IMT Mines Alès
École Mines-Télécom

Informatique concurrente et répartie

La communication inter-processus

Emmanuel Romagnoli

Plan

- La problématique de la communication inter-processus
- L'implémentation bas-niveau de l'exclusion mutuelle
- Les méthodes par attente active
- Les sémaphores
- Les moniteurs

La problématique de la communication inter-processus

Les SE multitâches

Les systèmes d'exploitation multitâches peuvent gérer plusieurs processus qui peuvent contenir un ou plusieurs threads.

Ces processus (lourds et légers) se partagent les ressources physiques de la machine qui les accueille (les disques durs, le processeur, la mémoire...).

L'accès concurrent au processeur est géré par le système d'exploitation au moyen d'un mécanisme d'exclusion temporel (un processus dispose pendant un quantum de temps de la machine de façon exclusive).

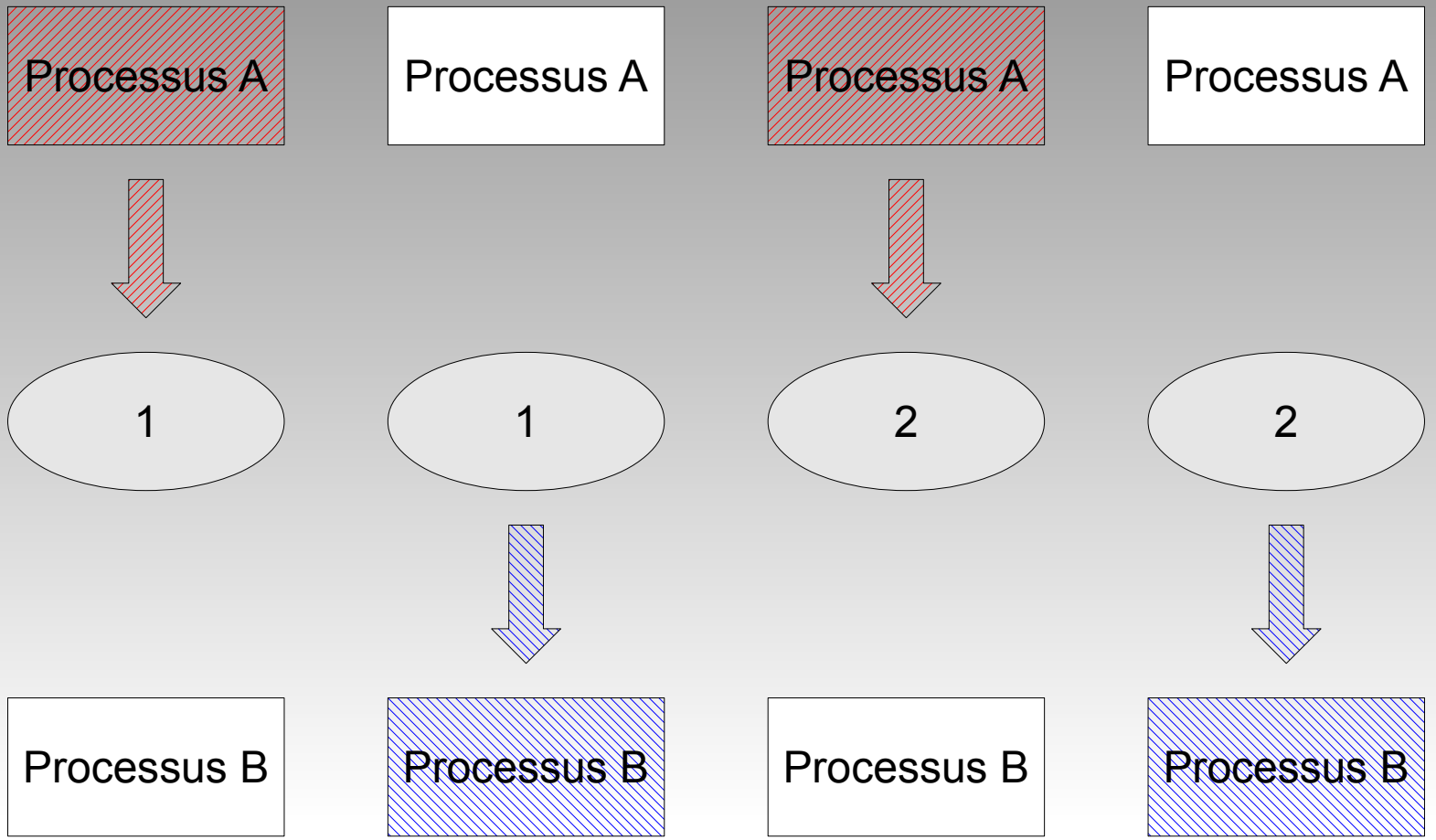
Les ressources communes

Ce mécanisme devient insuffisant si deux processus utilisent une ressource commune comme un fichier (un tube par exemple) ou une zone de mémoire commune, car nous ne pouvons pas faire d'hypothèse sur l'état d'avancement d'un processus dans le traitement de cette ressource.

En d'autres termes, nous ne sommes pas sûrs qu'un processus A ait terminé d'utiliser une ressource commune lorsqu'un processus B a la main et modifie cette même ressource.

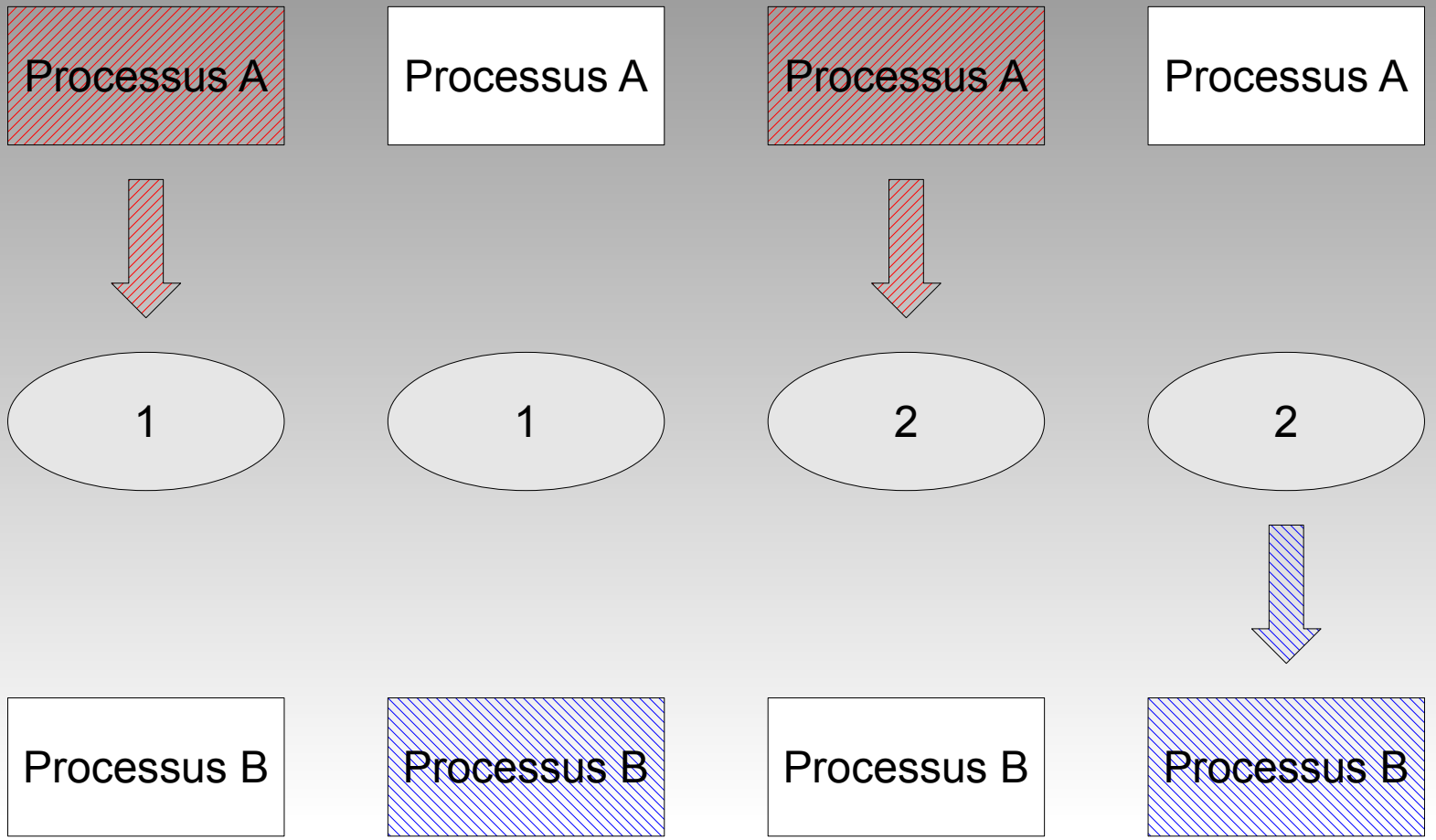
Le cas idéal

A écrit « 1 » dans la ressource commune. B lit « 1 » et fait un calcul. A écrit « 2 », B lit « 2 » et refait un calcul.



