# Sommaire

# Rappels de cours

Un sémaphore est un mécanisme empêchant deux processus ou plus d'accéder simultanément à une ressource partagée.

*Sur les voies ferrées, un sémaphore empêche deux trains d'entrer en collision sur un tronçon de voie commune. Sur les voies ferrées comme dans les ordinateurs, les sémaphores ne sont qu'indicatifs : si un machiniste ne voit pas le signal ou ne s'y conforme pas, le sémaphore ne pourra éviter la collision.*

De même si un processus ne teste pas un sémaphore avant d'accéder à une ressource partagée, le chaos peut en résulter.

Un sémaphore binaire n'a que deux états :

- 0 verrouillé (ou occupé),
- 1 déverrouillé (ou libre).

Un sémaphore général peut avoir un très grand nombre d'états car il s'agit d'un compteur dont la valeur initiale peut être assimilée au nombre de ressources disponibles.

Par exemple, si le sémaphore compte le nombre d'emplacements libres dans un tampon et qu'il y en ait initialement 5 on doit créer un sémaphore général dont la valeur initiale sera 5.

Ce compteur :

- Décroît d'une unité quand il est acquis (“verrouillé”).
- Croît d'une unité quand il est libéré (“déverrouillé”).

Quand il vaut zéro, un processus tentant de l'acquérir doit attendre qu'un autre processus ait augmenté sa valeur car le **sémaphore ne peut jamais devenir négatif**.

L'accès à un sémaphore se fait généralement par deux opérations :

- P : pour l'acquisition (*Proberen*, tester) ou **P**(uis-je) accéder à une ressource ?
- V : pour la libération (*Verhogen*, incrémenter) ou **V**(as-y) la ressource est disponible

La notion de sémaphore est implémentée dans la plupart des systèmes d'exploitation. Il s'agit d'un concept fondamental car il permet une solution a la plupart des problèmes d'exclusion. Ce mécanisme fourni par le noyau nécessite la mise en oeuvre d'une variable (le sémaphore) et de deux opérations atomiques associées P et V.

# Les Sémaphores POSIX

Les **sémaphores POSIX** sont disponibles dans la librairie standard C (GNU). Utilisée sur la grande majorité des systèmes Linux, la **glibc** offre donc une implémentation des sémaphores.

Le sémaphore sera de type **sem_t**.

La liste des appels disponibles pour gérer un sémaphore sem_t est la suivante :

- sem_init
- **sem_wait**, sem_trywait, sem_timedwait
- sem_post
- sem_getvalue,
- sem_open sem_close sem_unlink sem_destroy

Une présentation des sémaphores POSIX est accessible en faisant :

```
$ man sem_overview
```

Pour la compilation, il suffit d'inclure le fichier *header* **semaphore.h** :

```
#include <semaphore.h>
```

A l'édition de liens, il ne faudra pas oublier d'utiliser **-lrt** ou **-lpthread**.

```
$ gcc exemple.c -lpthread -o exemple
```

## L'opération Init()

**sem_init()** initialise le sémaphore non nommé situé à l'adresse pointée par **sem**. L'argument **value** spécifie la valeur initiale du sémaphore.

Le prototype :

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

L'argument **pshared** indique si ce sémaphore sera partagé entre les **threads** d'un processus ou entre **processus**.

Si pshared vaut 0, le sémaphore est partagé entre les **threads** d'un processus et devrait être situé à une adresse visible par tous les threads (par exemple, une variable globale ou une variable allouée dynamiquement dans le tas).

```
// Utilisation d'Init() avec des threads
sem_t s; // variable globale

void Init(sem_t *s, unsigned int value)
{
   sem_init(&s, 0, value);
}
```

Si pshared n'est pas nul, le sémaphore est partagé entre processus et devrait être situé dans une région de mémoire partagée.

