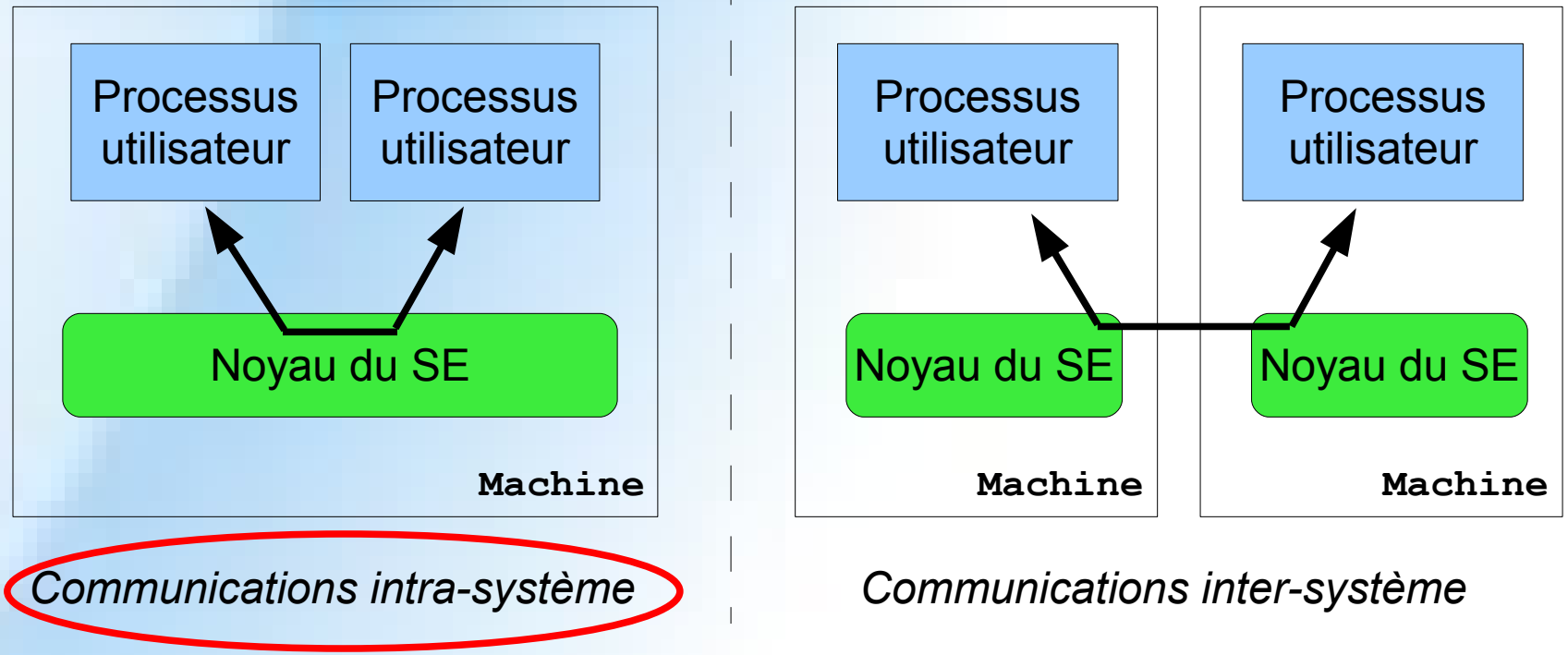
# Plan du cours

- Introduction aux systèmes d'exploitation
- Présentation générale d'UNIX
- Programmation shell
- Processus et parallélisme
- Ordonnancement
- **Communication et synchronisation**
- Gestion de la mémoire
- Gestion de E/S

