Cours « système d'exploitation »
2ème année
IUT de Caen, Département d'Informatique
(François Bourdon)

Chapitre 3
Synchronisation de Processus (partie−2)

Plan

3.3 Interactions de processus

Des processus qui agissent en parallèle peuvent coopérer (partage d'information ou accélération d'un calcul), ou être en compétition les uns par rapport aux autres pour acquérir des ressources, quand elles sont en quantité insuffisante.

La base de l'interaction est la communication.

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

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

Des interactions mal contrôlées peuvent être la cause d'un mauvais fonctionnement du système et d'une utilisation impropre des ressources.

Pour cela on peut regarder deux problèmes : **Déterminisme** et **blocage**.

Déterminisme d'un système de tâches

Définition : C'est l'étude de la possibilité de décider si un système de tâches donné, fournit pour chacun de ses composants, **la même suite de résultats.**

Soit, par exemple, deux processus qui accèdent sans contrôle à une même cellule mémoire M, contenant la valeur 10, le premier pour y ajouter 5, le deuxième pour doubler la valeur contenue dans M.

Suivant l'ordre d'accès à M des deux processus, on obtient comme valeur finale soit :

*(10 + 5) * 2 = 30,*

soit :

*(10 * 2) + 5 = 25.*

Ce problème est inhérent aux systèmes multi−programmés, puisque les processus peuvent être mis en attente à des instants quelconques, pour des durées qui dépendent de paramètres extérieurs.

Lorsque l'on sait résoudre ce problème (notion **d'interférence**), on peut envisager de transformer un processus séquentiel (une chaîne de tâches) en un système équivalent, où certaines tâches sont exécutées en parallèle (**parallélisme maximal**).

Blocage dans un système de tâches

Définition : Un blocage se produit lorsqu'un ensemble de processus est tel que chacun d'eux tient au moins une ressource et est en attente, pour poursuivre sa progression, d'une ressource tenue par l'un des autres processus.

Pour traiter ce problème on a le choix entre trois types de techniques :

- **prévention,**
- **détection,**
- **évitement.**

Les techniques de **prévention** : Garantir que les situations de blocage ne puissent pas se produire, en utilisant des contraintes très fortes.

Les techniques de **détection** : Laisser le blocage se produire, puis après l'avoir détecté, reprendre les ressources de certains processus pour les redistribuer à d'autres, qui peuvent ainsi poursuivre leur exécution.

Les techniques d'**évitement** : En contrôlant pas à pas la progression des processus, faire en sorte qu'à chaque étape d'attribution de ressources, il reste une possibilité au système d'échapper à un blocage.

