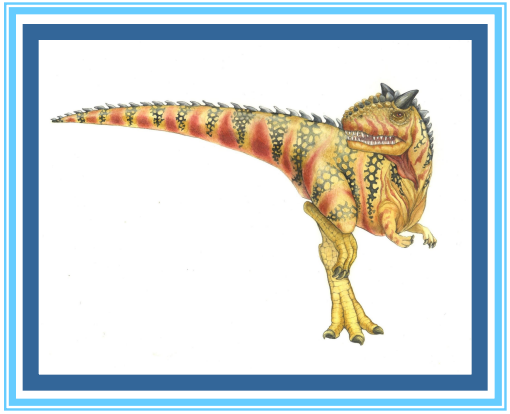
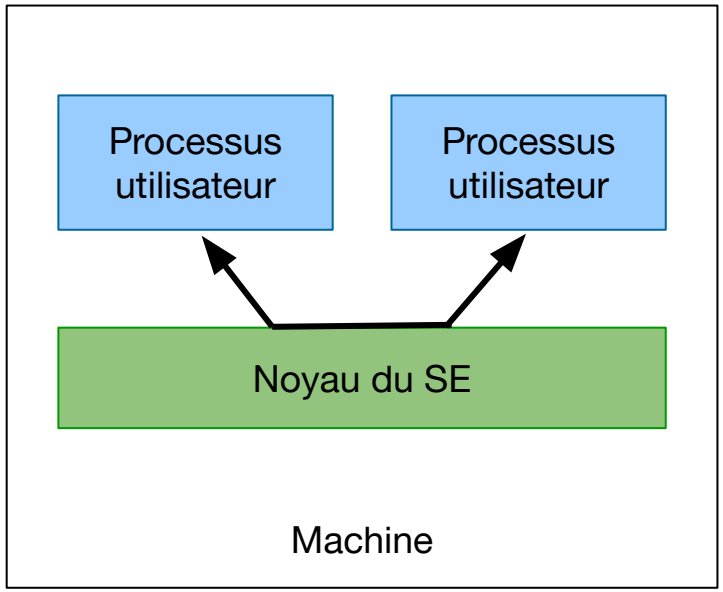
Chapitre 6 : Synchronisation



Contexte (1)

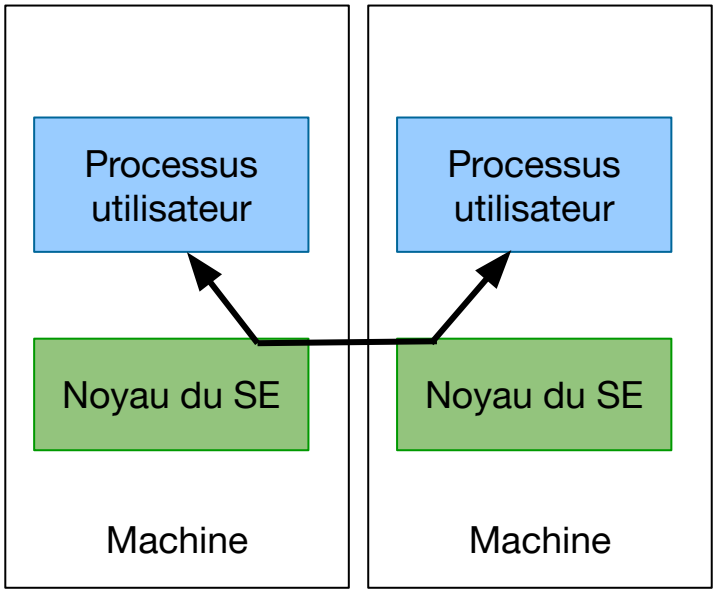
Processus **concurrents** vs. Processus distants

- **Communications intra-système**

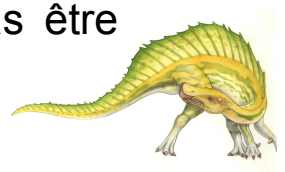


Dans les systèmes centralisés, les processus communiquent par l'intermédiaire de variables et d'objets partagés.

- **Communications inter-système**



Dans les systèmes répartis, où il n'existe pas de mémoire commune, les communications se font par messages et peuvent ne pas être instantanées.