## L'opération P()

**sem_wait()** décrémente (verrouille) le sémaphore pointé par sem. Si la valeur du sémaphore est plus grande que 0, la décrémentation s'effectue et la fonction revient immédiatement. Si le sémaphore vaut zéro, l'appel bloquera jusqu'à ce que soit il devienne disponible pour effectuer la décrémentation (c'est-à-dire la valeur du sémaphore n'est plus nulle), soit un gestionnaire de signaux interrompe l'appel.

sem_trywait() est pareil que sem_wait(), excepté que si la décrémentation ne peut pas être effectuée immédiatement, l'appel  renvoie une erreur (errno vaut EAGAIN) plutôt que bloquer.

sem_timedwait() est pareil que sem_wait(), excepté que l'on peut spécifier une limite sur le temps pendant lequel l'appel bloquera si la décrémentation ne peut pas être effectuée immédiatement.

```
void P(sem_t *s)
{
   sem_wait(s);
}
```

## L'opération V()

**sem_post()** incrémente (déverrouille) le sémaphore pointé par sem. Si, à la suite de cet incrément, la valeur du sémaphore devient supérieure à zéro, un autre processus ou thread bloqué dans un appel sem_wait() sera réveillé et procédera au verrouillage du sémaphore.

```
void V(sem_t *s)
{
   sem_post(s);
}
```

# Les sémaphores System V

Les **sémaphores System V** (semget, semop, etc.) sont une ancienne API de sémaphores. Les sémaphores POSIX fournissent une interface bien mieux conçue que celles de System V.

D'un autre coté, les sémaphores POSIX sont (peut-être) moins largement disponibles (particulièrement sur d'anciens systèmes) que ceux de System V.

Ces sémaphores font partie des mécanismes System V de communication entre processus (**IPC**) qui sont : les **files de messages**, les ensembles de **sémaphores** et les segments de **mémoire partagée**.

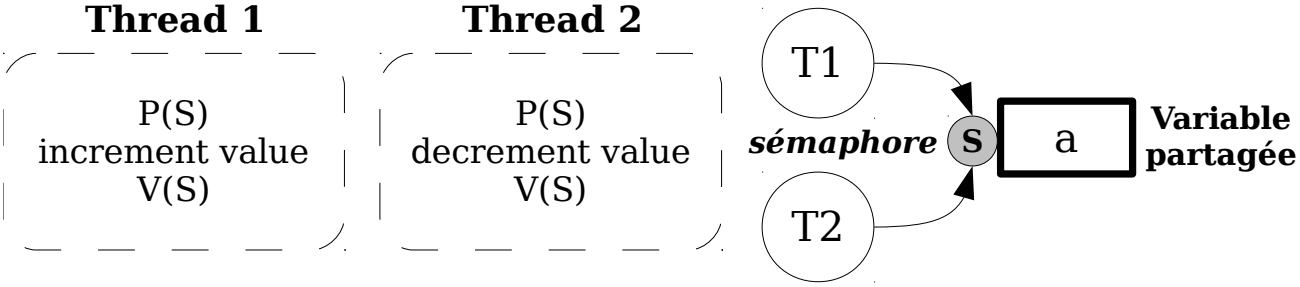
La commande **ipcs** fournit des informations sur les IPC.

# Séquence 1 : Exemple

Compiler et tester l'exemple **intro.c** qui met en oeuvre deux taches et un sémaphore d'exclusion mutuelle pour accéder à une section critique.

**1) À partir de la trace d'exécution, dessiner un diagramme temporel.**

À partir de cette exemple, on va définir la section critique pour ces deux tâches : une incrémentera la variable partagée et l'autre la décrémentera.