3.4 Synchronisation de processus

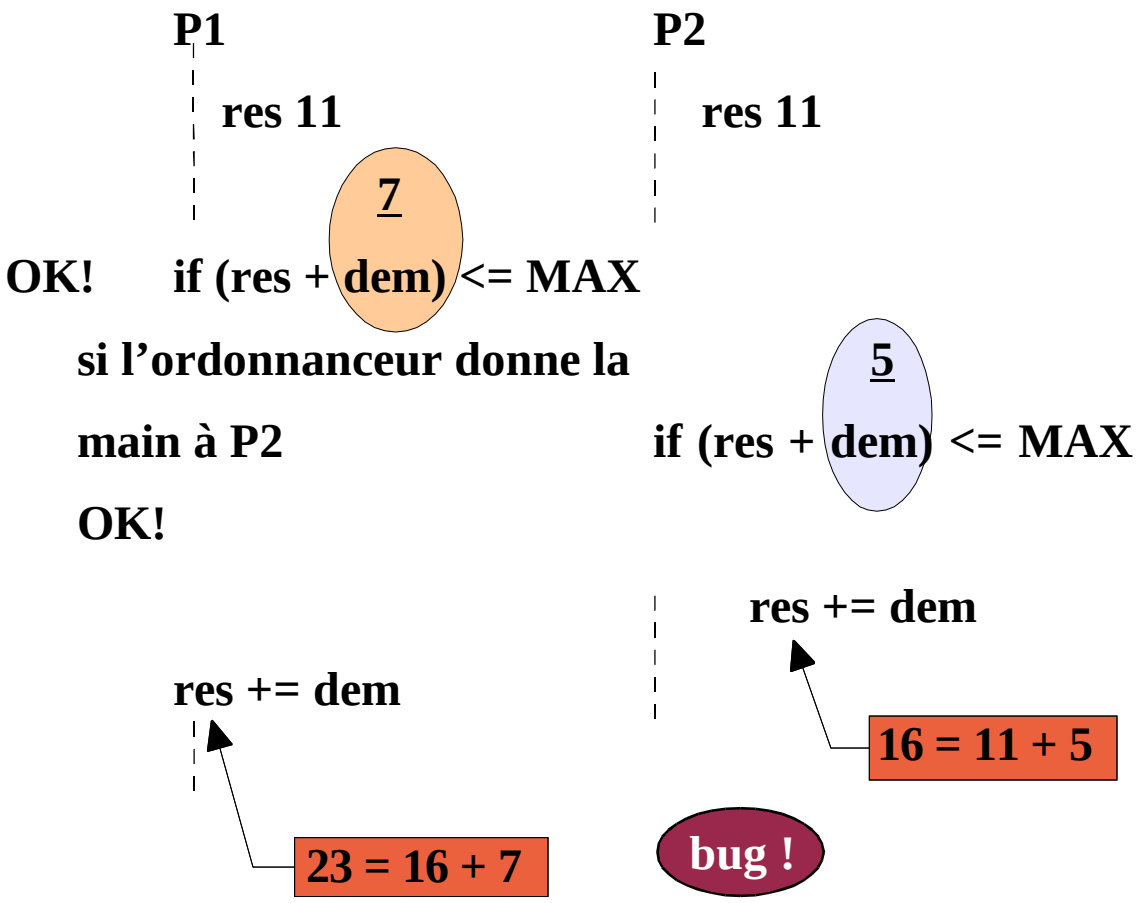
Nous avons vu qu'en définissant une relation de précédence sur un ensemble de tâches, on obtient un système de tâches satisfaisant les contraintes imposées en terme de production/consommation de ressources, mais aussi en terme de *parallélisation maximale* des tâches.

> *Problème* : Dans certains cas cette approche est insuffisante. Il se peut que plusieurs comportements (mots du langage) soient valides par rapport au graphe de précédence produit pour une situation donnée, alors que certains d'entre eux ne donnent pas le même résultat.

Exemple : La réservation de place de spectacle.

```
#define  MAX   20
int   res = 0;          /* nb de places réservées  */
void  reserver  (int dem)  {
        if ( res + dem <= MAX)   res += dem ;
        else ...      /* erreur  */ ;
}
```

si l'on a 2 processus



On s'aperçoit qu'éliminer les cas divergeant revient à rendre **indivisible (atomique)** la séquence de plusieurs tâches dans le graphe de précédence.

Les solutions mises en oeuvre ont en commun le fait qu'elles ajoutent toujours de nouvelles tâches au graphe de précédence initial.

La section critique

Définition : La **section critique** d'un programme donné, est une suite d'instructions de ce programme dont l'exécution est gérée en **exclusion mutuelle.**

Un processus parmi ceux qui s'exécutent en parallèle ne peut entrer en section critique si l'un des autres s'y trouve déjà.