Contexte (2)

- Plusieurs processus → **accès concurrents** aux ressources
- Une **ressource** désigne toute entité dont a besoin un processus pour s'exécuter :
  - ressource **matérielle** (processeur, périphérique, etc.)
  - ressource **logicielle** (variable)
- **3 phases** pour l'exploitation d'une ressource par un processus :
  - **sollicitation de la ressource**
  - **utilisation de la ressource**
  - **libération de la ressource**

Contexte (3)

- Une ressource est dite **critique** lorsque des accès concurrents à cette ressource peuvent mener à un état incohérent
- On parle aussi de **situation de compétition** (**race condition**) pour décrire une situation dont l'issue dépend de l'ordre dans lequel les opérations sont effectuées
- Une **section critique** est une section de programme manipulant une ressource critique
- Un mécanisme d'**exclusion mutuelle** sert à assurer l'atomicité des sections critiques relatives à une ressource critique

Accès aux ressources critiques (1)

- Cas d'**incohérence de données** : problème de la synchronisation relative de l'exécution des processus

process#1

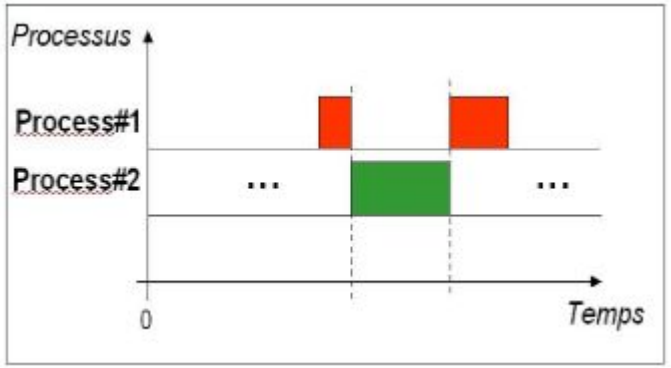
```
lire (N, Account)

N :=N + 100
écrire (N, Account)
```

process#2

```
lire (N, Account)
N :=N + 100
écrire (N, Account)
```

diagramme de gantt



➔ `Process#2` **ne doit pas accéder à N tant que** `process#1` **l'utilise !**

Accès aux ressources critiques (2)

- **Solution par exclusion mutuelle :**

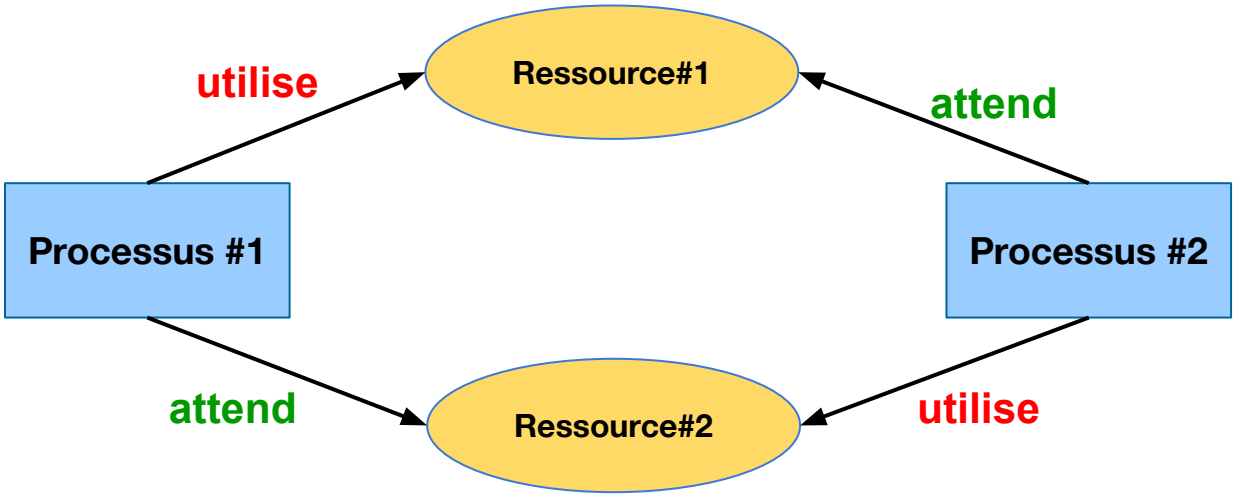
```
Variable Compte  : entier

Procédure Credit (somme_à_créditer : entier)
Début
   Début_section_critique()
       Compte = Compte + somme_à_créditer
   Fin_section_critique()
   Ecrire (« Opération de crédit effectuée »)
Fin

Procédure Débit (somme_à_débiter: entier)
Début
   Début_section_critique()
       Compte = Compte - somme_à_débiter
   Fin_section_critique()
   Ecrire(« Opération de débit effectuée »)
Fin
```

Accès aux ressources critiques (3)