# Contexte (1)

- Processus **concurrents** vs. Processus **distants**



# 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

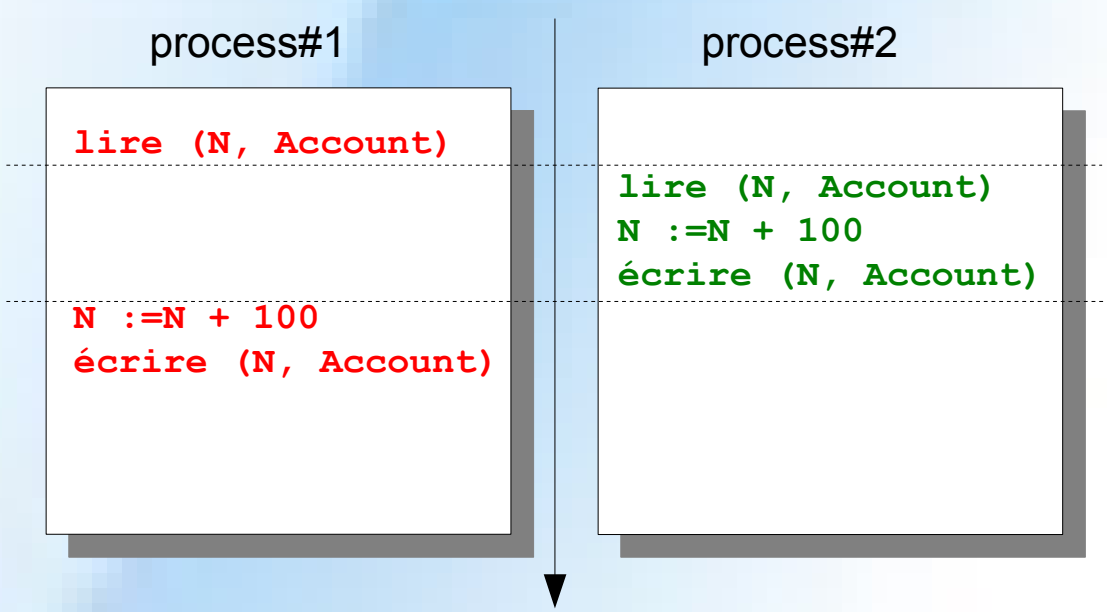
- On parle aussi de **situation de compétition** (*race condition*) pour décrire une situation dont l'issue dépend de l'ordre dans lequel les opérations sont effectuées

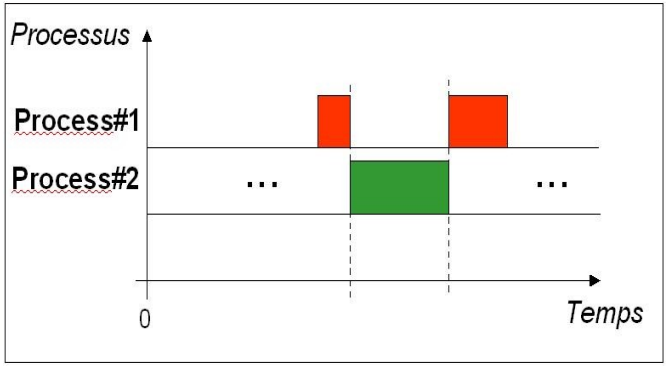
- Une **section critique** est une section de programme manipulant une ressource critique

- 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#2 ne doit pas accéder à N tant que process#1 l'utilise !

# Accès aux ressources critiques (2)

- Solution par exclusion mutuelle :