Un cas possible dans la réalité

A écrit « 1 » dans la ressource commune. B n'a pas le temps de lire. A écrit « 2 », B lit « 2 » et a manqué le « 1 ».



L'exclusion mutuelle

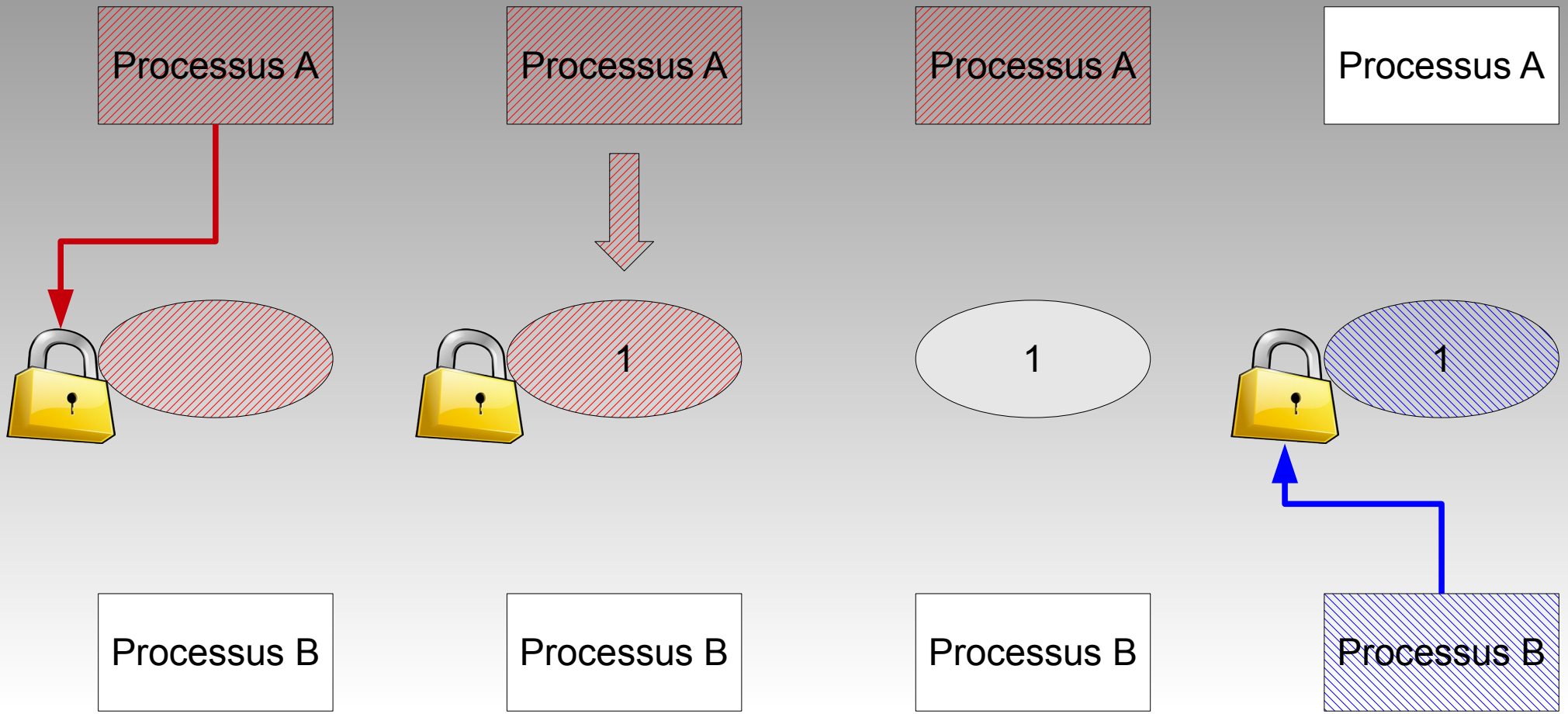
Pour éviter ce problème, le SE doit fournir un mécanisme qui garantit qu'une ressource commune à plusieurs processus, soit réservée à l'un d'entre eux pendant une certaine période.

Un tel mécanisme est appelé **mécanisme d'exclusion mutuelle** et il s'applique à une ressource commune que nous qualifions de **ressource critique**.

La portion de code qui utilise le mécanisme d'exclusion mutuelle pour accéder à cette ressource critique est appelée **section critique**.

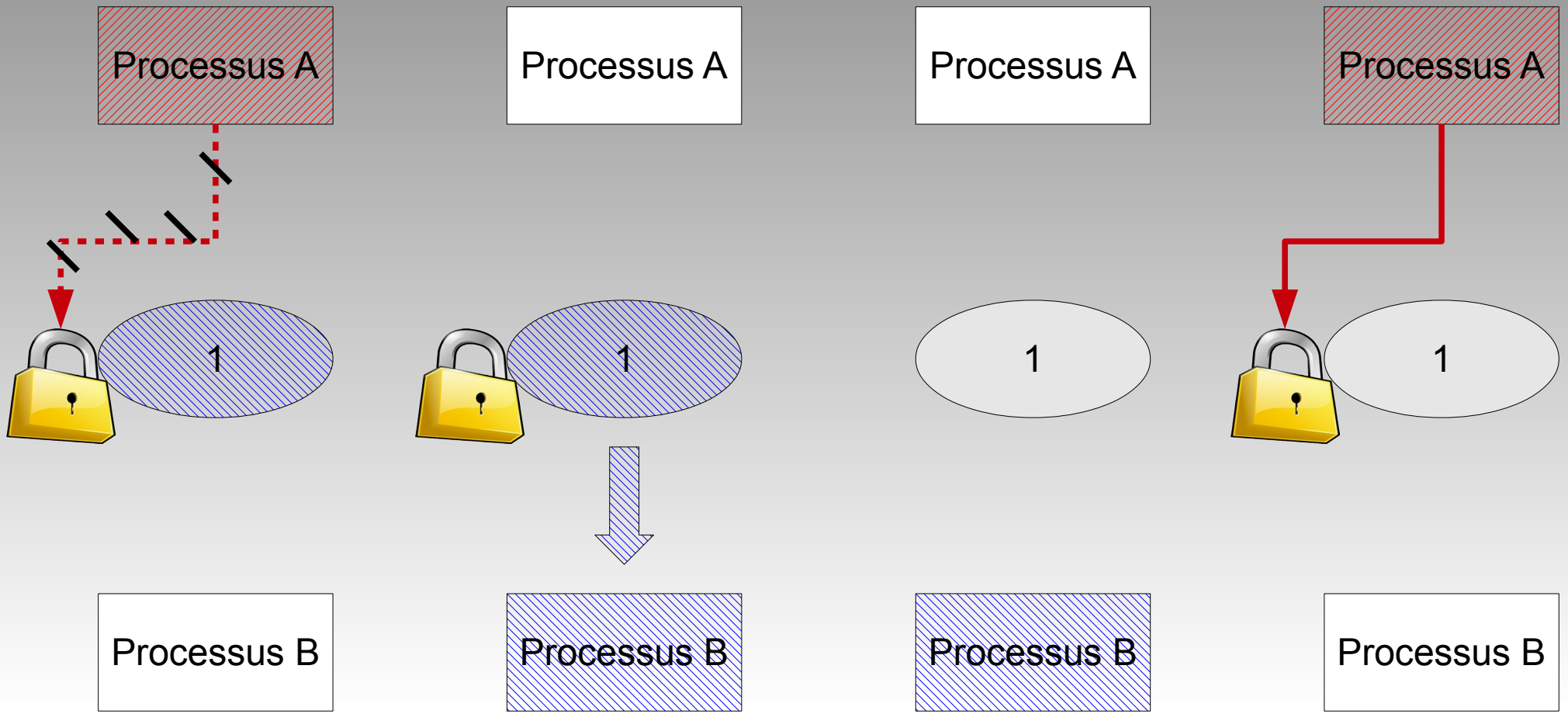
Exemple d'utilisation

A commence par placer un verrou puis écrit « 1 » et retire le verrou. B a la main : il place un verrou, mais perd la main.