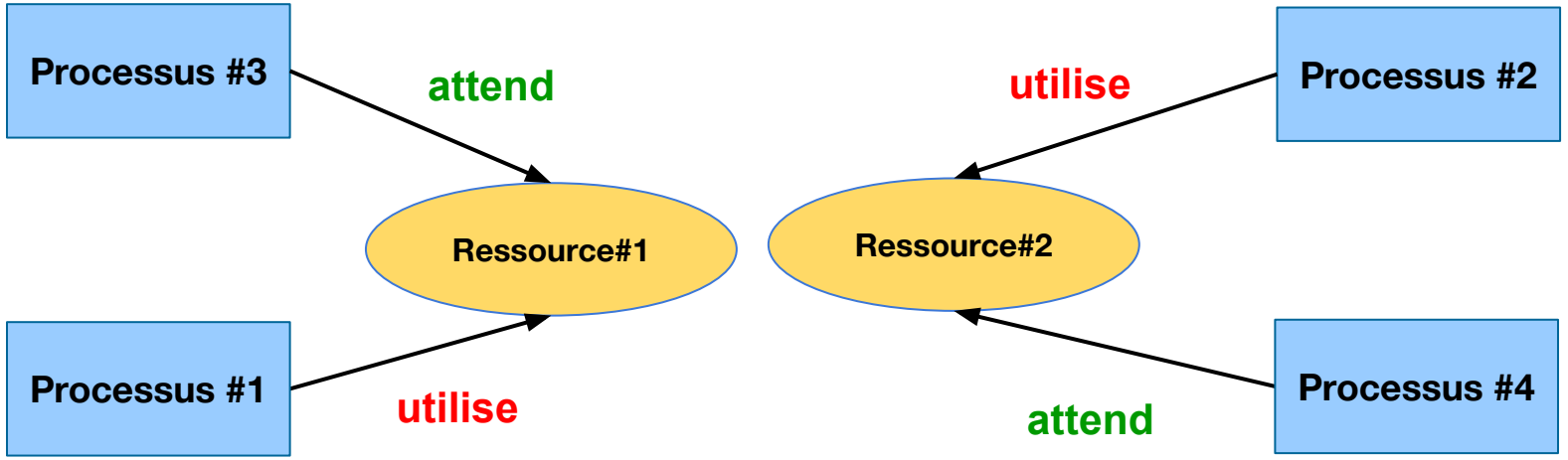
- Cas d'**interblocage** : ensemble de processus attendant chacun une ressource déjà possédée par un processus de l'ensemble



L'attente est infinie !

Accès aux ressources critiques (4)

- Cas de **coalition et famine** : ensemble de processus monopolisant des ressources au détriment d'autres processus.



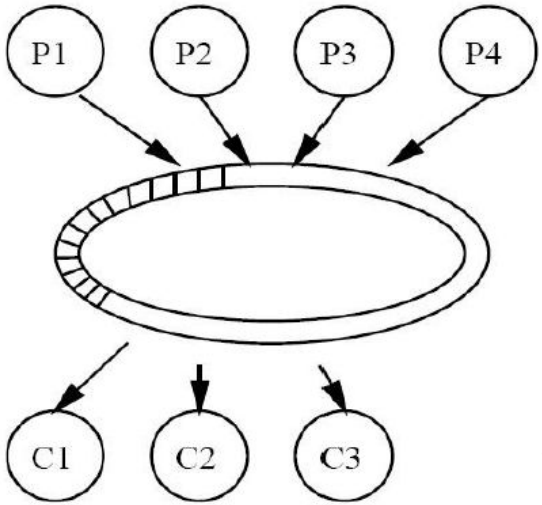
L'attente est indéfinie !

Paradigmes de concurrence

- Problème des producteurs/consommateurs
- Problème des lecteurs/rédacteurs
- Problème des philosophes

Problème des producteurs/consommateurs

- Un ensemble de processus, **divisés en deux catégories**, partagent une zone mémoire
  - Les premiers (**producteurs**) remplissent la mémoire partagée avec des éléments
  - Les seconds (**consommateurs**) utilisent ces éléments et les retirent de la mémoire
- **Exemple** : la file d'impression



Problème des lecteurs/rédacteurs (1)

- Un ensemble de processus, divisés en deux catégories, partagent une zone mémoire
  - Certains processus (**les lecteurs**) font des accès en lecture seule à cette zone
  - D'autres processus (**les rédacteurs)** modifient le contenu de cette zone, les rédacteurs sont parfois appelés écrivains

Problème des lecteurs/rédacteurs (2)

- **Principe**
  - Lorsqu'un rédacteur accède à la mémoire partagée, aucun autre processus (lecteur ou rédacteur) ne doit y avoir accès
  - Les lecteurs peuvent être plusieurs à utiliser la zone en même temps

- La mémoire partagée doit être protégée par exclusion mutuelle

- Les lecteurs n'ont besoin de cette exclusion mutuelle que dans le cas où aucun autre lecteur n'utilise la mémoire