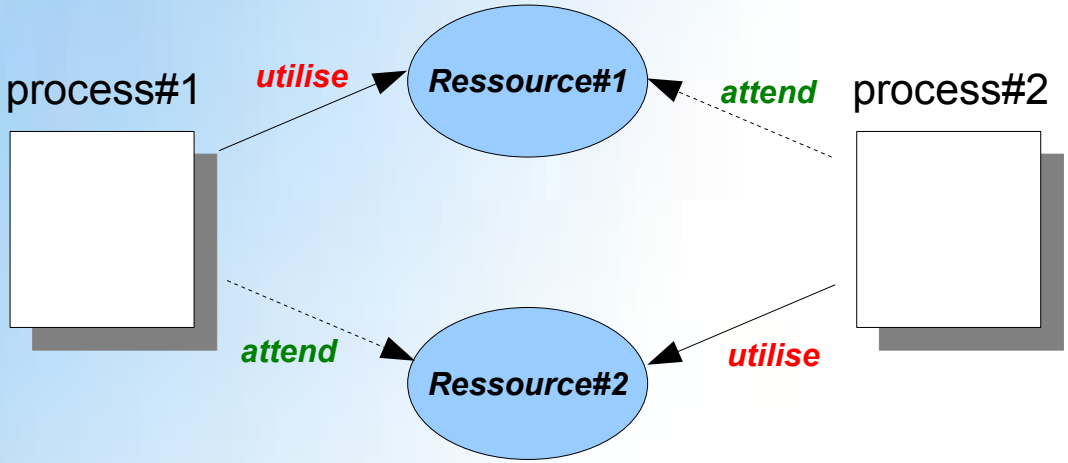
```
Variable Compte  : entier
```

```
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)

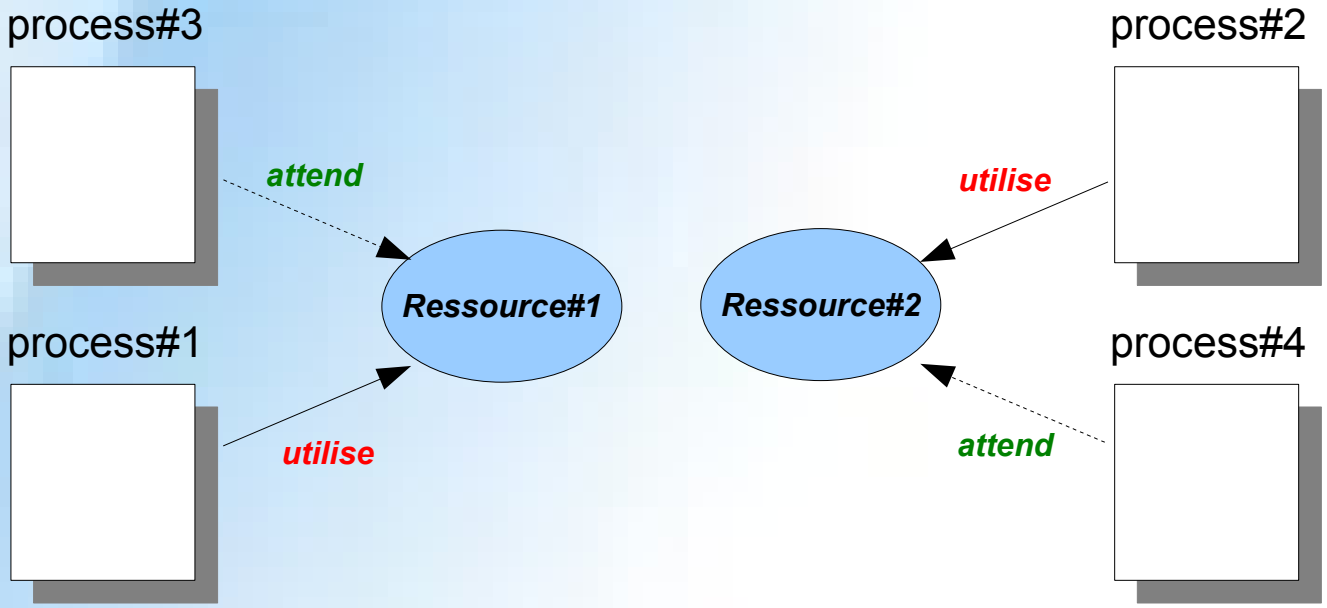
- Cas d'**interblocage** : ensemble de processus attendant chacun une ressource déjà possédée par un processus de l'ensemble




- 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 (1)

- Un ensemble de processus, divisés en deux catégories, partagent une zone mémoire

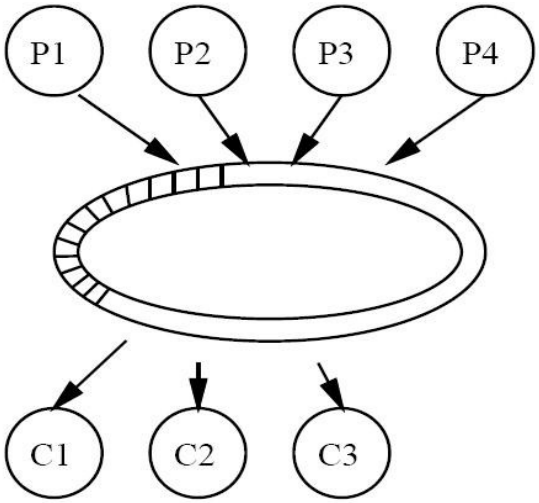
- Les premiers (**producteurs**) remplissent la mémoire partagée avec des éléments

- Les seconds (**consommateurs**) utilisent ces éléments et les retirent de la mémoire

- Exemple : la file d'impression

# Problème des producteurs/consommateurs (2)

Structure du système producteurs/consommateurs :




- Le Producteur ne dépose un objet que si le tampon il existe une case vide dans T.
- Le Consommateur ne prélève un objet que si il existe une case plein dans le tampon.
- Le Consommateur n'essaie pas de consommer un objet qui est en train d'être déposé par le Producteur (Pour éviter l'accès à la même case par le Producteur et le consommateur).
- Si le producteur est en attente parce que le tampon est plein, il doit être averti dès que cette condition cesse d'être vraie (une case devient vide).