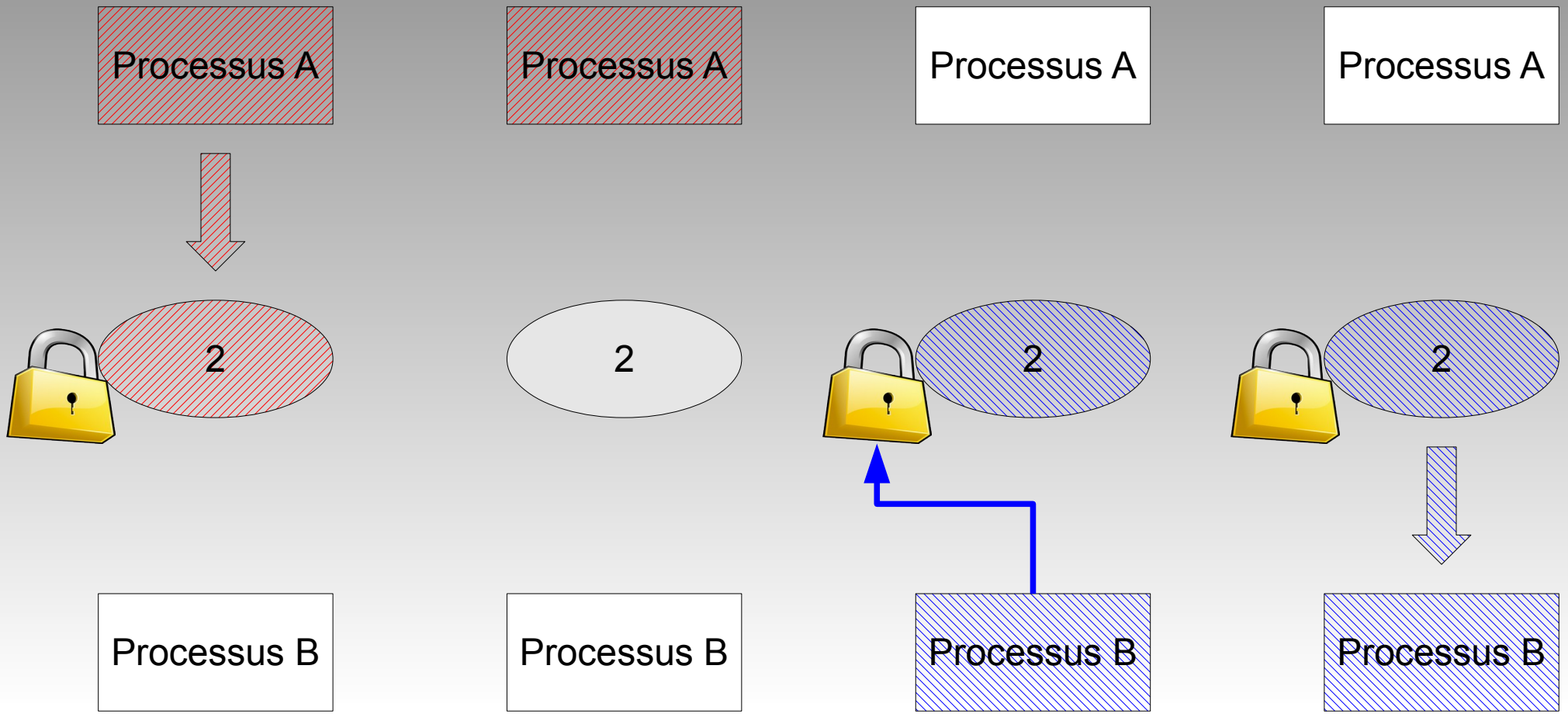
Exemple d'utilisation

A essaye de placer un verrou : il ne peut pas, il est mis en attente. B reprend la main et peut terminer son traitement.



Exemple d'utilisation

B retire a retiré son verrou donc A a pu placer le sien. Il peut maintenant écrire « 2 » qui sera lu plus tard par B.



L'implantation bas-niveau de l'exclusion mutuelle

Le masquage d'interruption

La modification de l'état des verrous est des opérations **atomiques**, autrement dit, elle **ne peut pas être interrompue**.

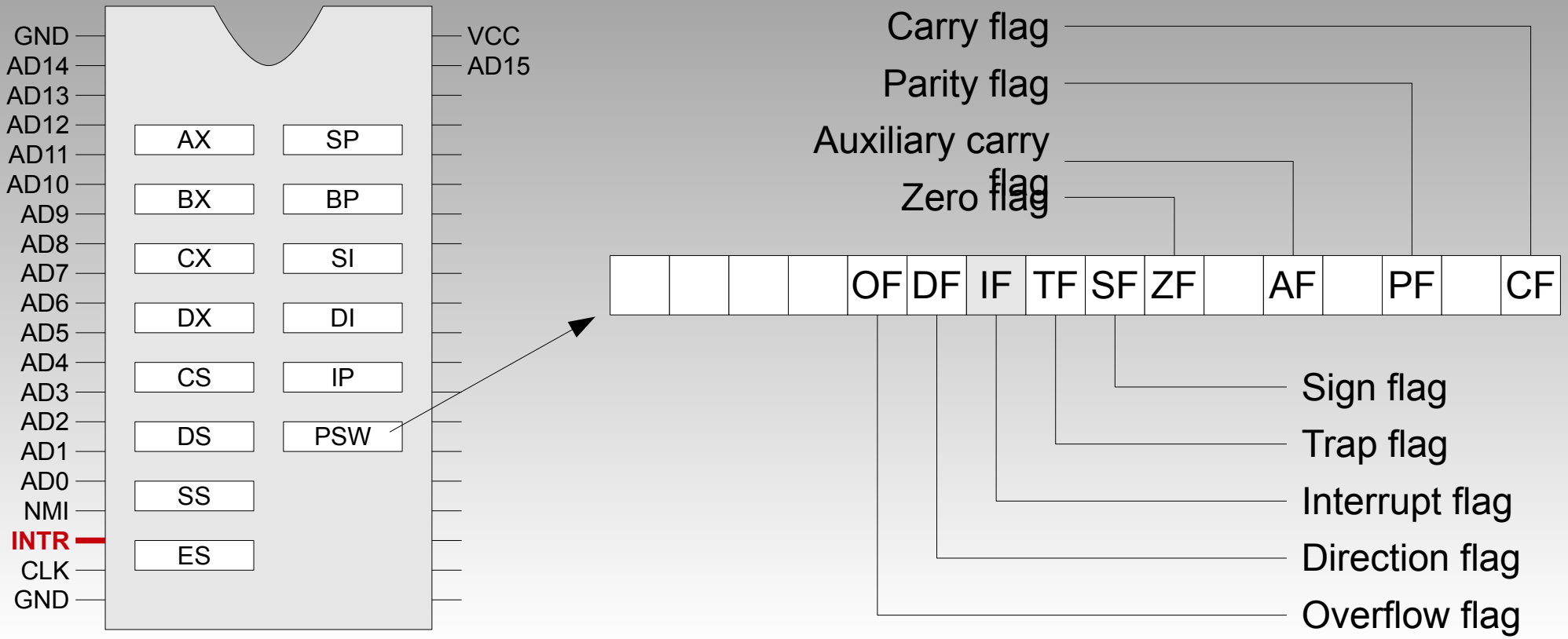
Pour garantir cette propriété, le processus qui effectue ce type d'opération **ne doit pas perdre la main**.