  - Utilisation d'un **compteur**

Problème des philosophes

Principe

- **5 philosophes** passent leur vie à manger et à penser (réfléchir)
- Chaque philosophe alterne entre deux actions :
  - **Réfléchir** — il ne touche à aucune fourchette.
  - **Manger** — pour manger, il a besoin de deux fourchettes : celle à sa gauche et celle à sa droite.
- Chaque philosophe a 3 états : « Je pense », « J'ai faim », « Je mange » par lesquels il passe toujours dans cet ordre
- Lorsqu'il a faim, un philosophe ne peut manger que si ses 2 voisins ne mangent pas, sinon il attend
- Lorsqu'il termine de manger, le philosophe réveille ses voisins et se remet à penser

Résolution des problématiques

Comment gérer les accès concurrents aux ressources ?

- Introduction de nouveaux mécanismes
  - **de communication**
  - **de synchronisation**

Communications inter-processus

- **I**nter-**P**rocess **C**ommunication (**IPC**) : méthodes permettant à plusieurs processus de communiquer entre eux

- **3 catégories de mécanismes :**
  - outils permettant aux processus de s'**échanger des données**
    - les fichiers
    - la mémoire partagée
  - outils permettant de **synchroniser des processus**
    - les sémaphores
    - les signaux
  - outils permettant d'**échanger des données et de synchroniser des processus**
    - les tubes
    - les files d'attente de messages

Communication par fichier

- Les fichiers constituent la manière primitive de communication. Approche coûteuse (accès au système de fichier du disque)

- Afin d'éviter les accès concurrents en lecture / écriture, on procède à des verrouillages (**Verrou**).

  - **Verrouillage de tout le fichier :**

    flock()

    LOCK_SH : verrouillage en mode partagé

    LOCK_EX : verrouillage en mode exclusif

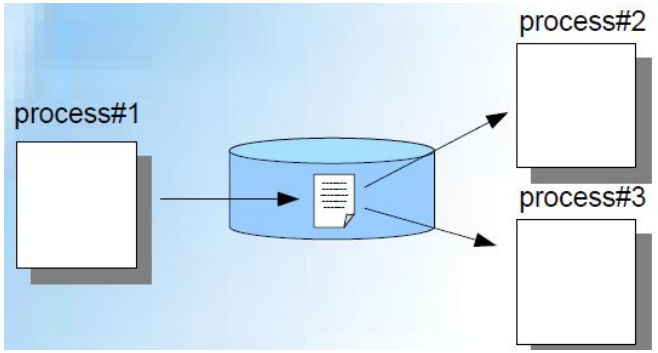
  - **Verrouillage d'une partie du fichier :**

    lockf()

    F_TEST : teste si une zone est verrouillée

    F_LOCK : verrouille une zone

    fcntl()



Communication par mémoire partagée

- Un segment de mémoire peut être simultanément attaché à l'espace virtuel de plusieurs processus, ou plusieurs fois à des adresses différentes d'un même processus

- **Primitives C de gestion d'un segment de mémoire partagée :**

  `shmget()` : **création ou accès** à un segment de mémoire existant

  `shmat()` : **attachement** à l'espace d'adressage d'un processus

  `shmdt()` : **détachement** du segment de mémoire partagée de l'espace d'adressage

  `shmctl()` : **contrôle du segment** (permissions d'accès, swapping, etc.)