- Si le consommateur est en attente parce que le tampon est vide, il doit être averti dès que cette condition cesse d'être vraie (une case devient plein).

# Problème des lecteurs/rédacteurs (1)

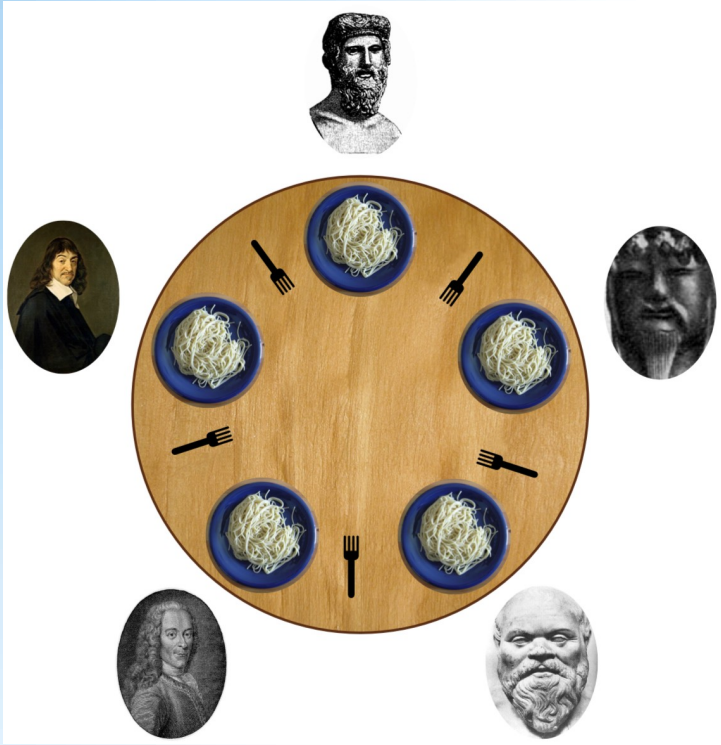
- Un ensemble de processus, divisés en deux catégories, partagent une zone mémoire
- Certains processus (**les lecteurs**) font des accès en lecture seule à cette zone
- D'autres processus (**les rédacteurs**) modifient le contenu de cette zone
- Les rédacteurs sont parfois appelés écrivains

# 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 (1)

- 5 philosophes passent leur vie à manger et à penser
- Pour manger, ils ont besoin de 2 fourchettes mais il n'y a que 5 fourchettes



# Problème des philosophes (2)

- **Principe** :
  - 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

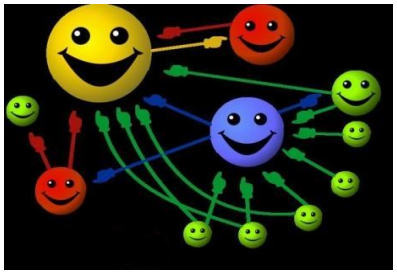
# Résolution des problématiques

- Comment gérer les accès concurrents aux ressources ?

⇒ Introduction de **nouveaux mécanismes**

**- de communication**

**- de synchronisation**



# Communications inter-processus

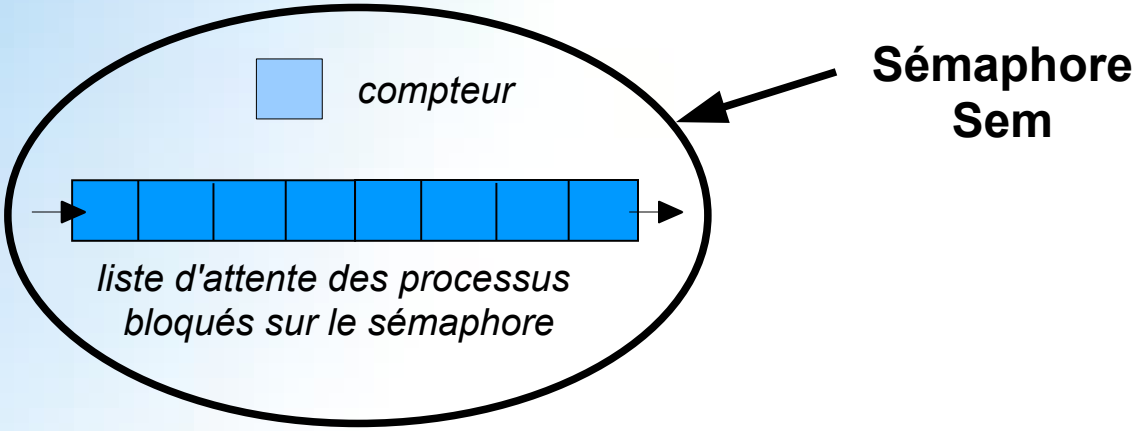
- **I**nter-**P**rocess **C**ommunication (**IPC**) : méthodes permettant à plusieurs processus de communiquer entre eux