Pour cela, on va bloquer le mécanisme de commutation des processus en inhibant le signal d'interruption grâce à une instruction assembleur **CLI** (CLear Interrupt flag).

On parle alors de **masquage d'interruption**.

Le masquage d'interruption

Le bit IF du processeur (par exemple, le 8086) est mis à « 0 », les signaux (en particulier le signal d'horloge) arrivant sur la patte INT ne sont donc pas considérés.



PWS signifie « Processor Status Word »

Le masquage d'interruption

Le bit IF peut être remis à « 1 » grâce à l'instruction « STI ». Si on suppose que l'état du verrou est stocké dans la position mémoire 110h du segment de données (le tas), le code de sa modification pourrait être celui-ci dessous.

```
CLI
mov b.[100h], 1
STI
```

Remarque : l'utilisation de CLI/STI est risquée, car si on oublie de faire un STI et que le processus part dans une boucle infinie, il n'y a plus moyen de récupérer la main (il faut rebooter).

Le masquage d'interruption

Ce petit code en assembleur peut être encapsulé dans une fonction en C :

void setVerrou (int v)

On va maintenant s'intéresser à l'utilisation de ce verrou dans des programmes de plus haut niveau.

Voir http://beuss.developpez.com/tutoriels/pcasm/
http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html

Les méthodes par attente active

L'alternance

L'exclusion mutuelle peut reposer sur une variable commune utilisée pour sélectionner le processus qui a le droit d'entrer en section critique.

```
setTour (0); // fonction atomique

while (1)
 {
  while (tour != 0); // attente
  sectionCritique ();
  setTour (1);
  sectionNonCritique ();
 }
```

Processus n°0

```
while (1)
 {
  while (tour != 1); // attente
  sectionCritique ();
  setTour (0);
  sectionNonCritique ();
 }
```

Processus n°1

L'alternance

Cette méthode a des inconvénients :

- Il s'agit d'une méthode par attente active donc si le processus ne peut pas entrer en section critique, il se contente de tester la valeur de tour (boucle while) et les cycles machine sont gaspillés
- Le processus le plus rapide cale sa vitesse sur le processus fonctionnement le moins vite
- Cette méthode impose une stricte alternance pour entrer dans la section critique.

Tester l'intention de l'autre – Le principe

Un autre mécanisme consiste à tester l'intention des autres processus avant de faire part de sa propre intention d'entrer en section critique (et de bloquer les autres processus).

```
int interet0=0;
int interet1=0;

while (1)
 {
  while (interet1 == 1); //attente
  setInteret0 (1);
  sectionCritique ();
  setInteret0 (0);
  sectionNonCritique ();
 }
```

Processus n°0

```
while (1)
 {
  while (interet0 == 1);
  setInteret1 (1);
  sectionCritique ();
  setInteret1 (0);
  sectionNonCritique ();
 }
```

Processus n°1

Tester l'intention de l'autre – Problème 1

Les 2 processus peuvent se retrouver en section critique s'ils perdent la main après la seconde boucle while (ils passent le test sur l'intérêt de l'autre).

```
int interet0=0;
int interet1=0;

while (1)
 {
  while (interet1 == 1); //attente
  setInteret0 (1);
  sectionCritique ();
  setInteret0 (0);
  sectionNonCritique ();
 }
```

Processus n°0

```
while (1)
 {
  while (interet0 == 1);
  setInteret1 (1);
  sectionCritique ();
  setInteret1 (0);
  sectionNonCritique ();
 }
```

Processus n°1

Tester l'intention de l'autre – Problème 2

On inverse deux lignes pour résoudre le problème 1, mais cette fois-ci il peut provoquer des interblocages entre les 2 processus qui veulent entrer en section critique.

```
int interet0=0;
int interet1=0;

while (1)
 {
  setInteret0 (1);
  while (interet1 == 1); //attente
  sectionCritique ();
  setInteret0 (0);
  sectionNonCritique ();
 }
```

Processus n°0

```
while (1)
 {
  setInteret1 (1);
  while (interet0 == 1);
  sectionCritique ();
  setInteret1 (0);
  sectionNonCritique ();
 }
```

Processus n°1

Modification de la boucle while

Si le processus détecte que son voisin souhaite entrer en section critique, il se désiste pendant un court laps de temps puis réaffirme sa volonté d'entrer en section critique.

```
int interet0=0;
int interet1=0;

while (1)
 {
  setInteret0 (1);
  while (interet1 == 1)
   {
    setInteret0 (0);
    sleep (rand());
    setInteret0 (1);
   }
  sectionCritique ();
  setInteret0 (0);
  sectionNonCritique ();
 }
```

```
while (1)
 {
  setInteret1 (1);
  while (interet0 == 1)
   {
    setInteret1 (0);
    sleep (rand());
    setInteret1 (1);
   }
  sectionCritique ();
  setInteret1 (0);
  sectionNonCritique ();
 }
```

La solution de Theodorus Dekker (1965)

On remplace le sleep avec un temps aléatoire par un test sur le tour. Il pourra alors reformuler son souhait quand son tour sera venu.

```
int interet0=0; int interet1=0;int tour=0;

while (1)
 {
  setInteret0 (1);
  while (interet1 == 1)
   {
    setInteret0 (0);
    while (tour != 0);
    setInteret0 (1);
   }
  sectionCritique ();
  setTour (1);
  setInteret0 (0);
  sectionNonCritique ();
 }
```

```
while (1)
 {
  setInteret1 (1);
  while (interet0 == 1)
   {
    setInteret1 (0);
    while (tour != 1);
    setInteret1 (1);
   }
  sectionCritique ();
  setTour (0);
  setInteret1 (0);
  sectionNonCritique ();
 }
```

Voir http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html

La solution de Gary L. Peterson (1981)

On fusionne les tests sur les valeurs de l'intérêt et du tour. Comme pour les deux algorithmes précédents, celui-ci peut être généralisé à plus de 2 processus.

```
int interet0=0;
int interet1=0;
int tour=0;

while (1)
 {
  setInteret0 (1);
  while (interet1 == 1 &&
         tour     != 0);
  sectionCritique ();
  setTour (1);
  setInteret0 (0);
  sectionNonCritique ();
 }
```

```
while (1)
 {
  setInteret1 (1);
  while (interet0 == 1 &&
         tour     != 1);
  sectionCritique ();
  setTour (0);
  setInteret1 (0);
  sectionNonCritique ();
 }
```

Voir Gary L. Peterson - Journal: Information Processing Letters - IPL , vol. 12, no. 3, pp. 115-116, 1981

Synthèse

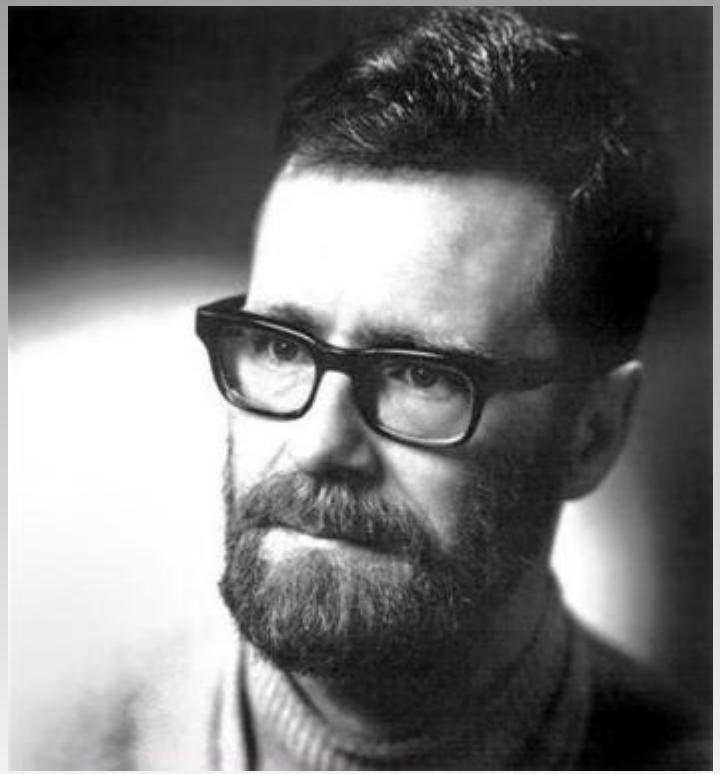
Les méthodes par attente active fonctionnent, mais sont à éviter :

- **gaspillage** de la puissance du CPU ;
- **risque d'interblocage** entre un processus de priorité basse qui est entré en section critique et un processus de priorité haute qui cherche à entrer en section critique après avoir été débloqué (après avoir obtenu la ressource dont il avait besoin).