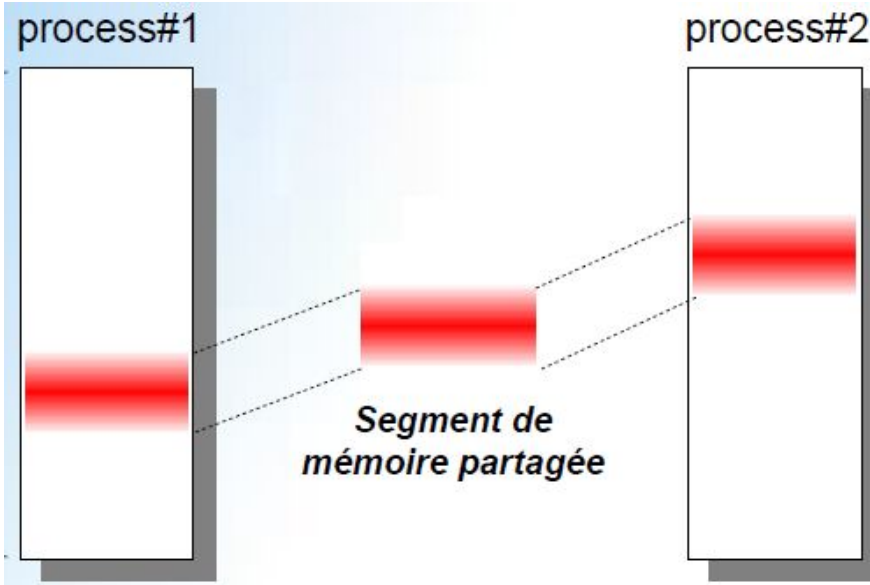
IPC_STAT

IPC_SET

SHM_LOCK

SHM_UNLOCK

…



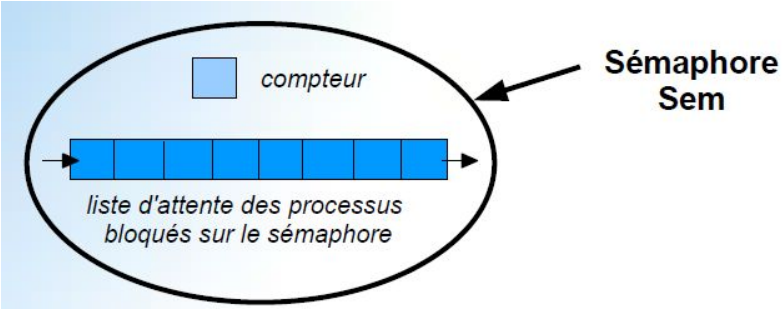
Synchronisation par sémaphore (1)

Qu'est-ce qu'un Sémaphore ?

- Un sémaphore est **une variable entière spéciale** contenant un compteur (valeur entière non négative) utilisée pour contrôler l'accès à des ressources partagées par plusieurs processus ou threads dans un environnement concurrent.

- gérant une file d'attente de processus attendant qu'advienne une condition particulière propre au sémaphore

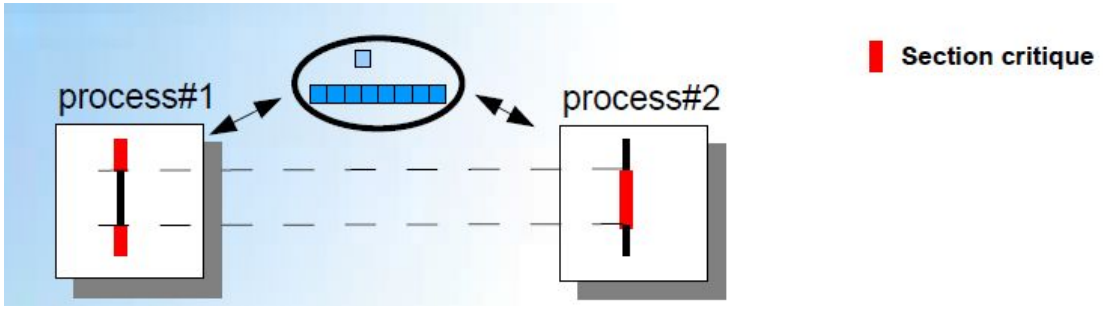
- 👉 **Il sert à synchroniser et à éviter les conflits d'accès** (race conditions) dans les programmes.



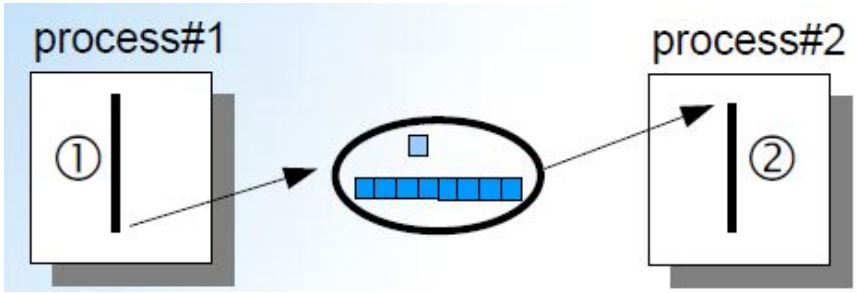
Synchronisation par sémaphore (2)

Types de Sémaphores

1. **Exclusion mutuelle : Sémaphore Binaire (Mutex)**
   - Sa valeur est soit 0 (occupé) soit 1 (libre).
   - Sert souvent à verrouiller une seule ressource.