- 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

# Synchronisation par sémaphore (1)

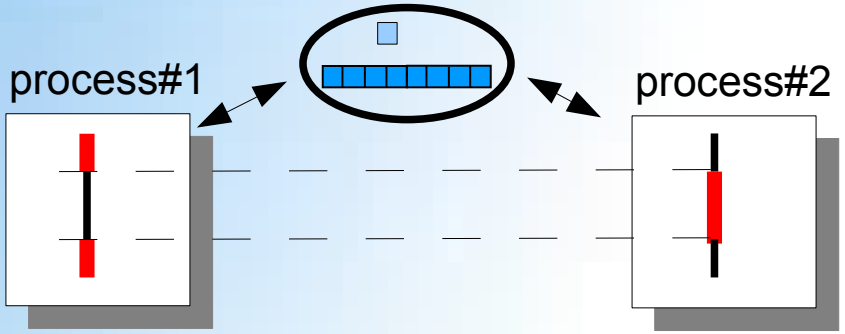
- Les sémaphores sont utilisés dans le **contrôle d'accès à une ressource**
- Un sémaphore est une **structure de données**
  - contenant un compteur (valeur entière non négative)
  - gérant une file d'attente de processus attendant qu'advienne une condition particulière propre au sémaphore



# Synchronisation par sémaphore (2)

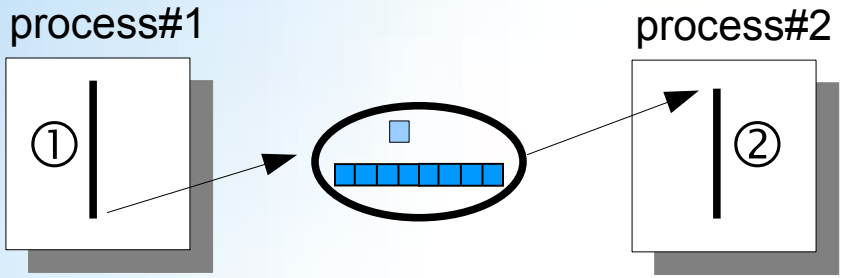
- Types de synchronisation possibles :

- **Exclusion mutuelle**




Section critique

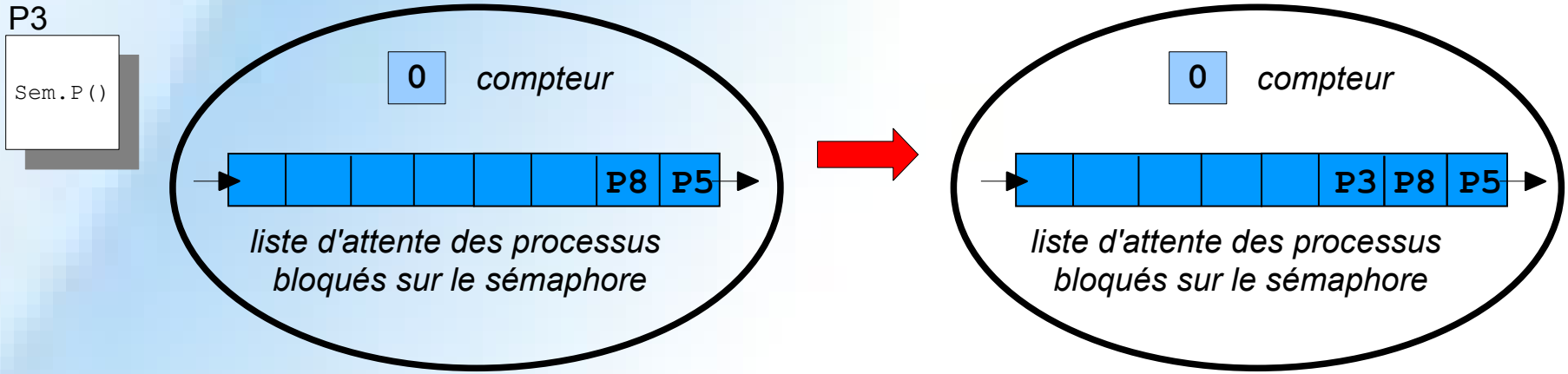
- **Barrière de synchronisation**



# Synchronisation par sémaphore (3)

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

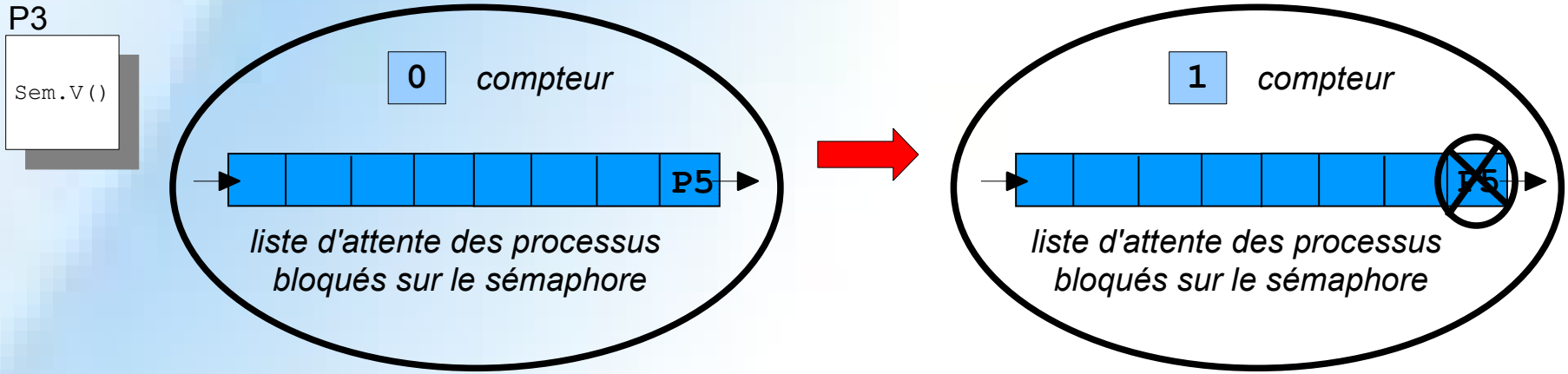
```
Sem.P() : si (Sem.compteur>0)
                alors Sem.compteur = Sem.compteur-1
                sinon insère_ce_processus(Sem.file)
            finsi
```



# 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)

- Types de sémaphores :
  - **Sémaphore binaire** : implémentation de l'exclusion mutuelle
    - le compteur ne peut admettre que les valeurs 0 ou 1
    - le compteur doit être initialisé à 1