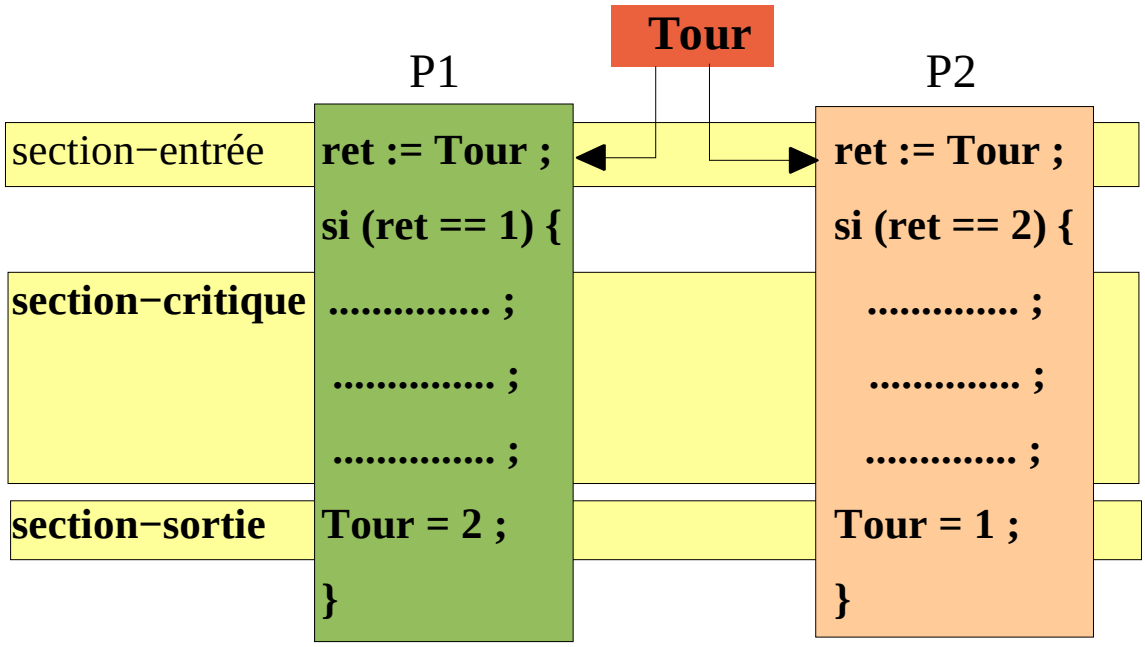
Intuitivement, un processus attend dans la section d'entrée qu'une certaine condition soit satisfaite pour entrer en section critique et signale sa sortie dans la section de sortie.

```
répeter
        <section restante>
    <section d'entrée>
        <section critique>
    <section de sortie>
jusqu'à faux;
```

Si dans certains systèmes il existe des dispositifs implantés au niveau matériel, qui assurent l'indivisibilité des instructions élémentaires, ce n'est pas le cas partout et surtout pas dans les systèmes répartis.

Il existe des **solutions logicielles** (utilisation de variables partagées) pour pallier à ces manques matériel, par exemple :

Soit deux processus $\mathbf{P_1}$ et $\mathbf{P_2}$ qui partagent une variable commune nommée ***Tour***, qui peut prendre les valeurs 1 ou 2. La ***section d'entrée*** consiste à consulter la valeur de ***Tour***. L'exécution de la ***section critique*** de $\mathbf{P_i}$ n'est autorisée que si ***Tour*** a la valeur ***i***. Lorsque la section critique est terminée, le processus exécute sa ***section de sortie*** en affectant à ***Tour*** le numéro de l'autre processus, permettant à celui−ci d'entrer dans sa propre ***section critique***.



Cette solution a des inconvénients en particulier si l'un des processus s'arrête définitivement, alors que la variable ***Tour*** n'a pas été changée.

Pour cela on impose une condition supplémentaire, appelée **condition de progression** :

> *Définition* : un processus bloqué en **section restante** ne peut pas empêcher un autre processus d'entrer en **section critique.**

D'autres algorithmes mettent en évidence que le problème de la *section critique* n'est pas simple à résoudre, surtout quand des conditions supplémentaires s'y greffent, comme la **progression** et l'**absence de blocage**.

Une dernière condition, appelée **attente bornée**, permet d'assurer une certaine équité entre les processus :

> *Définition* : Lorsqu'un processus est en attente de sa section critique, il existe une borne supérieure au nombre de fois où d'autres processus exécutent leur section critique.

Cette condition interdit à certains processus d'exécuter itérativement leur section critique, en laissant un processus attendre, pendant une durée arbitrairement longue, l'entrée dans sa propre section critique.