**2) Écrire le programme semaphores.1.c qui illustre la synchronisation de donnée en utilisant un sémaphore.**

# Séquence 2 : Rendez-Vous

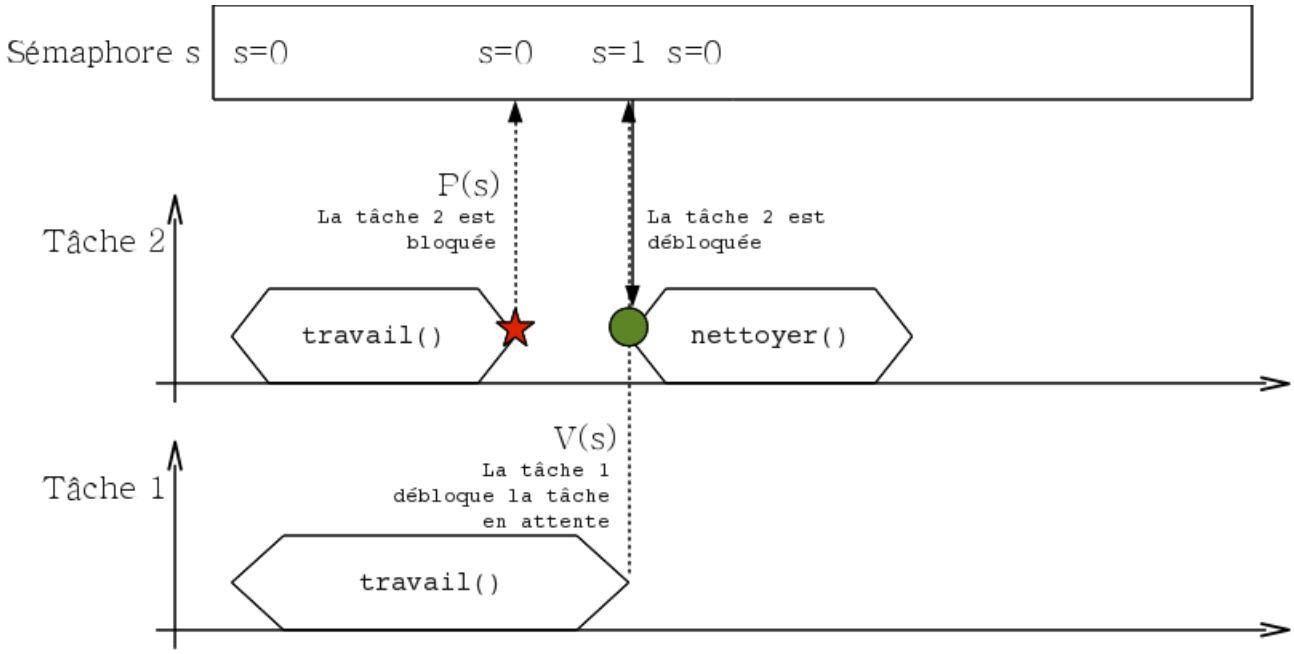
## Synchronisation de processus

On a deux tâches qui réalisent chacune leur travail (fonction travail()). On désire que la tâche 2 exécute une fonction nettoyer() lorsque la tâche 1 a terminé son travail.

**3) En utilisant un sémaphore de synchronisation, écrire la séquence de pseudo-code de T1 et T2 permettant d'établir cette synchronisation.**

**4) Compléter (les parties notées TODO) et tester le source rendez-vous1.c**

**5) Donner la trace d'exécution mettant en évidence la synchronisation entre les deux tâches.**



## Rendez-vous à N

Un processus doit attendre que n autres processus ou tâches soient parvenus à un endroit précis pour poursuivre son exécution.

On utilisera :

- un sémaphore mutex initialisé à 1 (pour l'accès à nb)
- un sémaphore Sattend initialisé à 0 (pour la synchronisation)
- un entier nb initialisé à 0 (pour compter les tâches)

Le pseudo-code de la barrière de synchronisation sera le suivant :

```
Fonction rendezVous
début
   P(mutex)
   nb := nb + 1
   SI(nb < N) /* tous arrivés ? */
   ALORS /* non */
      V(mutex)
      P(Sattend)
   SINON /* oui */
      POUR chaque tâche en attente FAIRE
         V(Sattend)
      FIN POUR
      nb := 0
      V(mutex)
   FIN SI
fin
```

**6) Coder la fonction rendezVous (la partie notée TODO) et tester le source rendez-vous2.c**

**7) Donner la trace d'exécution mettant en évidence la synchronisation entre les N tâches.**

# Séquence 3 : Producteur/Consommateur

Un producteur produit des informations et les place dans un tampon (une file de type FIFO). Un consommateur vide le tampon et consomme les informations.