  - **Sémaphore compteur** : implémentation de la barrière de synchronisation

| Proc1 | Proc2 | Proc3 | Proc4 | État de S | temps |
|---|---|---|---|---|---|
| Init(S,2) | | | | K= L = | |
| P(S) | | | | | |
| | P(S) | | | | |
| | | P(S) | | | |
| | V(S) | | | | |
| | | | P(S) | | |
| V(S) | | | | | |
| P(S) | | | | | |
| | P(S) | | | | |
| | | V(S) | | | |

# Résolution du problème des producteurs/consommateurs

On introduit donc deux variables (de type sémaphore) caractérisant l'état du tampon :
- NPLEIN : nombre des cases pleins (contiens des objets) dans le tampon (début : 0) ;
- NVIDE : nombre des cases vides disponibles dans le tampon (N au début);

**Déclaration et initialisation**

Semaphore NPLEIN, NVIDE;
init(NPLEIN , 0); //// car il existe 0 cases plein au début.
init(NVIDE , N); // car il existe N cases vides au début.
Objet T [N] ; // tableau contient des objets;

| Processus Producteur ( ) | Processus Consommateur ( ) |
|---|---|
| { Objet obj ; Repeter { **Obj = Produire_un_objet ();** **P(NVIDE ) ;** **Deposer(Obj, T);** **V(NPLEIN ) ;** }tant que vrai ; } | { Objet obj ; Repeter { **P(NPLEIN ) ;** ***Obj* = Prélever _un_objet (T);** **V(NVIDE ) ;** Consommer_un_objet *(Obj);* }tant que vrai ; } |

# Résolution du problème des lecteurs/rédacteurs

## Lecteurs - Rédacteurs