Certains algorithmes comme celui de *Peterson* respectent ces quatres conditions :

- exclusion mutuelle,
- absence de blocage,
- progression,
- attente bornée.

Il existe des solutions qui utilisent des **dispositifs matériels** particuliers pour résoudre le problème de la section critique. Ces solutions sont utilisées dans la pratique sur systèmes monoprocesseurs centralisés.

Désarmement des interruptions

On désarme les interruptions pendant toute la durée de la *section critique* et on les réarme en sortie de la *section critique*.

L'unité centrale reste allouée à ce processus jusqu'à la fin de sa **section critique**, ce qui garantit l'***exclusion mutuelle.***

Le problème c'est qu'il peut bloquer un processus prioritaire, d'où la difficulté de mettre cette possibilité dans les mains des utilisateurs.

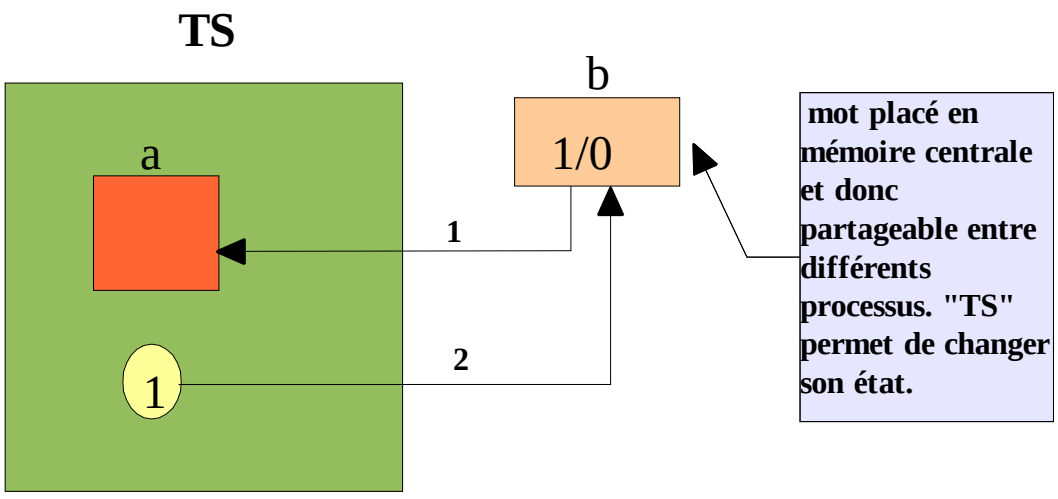
Instruction Test−and−Set

De nombreux ordinateurs disposent d'une instruction élémentaire (**Test−and−Set**, **TS** ou **TAS**) exécutée par le matériel, qui permet de lire et d'écrire le contenu d'un mot de la mémoire de manière indivisible.