Les sémaphores

Le principe

Le mécanisme des sémaphores a été proposé par Edsger Dijkstra en 1965.



Voir http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF

Le principe

Le sémaphore est une structure de synchronisation qui se compose des éléments suivants :

- un entier « compteur » ;
- une file d'attente de type FIFO contenant des références vers les processus en attente de section critique ;
- deux opérations atomiques notées P et V qui permettent de manipuler le sémaphore ;
- une opération atomique de création du sémaphore qui prend en paramètre la valeur de départ.

La primitive P

P est la première lettre du terme hollandais Passeren (attendre). Cette primitive décrémente le compteur et si sa valeur est strictement inférieure à 0, il **bloque** le processus courant.

```
compteur ← compteur-1
si (compteur < 0)
  alors bloquer le processus courant
        ajouter le PID à la file d'attente
finsi
```

Voir http://shtroumbiniouf.free.fr/CoursInfo/Systeme2/Cours/Parallelisme/Semaphores/Semaphores.html

La primitive V

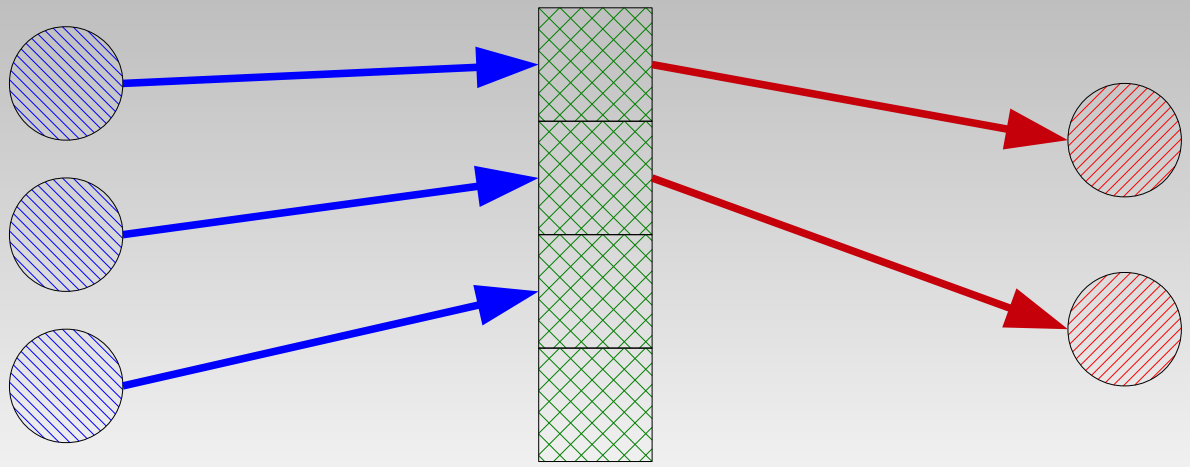
V est la première lettre du terme hollandais Vrijmaken (signaler). Cette primitive incrémente le compteur et si sa valeur est supérieure ou égale à 0, il **débloque** le premier processus de la file FIFO.

```
compteur ← compteur+1
si (compteur >= 0)
  alors débloquer le premier processus de la file
        retirer ce processus de la file
finsi
```

Voir http://shtroumbiniouf.free.fr/CoursInfo/Systeme2/Cours/Parallelisme/Semaphores/Semaphores.html

Le producteur-consommateur

Un ou plusieurs processus producteurs déposent des éléments dans un tampon de **taille finie** (N emplacements). Un ou plusieurs processus consommateurs qui retirent ces éléments. Le tampon peut être un fichier, un tube, une zone de mémoire commune.



Voir http://shtroumbiniouf.free.fr/CoursInfo/Systeme2/Cours/Parallelisme/Semaphores/Semaphores.html

Le producteur-consommateur

Le modèle doit fonctionner en respectant les contraintes suivantes :

- le producteur doit s'endormir/se bloquer quand le tampon est plein ;
- le consommateur doit réveiller/débloquer le producteur quand le tampon n'est plus plein ;
- le consommateur doit s'endormir/se bloquer quand le tampon est vide ;
- le producteur doit réveiller/débloquer le consommateur quand le tampon n'est plus vide.

Le producteur-consommateur

Voici un « pseudo-code » du producteur-consommateur.

```
Semaphore mutex   (1);
Semaphore plein   (0);
Semaphore vide    (N);
Tampon    tampon  (N);

// producteur

while (1)
 {
  Obj o = produire();
  vide.P();
  mutex.P()
  tampon.placer (o);
  mutex.V();
  plein.V();
 }
```

```
// consommateur

while (1)
 {
  plein.P();
  mutex.P();
  Obj o = tampon.prendre();
  mutex.V();
  vide.V();
  consommer (o);
 }
```

Les lecteurs/rédacteurs

Un système de gestion d'information doit respecter quelques règles de fonctionnement afin de ne pas retourner de résultats incohérents :

- Plusieurs lecteurs peuvent consulter la base de données en même temps : accès concurrent en lecture.
- Un seul rédacteur a le droit d'écrire dans la base de données : accès exclusif en écriture.
- Lorsqu'une lecteur intervient, aucune écriture n'est possible et inversement : accès en lecture et en écriture en exclusion mutuelle.

Les lecteurs/rédacteurs

En 1971, P.J. Courtois, F. Heymans et D.L. Parnas ont proposé un article consacré au problème des lecteurs/rédacteurs. Ils proposent 3 variantes :

- la première donne la priorité aux lecteurs;
- la seconde donne la priorité aux rédacteurs;
- la troisième accorde la même priorité aux deux protagonistes.

Voir http://cs.nyu.edu/~lerner/spring10/MCP-S10-Read04-ReadersWriters.pdf

Les lecteurs/rédacteurs – Version 1

Le codage de la variante « lecteur prioritaire » utilise :

- nbLecteurs : une variable qui comptabilise le nombre de processus qui lisant simultanément le SI ;
- mutexLecteur : un sémaphore qui assure qu'un seul lecteur modifie la variable nbLecteur ;
- mutexRedacteur : un sémaphore qui garantit qu'au plus un rédacteur modifie le SI ;
- mutexModeAccess : un sémaphore qui permet de garantir l'exclusion des modes d'accès au SI.

Les lecteurs/rédacteurs – Version 1

```
Entier    nbLecteurs      (0)
Semaphore mutexLecteur    (1);
Semaphore mutexRedacteur  (1);
Semaphore mutexModeAcces  (1);

void demandeDeLecture ()
 {
  mutexLecteur.P();          /* bloque les autres lecteurs */
   nbLecteurs++;
   if (nbLecteurs == 1)      /* premier lecteur bloque */
    mutexModeAcces.P();      /* le système en mode lecture */
  mutexLecteur.V();          /* débloque les autres lecteurs */
 }

void finDeLecture ()
 {
  mutexLecteur.P();          /* bloque les autres lecteurs */
   nbLecteurs--;
   if (nbLecteurs == 0)      /* dernier lecteur débloque */
    mutexModeAcces.V();      /* le système du mode lecture */
  mutexLecteur.V();          /* débloque les autres lecteurs */
 }
```

Les lecteurs/rédacteurs – Version 1

```
void demandeDEcriture ()
 {
  mutexRedacteur.P();     /* bloque les autres rédacteurs */
  mutexModeAcces.P();     /* bloque en mode écriture */
 }

void finDEcriture ()
 {
  mutexModeAcces.V();     /* débloque le mode écriture */
  mutexRedacteur.V();     /* débloque les autres rédacteurs */
 }
```

```
void lecteur ()
 {
  while (1)
   {
    demandeDeLecture ();
    lectureDesDonnées ();
    finDeLecture ();
   }
 }
```

```
void redacteur ()
 {
  while (1)
   {
    demandeDEcriture ();
    ecritureDesDonnées ();
    finDEcriture ();
   }
 }
```