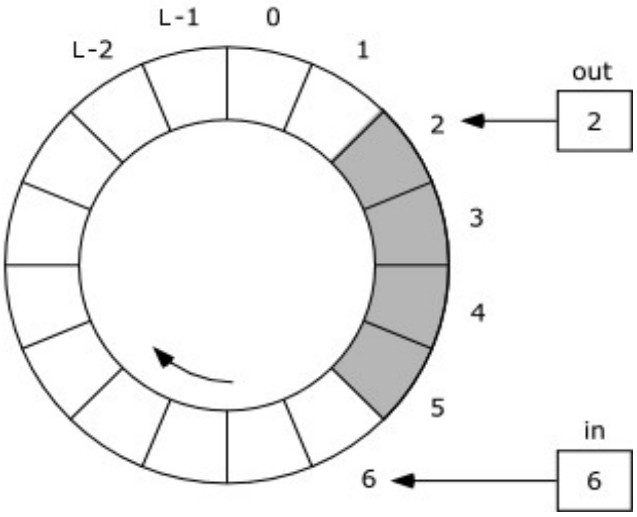
La tâche Producteur réalise les actions suivantes :

```
construireRequete();
deposerRequete(requete);
```

La tâche Consommateur réalise les actions suivantes :

```
retirerRequete();
traiterRequete(requete);
```

La file est gérée comme un tampon circulaire (L représente ici la capacité de la file ).




La file est gérée à l'aide de 2 variables partagée in et out :

- in pointe sur le premier emplacement libre
- out pointe sur la prochaine requête à retirer

Pour maintenir sans débordement, un indice i sur un tampon circulaire de taille L, on utilise l'opérateur MODULO %. Pour faire "avancer" l'indice i dans le tampon circulaire, on procédera donc de la manière suivante :

```
i = (i + 1) % L;
```

## Situation 1

On se propose d'examiner la situation de synchronisation suivante :

- Une seule tâche produit des requêtes et une seule tâche les retire.
- La file est bornée de taille L.

Le Producteur ne peut déposer que s'il existe une place libre dans la file.

Le Consommateur ne peut prélever une requête que si elle a été préalablement déposée.

La file est une ressource épuisable mais n'est pas une ressource critique car le Producteur et le Consommateur n'en manipulent pas la même partie.

La synchronisation du Producteur vis à vis de la file est à mettre en place.

La synchronisation du Consommateur vis à vis du Producteur est à mettre en place.

On utilisera donc deux sémaphores qu'il vous faudra initialiser :

```
sem_t nbRequetes;
sem_t nbCasesVides;
```

**8) Coder les tâches producteur et consommateur (les parties notées TODO) et tester le source producteur1.c**

**9) Fournir une trace d'exécution mettant en évidence la synchronisation entre les 2 tâches.**

## Situation 2

On se propose maintenant d'examiner la situation de synchronisation suivante :

- Plusieurs tâches produisent des requêtes. Plusieurs tâches les retirent.
- La file est bornée de taille L.

Le Producteur ne peut déposer que s'il existe une place libre dans la file.

Le Consommateur ne peut prélever une requête que si elle a été préalablement déposée.

La file est une ressource épuisable mais elle est aussi une ressource critique pour les Producteurs qui doivent déposer en exclusion mutuelle.

La file est une ressource critique pour les Consommateurs qui doivent prélever en exclusion mutuelle.

La synchronisation du Producteur vis à vis de la file est à mettre en place.

La synchronisation du Consommateur vis à vis du Producteur est à mettre en place.

On ajoutera 2 mutex :

- un sur in pour assurer le dépôt en exclusion mutuelle
- un sur out pour assurer le prélèvement en exclusion mutuelle

On utilisera donc quatre sémaphores qu'il vous faudra initialiser :

```
sem_t nbRequetes;
sem_t nbCasesVides;
sem_t mutexP;
sem_t mutexC;
```

**10) Coder les tâches producteur et consommateur (les parties notées TODO) et tester le source producteur2.c**

**11) Fournir une trace d'exécution mettant en évidence la synchronisation entre les N tâches.**

# Séquence 4 : Lecteur-Rédacteur

N tâches réparties en 2 catégories : les Lecteurs et les Rédacteurs, se partagent une ressource commune : le Fichier. Les lecteurs peuvent lire simultanément le fichier. Les rédacteurs doivent avoir un accès exclusif au fichier : lorsqu'un Rédacteur écrit, aucune Lecture ni aucune Ecriture n'est possible.