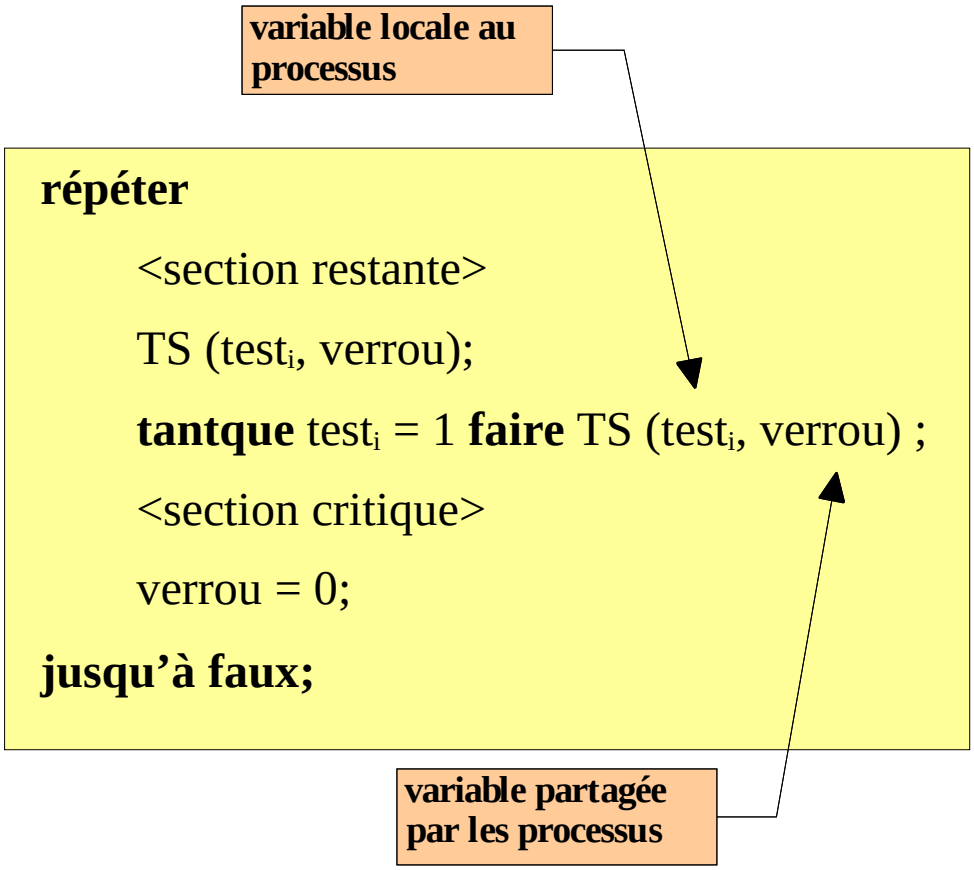
```
procédure TS ( var a, b: entier) ;
        début
                a := b ;
                b := 1 ;
        fin;
```

Cette instruction a deux opérandes, un registre **a** et un mot **b** (partageable) de la mémoire centrale. Elle copie le mot dans le registre et place la valeur 1 dans le mot.

Cette instruction est toujours **exécutée de manière indivisible**.