Les lecteurs/rédacteurs – Version 2

La variante « rédacteur prioritaire » utilise :

- nbLecteurs : une variable qui comptabilise le nombre de processus qui lisent le SI en mode concurrent
- nbRedacteurs : une variable qui comptabilise le nombre de processus qui écrivent dans le SI en mode exclusif.
- mutexLecteur : pour avoir un seul lecteur en face d'un rédacteur
- mutexRedacteur : pour un accès exclusif à nbRedacteurs
- mutex : pour un accès exclusif à nbLecteurs
- recteur et rédacteur : pour bloquer le système dans un mode de fonctionnement

Les lecteurs/rédacteurs – Version 2

```
Entier    nbLecteurs      (0);
Entier    nbRedacteur     (0);
semaphore mutexLecteur    (1);
semaphore mutexRedacteur  (1);
semaphore redacteur       (1);
semaphore lecteur         (1);
semaphore mutex           (1);

void demandeDeLecture ()
 {
  mutex.P();              /* bloque les autres lecteurs qui veulent lire*/
   lecteur.P();           /* essaye d'entrer en mode lecture */
    mutexLecteur.P();     /* bloque les autres lecteurs */
     nbLecteurs++;
     if (nbLecteurs == 1) /* si c'est le premier lecteur */
      redacteur.P();      /* interdir le mode écriture */
    mutexLecteur.V();     /* débloque les autres lecteurs */
   lecteur.V();           /* débloque le mode lecture */
  mutex.V();              /* debloque les autres lecteurs qui veulent lire */
 }

void finDeLecture ()
 {
  mutexLecteur.P();       /* bloque les autres lecteurs */
   nbLecteurs--;
   if (nbLecteurs == 0)   /* si c'était le dernier lecteur */
    redacteur.V();        /* autorise le mode écriture */
  mutexLecteur.V();       /* débloque les autres lecteurs */
 }
```

Les lecteurs/rédacteurs – Version 2

```
void demandeDEcriture ()
 {
  mutexRedacteur.P();     /* bloque les autres rédacteurs */
   nbRedacteur++;
   if (nbRedacteur == 1) /* si c'est le premier rédacteur */
    lecteur.P();         /* interdire le mode lecture */
  mutexRedacteur.V();     /* débloque les autres rédacteurs */
  redacteur.P();          /* essaye de passer en mode écriture */
 }

void finDEcriture ()
 {
  redacteur.V();           /* débloque le mode écriture */
  mutexRedacteur.P();      /* bloque autres rédacteurs */
   nbRedacteur--;
   if (nbRedacteur == 0)   /* si c'était le dernier rédacteur */
    lecteur.V();           /* autoriser le mode lecture */
  mutexRedacteur.V();      /* débloque autres rédacteurs */
 }
```

```
void lecteur ()
 {
  while (1)
   {
    demandeDeLecture ();
    lectureDesDonnées ();
    finDeLecture ();
   }
 }
```

```
void redacteur ()
 {
  while (1)
   {
    demandeDEcriture ();
    ecritureDesDonnées ();
    finDEcriture ();
   }
 }
```

Les lecteurs/rédacteurs – Version 3

La variante « pas de priorité » utilise :

- nbLecteurs : une variable qui comptabilise le nombre de processus qui lisent le SI en mode concurrent
- mutexLecteur : un sémaphore pour un accès exclusif à nbLecteurs
- mutexModeAcces : un sémaphore pour bloquer le système en mode lecture ou écriture
- mutexFifo : un sémaphore pour ne traiter qu'une demande à la fois

Les lecteurs/rédacteurs – Version 3

```
Entier    nbLecteurs      (0)
Semaphore mutexFifo       (1);
Semaphore mutexLecteur    (1);
Semaphore mutexModeAcces  (1);

void demandeDeLecture ()
 {
  mutexFifo.P();          /* bloque les autres processus */
   mutexLecteur.P();      /* bloque les autres lecteurs */
    nbLecteurs++;
    if (nbLecteurs == 1)  /* premier lecteur bloque */
     mutexModeAcces.P();  /* le systeme en mode lecture */
   mutexLecteur.V();      /* débloque les autres lecteurs */
  mutexFifo.V();          /* débloque les autres processus */
 }

void finDeLecture ()
 {
  mutexLecteur.P();       /* bloque les autres lecteurs */
   nbLecteurs--;
   if (nbLecteurs == 0)   /* dernier lecteur debloque */
    mutexModeAcces.V();   /* le systeme du mode lecture */
  mutexLecteur.V();       /* débloque les autres lecteurs */
 }
```

Les lecteurs/rédacteurs – Version 3

```
void demandeDEcriture ()
 {
  mutexFifo.P();           /* bloque les autres processus */
  mutexModeAcces.P();      /* bloque en mode écriture */
 }

void finDEcriture ()
 {
  mutexModeAcces.V();      /* débloque le mode écriture */
  mutexFifo.V();           /* débloque les autres processus */
 }
```

```
void lecteur ()
 {
  while (1)
   {
    demandeDeLecture ();
    lectureDesDonnées ();
    finDeLecture ();
   }
 }
```

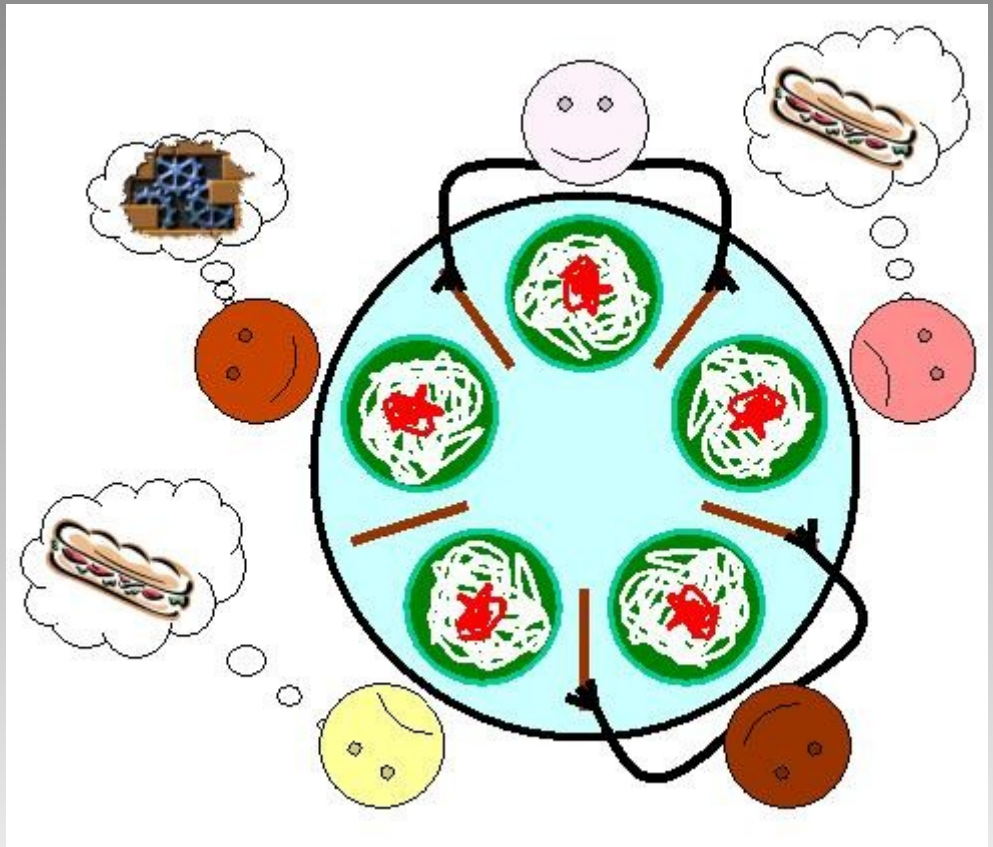
```
void redacteur ()
 {
  while (1)
   {
    demandeDEcriture ();
    ecritureDesDonnées ();
    finDEcriture ();
   }
 }
```

Les philosophes

Le problème a été proposé et résolu par Dijkstra en 1965.