```
int NbL = 0;
Semaphore redact=1, mutex=1;
```

```
Redacteur( )
{   while(1) {
        P(redact);
        ecrire();
        V(redact);
    }
}
```


```
Lecteur() {
    while(1) {
        P(mutex);
        if (NbL == 0)  P(redact);
        NbL++;
        V(mutex);
        lire();
        P(mutex);
        NbL--;
        if(NbL == 0) V(redact);
        V(mutex);
    }
}
```

Risque de famine pour les rédacteurs

# Résolution du problème Problème des philosophes

```
class Philosophe extends Thread { // —— Premiere solution (Java)
  static Semaphore[ ] s = new Semaphore[5];
  Philosophe (int x) { i = x; }
  int i;

  public void run() {
    while (true) {
      // penser
      Semaphore.P(s[i]);
      Semaphore.P(s[(i+1)%5]);
      // manger
      Semaphore.V(s[i]);
      Semaphore.V(s[(i+1)%5]);
  } }

  public static void main (String[ ] args) {
    for (int i = 0; i < s.length; ++i) {
      Philosophe phi = new Philosophe(i);
      phi.start();
} } }
```

Sureté, mais interblocage.

# Résolution du problème Problème des philosophes

```
class Philosophe1 extends Thread { // ———————— Deuxième solution
  static int[ ] f = {2, 2, 2, 2, 2};
  static ConditionPosix[ ] manger = new ConditionPosix[5];
  int i;

  Philosophe1 (int x) { i = x; }

  ...

  public static void main (String[ ] args) {
    for (int i = 0; i < f.length; ++i) {
      Philosophe1 phi = new Philosophe1(i);
      phi.start();
} } }
```

- $f[i]$ est le nombre de fourchettes disponibles pour $\Phi_i$

# Résolution du problème Problème des philosophes

```
static synchronized void prendreFourchettes(int i) {
  while (f[i] != 2)
    waitPosix (manger[i]);
  -- f[(i-1) % 5]; -- f[(i+1) % 5];
}

static synchronized void relacherFourchettes(int i) {
  int g = (i-1) % 5, d = (i+1) % 5;
  ++ f[g]; ++ f[d];
  if (f[d] == 2)
    notifyPosix (manger[d]);
  if (f[g] == 2)
    notifyPosix (manger[g]);
}

public void run() {
  while (true) {
    // penser
    prendreFourchettes(i);
    // manger
    relacherFourchettes(i);
  } }
```

famine, si, par exemple, les philosophes 1 et 3 complotent contre le philosophe 2, qui

# 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