Les programmes Rédacteur réalisent les actions suivantes :

```
ecrireFichier();
```

Les programmes Lecteur réalisent les actions suivantes :

```
lireFichier();
```

## Situation 1

On se propose d'examiner la situation de synchronisation suivante dans le cadre du problème des lecteurs et des rédacteurs :

Lorsque aucun rédacteur ne lit le fichier, Lecteurs et Rédacteurs ont même priorité. Les Lecteurs ont priorité sur les rédacteurs si un Lecteur occupe déjà le fichier.

Le premier Lecteur doit donc bloquer l'accès au fichier pour les éventuels Rédacteurs et laisser les autres Lecteurs accéder au fichier.

Le dernier Lecteur qui libère le fichier doit permettre aux éventuels Rédacteurs d'accéder au fichier.

Les Lecteurs doivent donc se compter (variable nbLecteurs) et doivent accéder à cette variable en exclusion mutuelle.

On utilisera donc deux sémaphores qu'il vous faudra initialiser :

```
sem_t mutexL; // pour gérer l'accès à la variable nbLecteurs
partagée
sem_t w; // pour gérer l'accès en écriture
```

**12) Coder les tâches redacteur et lecteur (les parties notées TODO) et tester le source lecteur1.c**

**13) Fournir une trace d'exécution mettant en évidence la synchronisation entre les N tâches.**

## Situation 2

On se propose maintenant d'examiner la situation de synchronisation suivante :

Les Lecteurs ont toujours priorité sur les rédacteurs. Le cas où un lecteur attend est celui où un Rédacteur occupe le fichier. Un Rédacteur n'accède au fichier que si aucun lecture n'est en cours ou en attente.

Tant qu'un Lecteur accède au fichier, aucun Rédacteur ne peut y accéder. Le seul moment où un Rédacteur peut obtenir la priorité sur un Lecteur est le cas où pendant qu'un Rédacteur accède au fichier et avant qu'il ne termine d'y accéder, un autre Rédacteur puis un Lecteur demandent l'accès. Pour garantir que, dans cette situation, c'est un Lecteur qui obtiendra bien la priorité, il faut pouvoir bloquer les Rédacteurs sur un autre sémaphore mutexR en amont de leur opération P(w).

On ajoutera donc un sémaphore :

```
sem_t mutexR;
```

**14) Coder les tâches redacteur et lecteur (les parties notées TODO) et tester le source lecteur2.c**

**15) Fournir une trace d'exécution mettant en évidence la synchronisation entre les N tâches.**

# Bonus : Le problème du coiffeur endormi

Il s'agit d'un de ces problèmes de synchronisation mis sous une forme "plaisante". On en trouve une application presque directe dans certains mécanismes des systèmes d'exploitation (comme l'ordonnancement des accès disque).

Ce problème classique de synchronisation est le suivant : un coiffeur dispose dans son salon d'un fauteuil de coiffure et de N chaises pour les clients en attente. S'il n'y a pas de clients, le coiffeur dort.

À l'arrivée du premier client, il se réveille et s'occupe de lui. Si un nouveau client arrive pendant ce temps, il attend sur l'une des chaises disponibles. Mais si toutes les chaises sont occupées, le client s'en va.

Enfin lorsque le coiffeur a fini de couper les cheveux du dernier client, il peut se rendormir.

Indications :

- utiliser un sémaphore pour gérer les clients en attente, et un autre indiquant si le coiffeur est disponible.
- ajouter une variable indiquant le nombre de clients en attente.

**16) Faut-il un mutex sur la variable indiquant le nombre de clients en attente ?**

**17) Dessiner le problème.**

**18) Compléter le programme coiffeur.c fourni en évitant les accès concurrents.**

**19) Faire une trace de votre programme.**