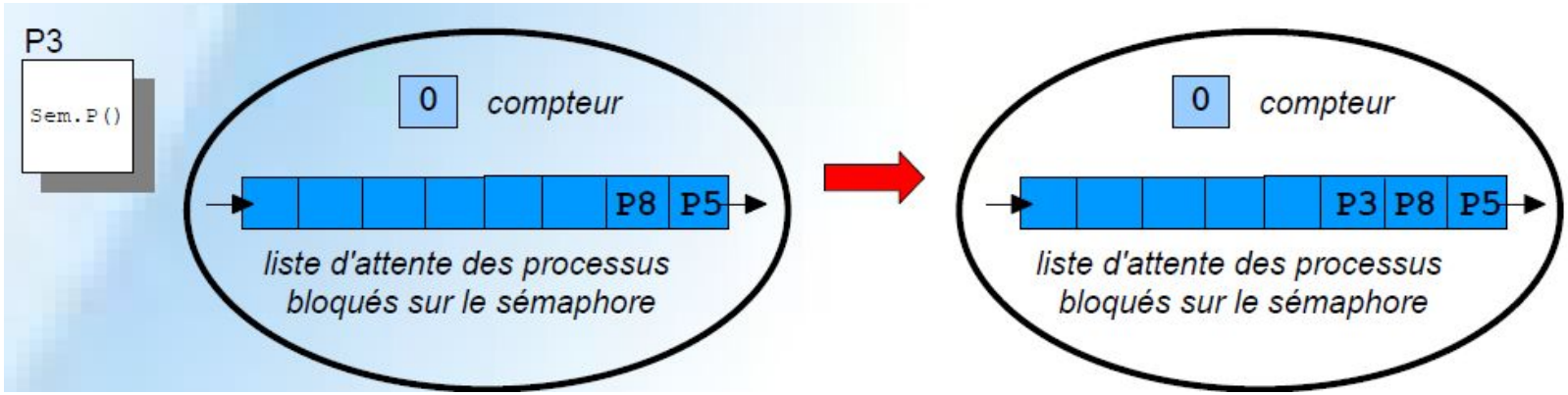
2. **Barrière de synchronisation : Sémaphore Comptant**
   - Sa valeur est un nombre entier $N \geq 0$.
   - Contrôle l'accès à N ressources identiques.
   - Exemple : 5 connexions réseau autorisées, donc le sémaphore commence à 5.



Synchronisation par sémaphore (3)

- 1ère opération : **test de prise** du sémaphore **(« Puis-je ?)**

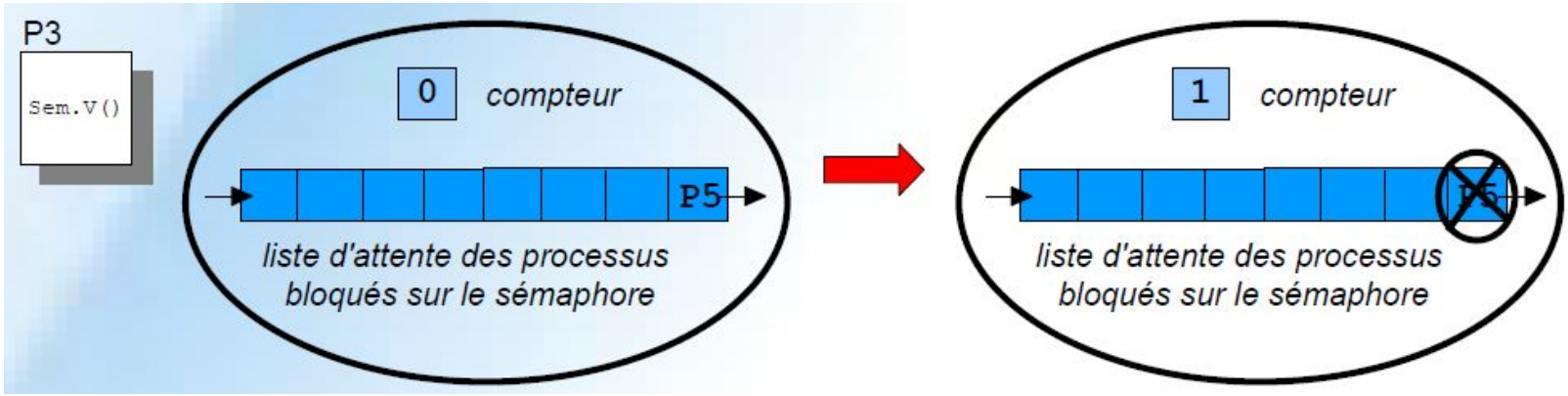
```
Sem.P() : si (Sem.compteur>0)
          alors Sem.compteur = Sem.compteur - 1
          sinon insère_ce_processus(Sem.file)
       finsi
```



Synchronisation par sémaphore (4)

- 2ème opération **: libération** du sémaphore **(Vas-y ! »)**

```
Sem.V() : Sem.compteur = Sem.compteur + 1
        si (Sem.compteur > 0)
            alors extrait_un_processus(Sem.file)
        finsi
```