Au moins 5 philosophes sont assis autour d'une table pour manger un plat de spaghettis avec des baguettes.

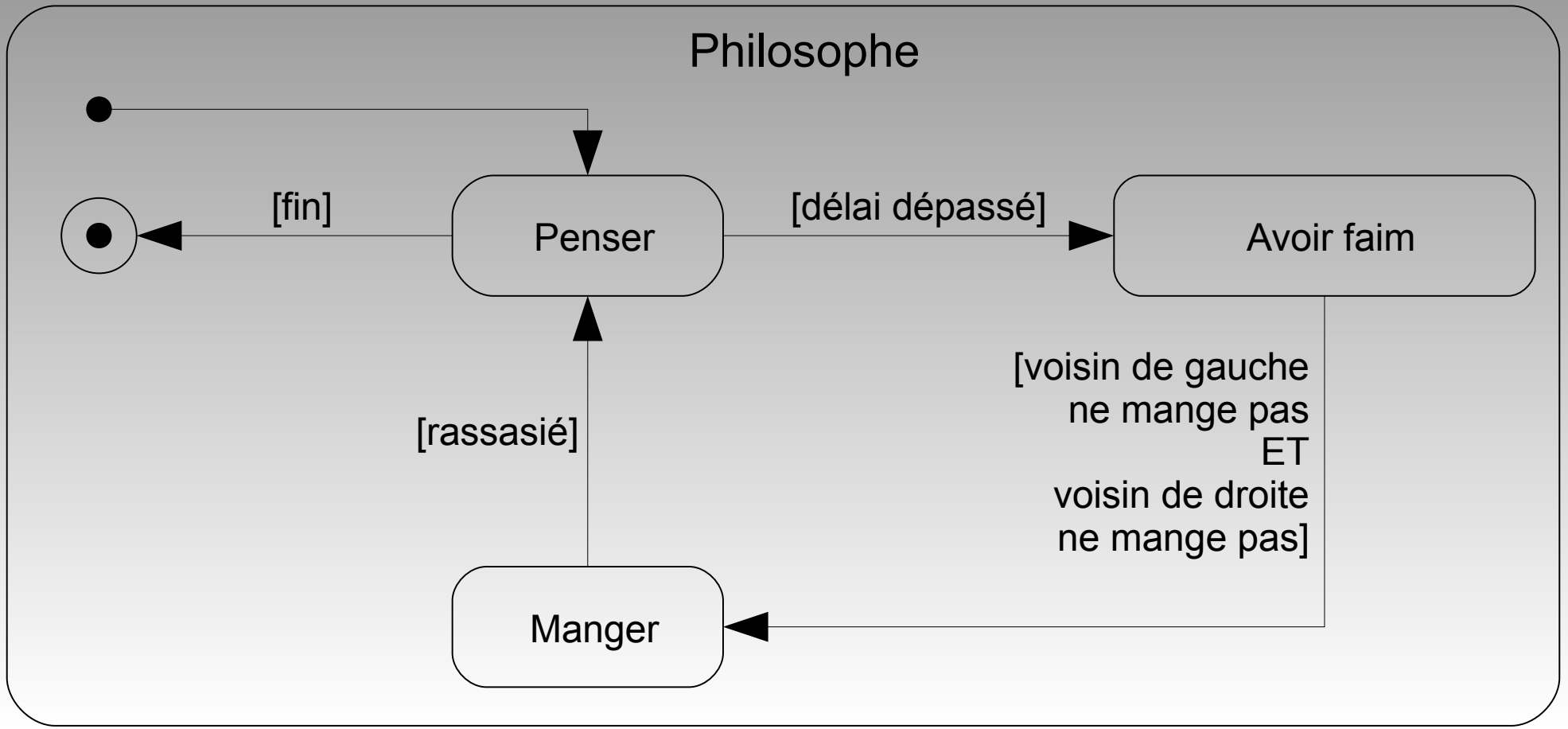
Chaque philosophe possède une baguette, mais il doit emprunter celle d'un voisin afin disposer de 2 baguettes.



Voir http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF
http://en.wikipedia.org/wiki/Dining_philosophers_problem

Les philosophes

Le comportement des philosophes peut être modélisé par le diagramme états-transitions ci-dessous :



Les philosophes

Pour implanter ce problème, il est nécessaire d'utiliser :

- un tableau d'entier pour conserver l'état de chaque philosophe ;
- un sémaphore mutex pour protéger l'accès à ce tableau d'entiers ;
- un tableau de sémaphores qui permet de bloquer les philosophes sur l'accès aux baguettes

Les philosophes

```
#define N 5
#define GAUCHE (i-1)%N
#define DROITE (i+1)%N
#define PENSER 0
#define FAIM   1
#define MANGER 2

int etat[N];
semaphore mutex;
Semaphore signPhil[N];

void philosophe (int i)
 {
  while (1)
   {
    penser ();
    prendreBaguette (i);
    manger ();
    poserBaguette (i);
   }
 }

void prendreBaguette (int i);
 {
  mutex.P();
   etat[i] = FAIM;
   tester (i);
  mutex.V();
  signPhil[i].P();    /* Attente des baguette */
 }
```

```
void poserBaguette (int i)
 {
  mutex.P();
   etat[i] = PENSER;
   tester (GAUCHE);
   tester (DROITE);
  mutex.V();
 }

void tester (int i)
 {
  if ((etat[i]       == FAIM)   &&
      (etat[GAUCHE] != MANGE)  &&
      (etat[DROITE] != MANGE))
   {
    etat[i] = MANGE;
    signPhil[i].V();   /* Il se débloque */
   }
 }
```

Les moniteurs

Les limites des sémaphores

Les sémaphores sont puissants, mais ils sont complexes à manipuler, car cela oblige à mélanger les primitives de synchronisation avec le reste du code.

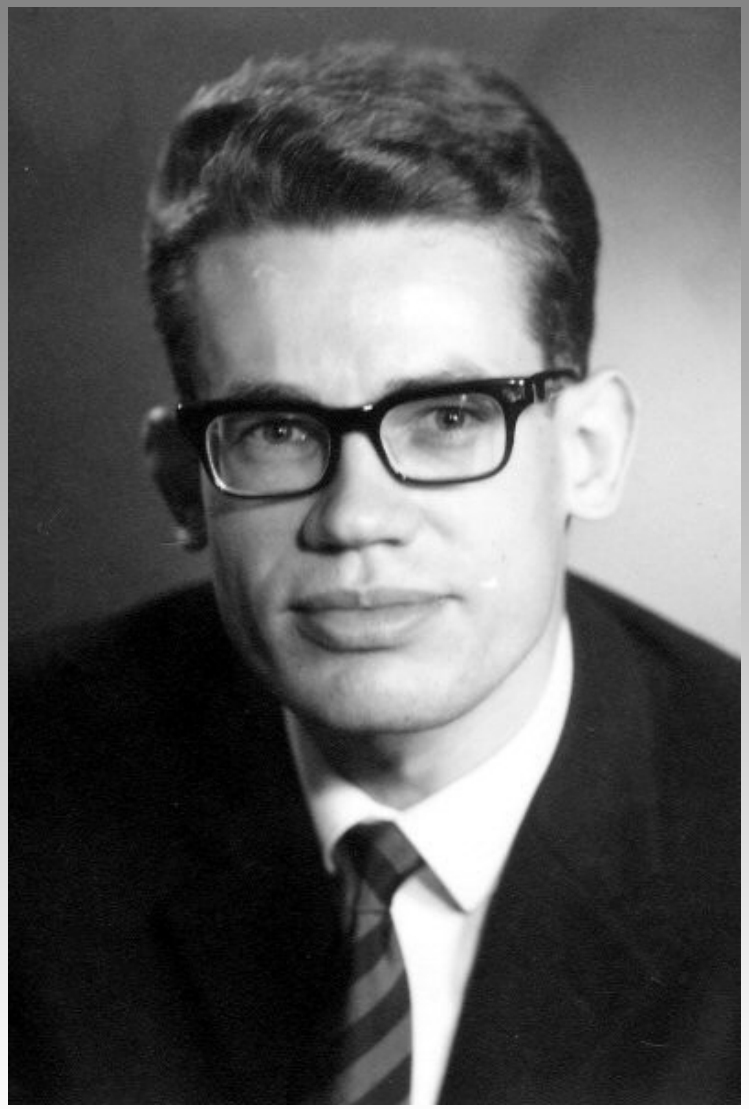
En outre, ils sont dangereux, car s'ils sont mal utilisés, ils peuvent provoquer des interblocages.

