est l'indice du sémaphore dans ce tableau
struct  sembuf   operation;  // Une seule opération
operation.sem_num = ind;     // Le sémaphore visé est celui d'indice 0
operation.sem_op = -1;       // Pour faire l'opération P
operation.sem_flg = 0;       // Ou  operation.sem_flg = SEM_UNDO
semop (id, &operation, 1); }
void   V (int   id, int ind)
{ struct  sembuf   operation;  // Une seule opération
operation.sem_num = ind;     // Le sémaphore visé est celui d'indice 0
operation.sem_op = 1;       // Pour faire l'opération V
operation.sem_flg = 0;       // Ou  operation.sem_flg = SEM_UNDO
semop (id, &operation, 1); }
```

Les opérations P et V (2)

```
void   PV (int   id, int ind, int semp_op)
{    // sem_op  est la valeur pour décrémenter ou incrémenter le sémaphore
d'indice ind dans le tableau d'identifiant id
struct  sembuf   operation;  // Une seule opération
operation.sem_num = ind;     // Le sémaphore visé est celui d'indice 0
operation.sem_op = semp_op;     // Pour décrémenter le sémaphore de |semp_op|
operation.sem_flg = 0;
semop (id, &operation, 1);
}
```


Création d'un segment de mémoire partagée

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

int shmget(key_t *Clé , size_t taille , int Options);

- Les arguments **Clé** et **Options** ont le même rôle que dans **semget**.
- Le 2ème argument est la taille en **octets** du segment de mémoire.
- La valeur de retour est l'identifiant du segment crée ou -1 en cas d'échec.

Exemple:

```
int segid;
segid = shmget(IPC_PRIVATE, 512, IPC_CREAT|0666);
.....
```

Attachement du segment au processus

void * shmat(int segid, void * adr, int options)

- **segid**: identifiant du segment mémoire.
- **adr**: adresse virtuelle d'attachement ou **NULL si le système doit la gérer.**

void * représente un pointeur **générique** qui est un pointeur compatible avec tout type de pointeurs.

- **options**: droit d'accès au segment par le processus (***lecture seule***, ***lecture-écriture***)
- valeur de retour: **adresse d'attachement** ou **-1** en cas d'échec.

Exemples: char * adrat; int * etat;

1) adrat = shmat(segid, NULL, **SHM_RDONLY);** */* lecture seule */*

2) etat= shmat(segid, NULL, **0);** */* ou* **SHM_RND** *lecture-écriture*/*