Synchronisation par sémaphore (5)

Primitives C de gestion des sémaphores UNIX (IPC System V) :

semget()

création d'un tableau de sémaphores

semop()

test de prise d'un sémaphore (sem.P())

libération d'un sémaphore (sem.V())

semctl()

initialisation d'un sémaphore

suppression d'un sémaphore

Synchronisation par signaux (1)

- Les **signaux** permettent **d'avertir** simplement un processus **de l'arrivée** d'un **événement**

- Les signaux sont limités au rôle de **notification** d'événement

Synchronisation par signaux (2)

- A chaque signal est associé un comportement par défaut :
  - **abort** : génération d'un fichier core et arrêt du processus
  - **exit** : terminaison du processus sans génération d'un fichier core
  - **ignore** : le signal est ignoré
  - **stop** : suspension du processus
  - **continue** : reprendre l'exécution si le processus a été suspendu
- Il existe environ 32 signaux prédéfinis identifiés par un nom et un numéro

Synchronisation par signaux (3)

Numéro	Nom	Comportement par défaut
1	SIGUP	envoyé lors de la déconnexion
2	SIGINT	interruption clavier
3	SIGQUIT	quitter depuis le clavier
6	SIGABRT	terminaison anormale
8	SIGFPE	erreur opération en virgule flottante
9	SIGKILL	destruction inconditionnelle du processus
10	SIGUSR1	à disposition de l'utilisateur
11	SIGSEGV	référence mémoire invalide
12	SIGUSR2	à disposition de l'utilisateur
13	SIGPIPE	erreur dans un tube (écriture sans lecteur)
14	SIGALARM	signal d'horloge
15	SIGTERM	terminaison normale d'un processus
18	SIGCONT	reprise d'exécution, si stoppé
19	SIGSTOP	stoppe l'exécution d'un processus

Synchronisation par signaux (4)

- 2 façons d'envoyer un signal à un processus :

  - **Un caractère de contrôle :**

    CRTL-C envoie SIGINT (interruption)

    CRTL-Z envoie SIGSTOP (suspension)

    CRTL-\ envoie SIGQUIT (fin)

  - **En ligne de commande :**

    kill [-no_signal] no_processus

    kill -9 4014

    kill [-nom_signal] no_processus

    kill -SIGUSR1 2584

Synchronisation par signaux (5)

- Modification du comportement par défaut d'un signal :

```
trap 'instruction1; instruction2;...' nom_signal
trap 'echo fini;exit 1' SIGUSR1
```

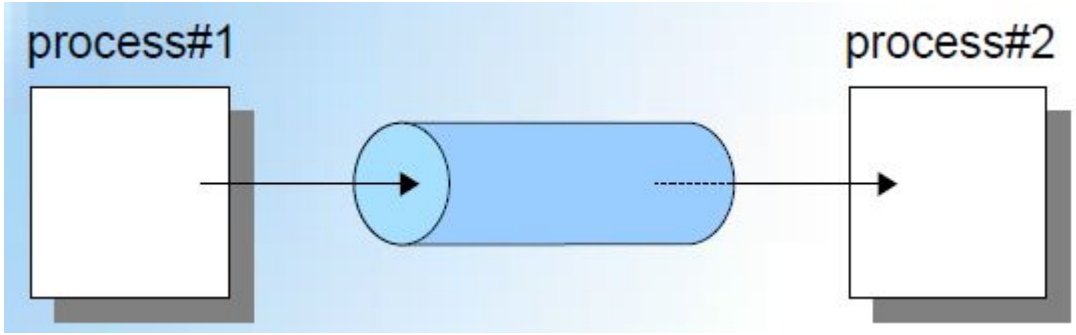
Primitives C de gestion des signaux :

kill() : envoie un signal vers un processus

sigaction() : spécifie une action associée à un signal

Echanges et synchro. par tubes (1)

- Un **tube** est un **flux de données** permettant l'échange unidirectionnel de données entre deux processus s'exécutant sur une même machine



- Les processus souhaitant communiquer doivent être de la même famille (ex. père/fils)

- Les tubes peuvent être considérés comme des fichiers ouverts (sans l'inconvénient des accès coûteux au système de fichier du disque)

Echanges et synchro. par tubes (2)

- Utilisation simple en **langage shell** (opérateur '|')

```
ls | more
```

Primitives C pour la gestion des tubes :

`pipe()` : création d'un tube (renvoie une paire de descripteurs du fichier)