Voir http://padiou.perso.enseeiht.fr/2AI/SYNCHRO/main.pdf
http://www.uio.no/studier/emner/matnat/ifi/INF3151/h10/undervisningsmateriale/monitors.pdf

Les limites des sémaphores

Pour résoudre ces problèmes, Per Brinch Hansen a proposé, en 1972, un mécanisme de haut niveau pour gérer l'accès à des ressources partagées.

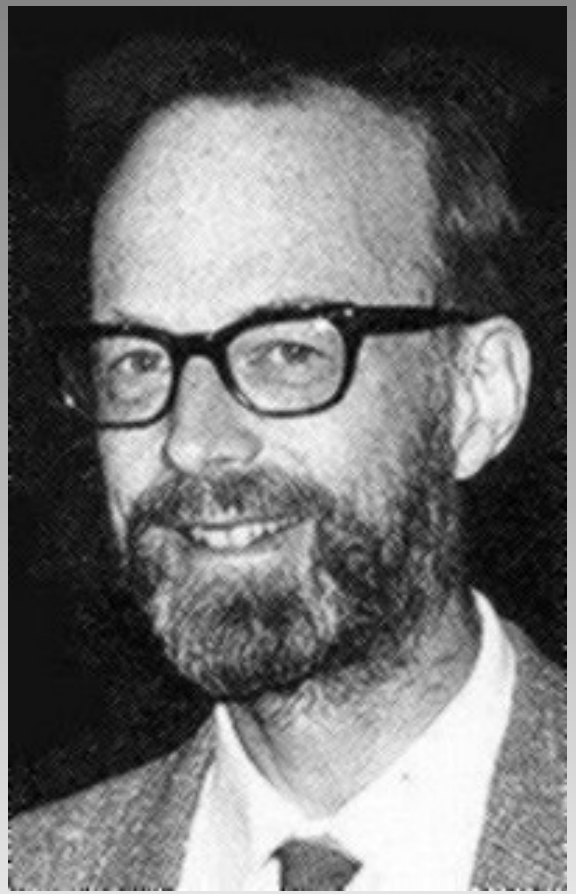
La description de ce mécanisme est complétée en 1975 par la proposition d'un Pascal concurrent.



Voir http://brinch-hansen.net/papers/1972a.pdf
http://brinch-hansen.net/papers/1975a.pdf

Les limites des sémaphores

Suite à cette première proposition, Sir Charles Antony Richard Hoare écrit, en 1974, un article « Monitors: an operating system structuring concept ».

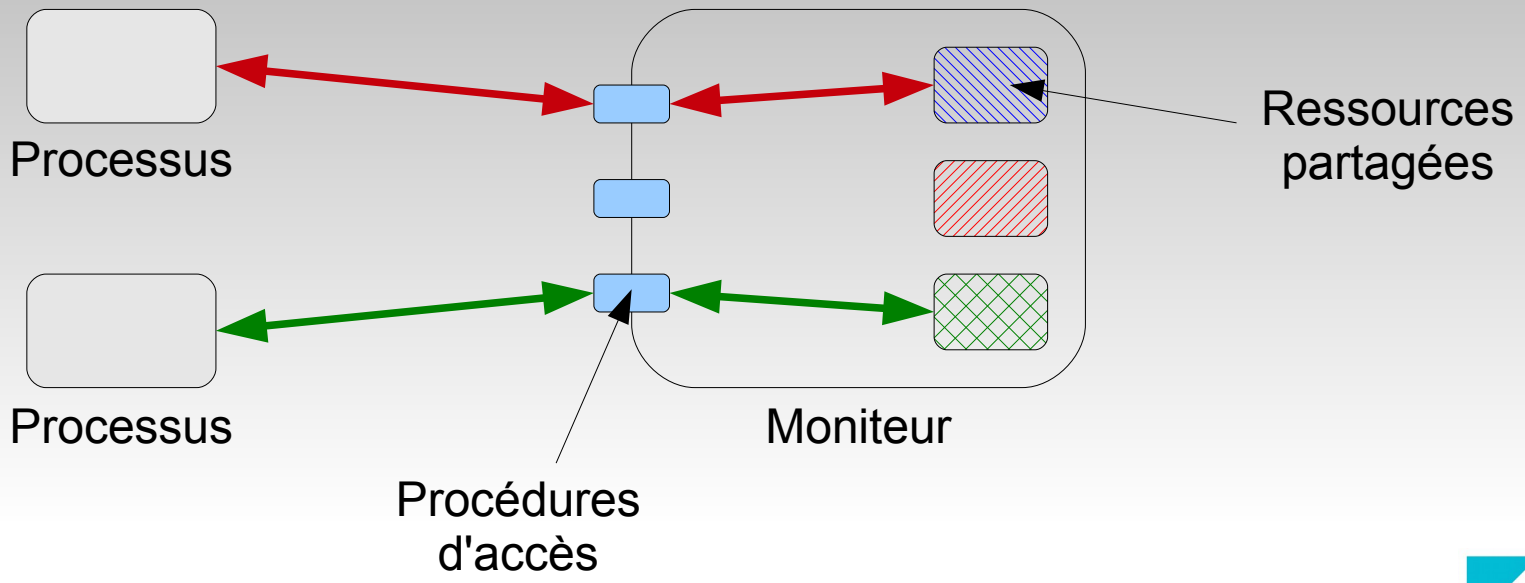


Voirhttp://www.google.fr/url?sa=t&source=web&cd=1&sqi=2&ved=0CCQQFjAA&url=http%3A%2%2Fciteseer.ist.psu.edu%2Fviewdoc%2Fdownload%3Bjsessionid%3D05F44914ED082F46A91D42DCDE58DB78%3Fdoi%3D10.1.1.91.3720%26rep%3Drep1%26type%3Dpdf&rct=j&q=C.%20A.%20R.%20Hoare.%20Monitors%20%3A%20an%20operating%20system%20structuring%20concept.&ei=3yJVTvLwLqHb4QTfney0Bw&usg=AFQjCNH4TF80GXyiiU1HV_25qF9lyBEUhQ&cad=rja

Le principe

Un moniteur encapsule des ressources communes qui sont accédées par procédures exécutées en exclusion mutuelle.

Par construction, un seul processus peut donc se trouver dans le moniteur (utiliser une de ses procédures) à un moment donné.



Le principe

Ce moniteur est complété par des conditions, des variables spéciales sur lesquelles les processus peuvent être bloqués. Il faut donc gérer, au sein de ces conditions des files d'attente de processus bloquées.

Une condition est utilisée au travers de deux primitives :

- wait qui bloque le processus appelant et libère l'accès exclusif au moniteur (le moniteur n'est pas verrouillé par un processus bloqué) ;
- signal qui débloque un processus de la file d'attente (ou ne fait rien dans le cas contraire.

Le producteur-consommateur

Voici une nouvelle version du producteur-consommateur.

```
Moniteur tampon
 {
  Condition plein (faux);
  Condition vide  (vrai);
  Entier    cpt   (0);

  Procedure placer (objet)
   {
    Si (cpt = N) wait (plein);
    // on place
    compteur++;
    Si (cpt = 0) signal (vide);
   }

  Procedure prendre () : objet
   {
    Si (cpt = 0) wait (vide);
    // on retire
    compteur--;
    Si (cpt = N-1) signal (plein);
   }
```

```
// producteur

while (1)
 {
  Obj o = produire();
  tampon.placer (o);
 }
```

```
// consommateur

while (1)
 {
  Obj o = tampon.prendre();
  consommer (o);
 }
```