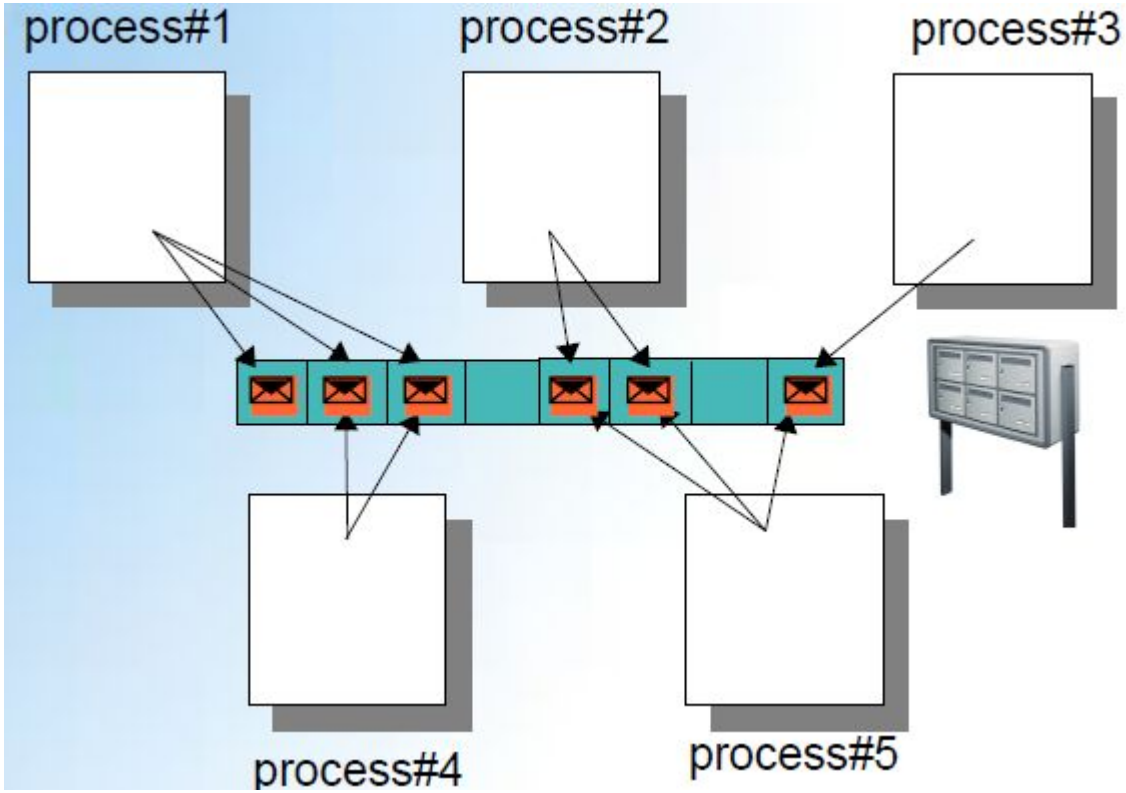
`write()` : écriture dans un tube

`read()` : lecture du contenu d'un tube

`close()` : suppression d'un tube

Echanges et synchro. par files de messages (1)

- Les files de messages permettent une **communication indirecte** entre processus (mécanisme de « **boîte aux lettres** »)



Echanges et synchro. par files de messages (2)

- Deux aspects sont à gérer :
  - **la communication**
    - identification unique du destinataire et de l'expéditeur du message
    - description exacte des données
    - description de l'adressage
  - **la synchronisation**
    - envoi bloquant ou non
    - réception bloquante ou non
    - priorité des messages

  - Un message ne peut être lu qu'une seule fois

Echanges et synchro. par files de messages (3)

- Primitives C de gestion des files de messages (type IPC System V) :
  - `msgget()`
    création d'une nouvelle file de messages
    accès à une file de messages existante
  - `msgsend()`
    envoi d'un message dans une file
  - `msgctl()`
    contrôle d'une file (suppression, permissions, etc.)

Echanges et synchro. par files de messages (4)

- Primitives C de gestion des files de messages (type POSIX) :
  - `mq_open()`
    création d'une nouvelle file de messages
    accès à une file de messages existante
  - `mq_close()`
    fermeture d'une file de messages (sans la détruire)
  - `mq_unlink()`
    destruction d'une file de messages
  - `mq_send()` / `mq_timedsend()`
    envoi d'un message dans une file
  - `mq_receive()` / `mq_timedreceive()`
    récupération d'un message dans une file
  - `mq_getattr()`
    gestion des attributs d'une file de messages

Conclusion sur les mécanismes IPC

- Mécanismes divers et variés répondant plus ou moins bien aux problèmes de communication inter-processus

- Charge au développeur de choisir quels mécanismes utiliser en fonction :
  - des besoins de l'application
  - du coût de développement

- **Complexité** du développement vs. **performance** du programme

End of Chapter 6