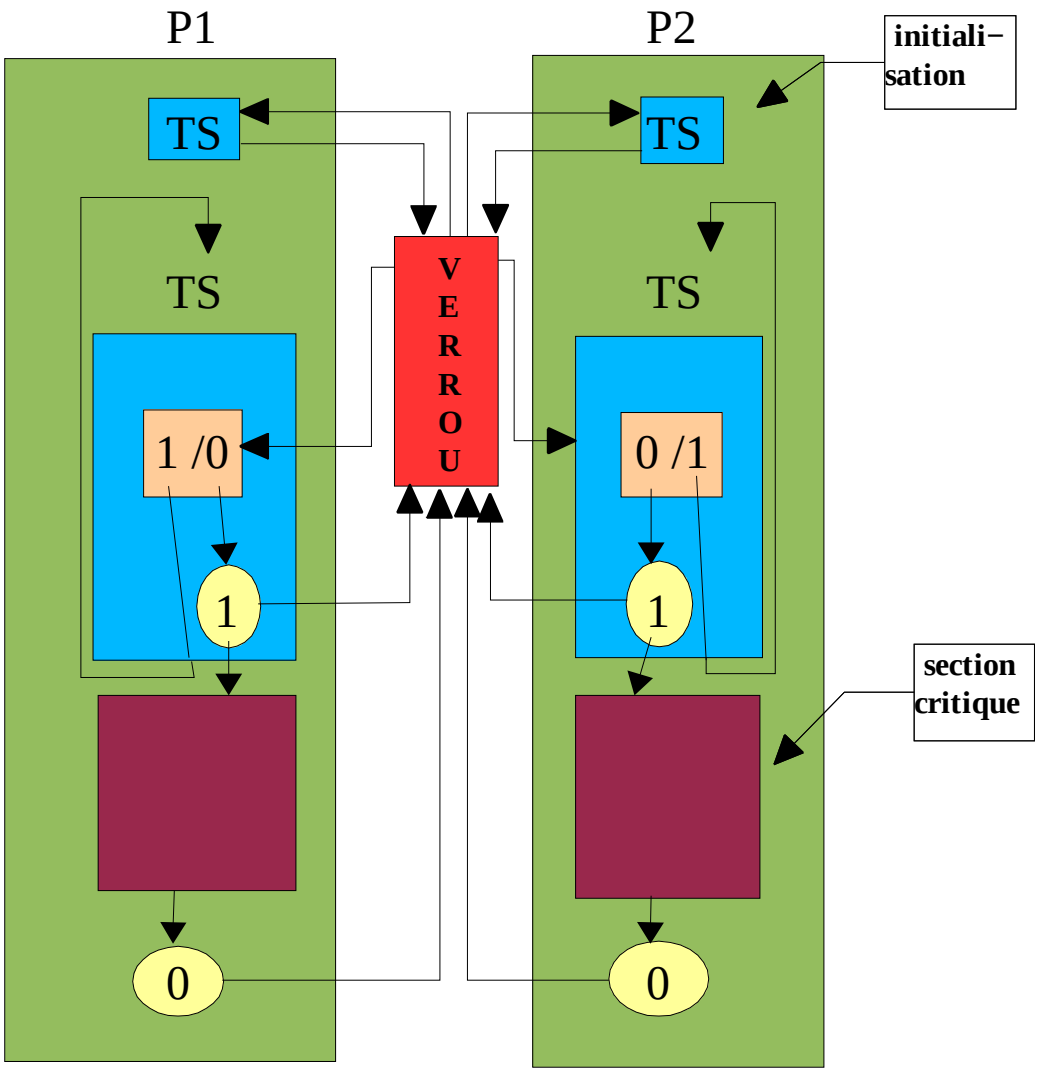


Deux processus partagent une variable appelée ***verrou*** :



Si $test_i$ vaut 1 **alors** le verrou est mis

Sinon ($test_i$ vaut 0) **alors** P_i peut entrer en **section critique**



Cette solution satisfait la contrainte d'***exclusion mutuelle***, mais pas celle de l'***attente bornée*** (nombre max. de fois où un processus peut exécuter consécutivement sa section critique), qui devra être assurée par d'autres moyens.

Dans le cas de systèmes multi−processeurs, **TS** peut être utilisée avec des limitations.

- Problème de l'**entrelacement** des accès à la variable partagée.

- Problème de l'**attente active** : en attendant que le verrou soit libéré, le processus fait une boucle inutile (tantque ...).

- Enfin il existe un cas de **blocage** difficile à détecter : si $\mathbf{P_1}$ est dans sa *section critique* et qu'un processus $\mathbf{P_2}$ arrive avec une très grande priorité, alors $\mathbf{P_1}$ est mis en attente. Or si $\mathbf{P_2}$ veut entrer en *section critique*, il va tester le verrou qui a été bloqué par $\mathbf{P_1}$. D'où le blocage.

Une solution consiste à mettre les processus qui trouvent le verrou à **1**, automatiquement en état "**en attente**". Ils ne concourrent plus pour l'unité centrale.

Symétriquement, la libération du verrou est associée au réveil par le système de l'un des processus bloqués.

Il subsiste des problèmes de synchronisation entre les processus, d'où l'utilisation de certaines primitives dédiées comme les **sémaphores**.

Les sémaphores

C'est un mécanisme de synchronisation entre processus. Un sémaphore **S** est une variable à valeurs entières positives ou nulle, manipulable par l'intermédiaire de deux opérations **P** (***proberen***) et **V** (***verhogen***) :

P (S) : si S ≤ 0, alors mettre le processus en attente ;
sinon : S := (S − 1).

V (S) : S := (S + 1) ; réveil d'un processus en attente.

P (S) : Cette opération correspond à une tentative de franchissement. S'il n'y a pas de jeton pour la section critique alors attendre, sinon prendre un jeton et entrer dans la section.

V (S) : Rendre son jeton à la sortie de la section critique. Chaque dépôt de jeton V (S) autorise le passage d'une personne. Il est possible de déposer des jetons à l'avance.

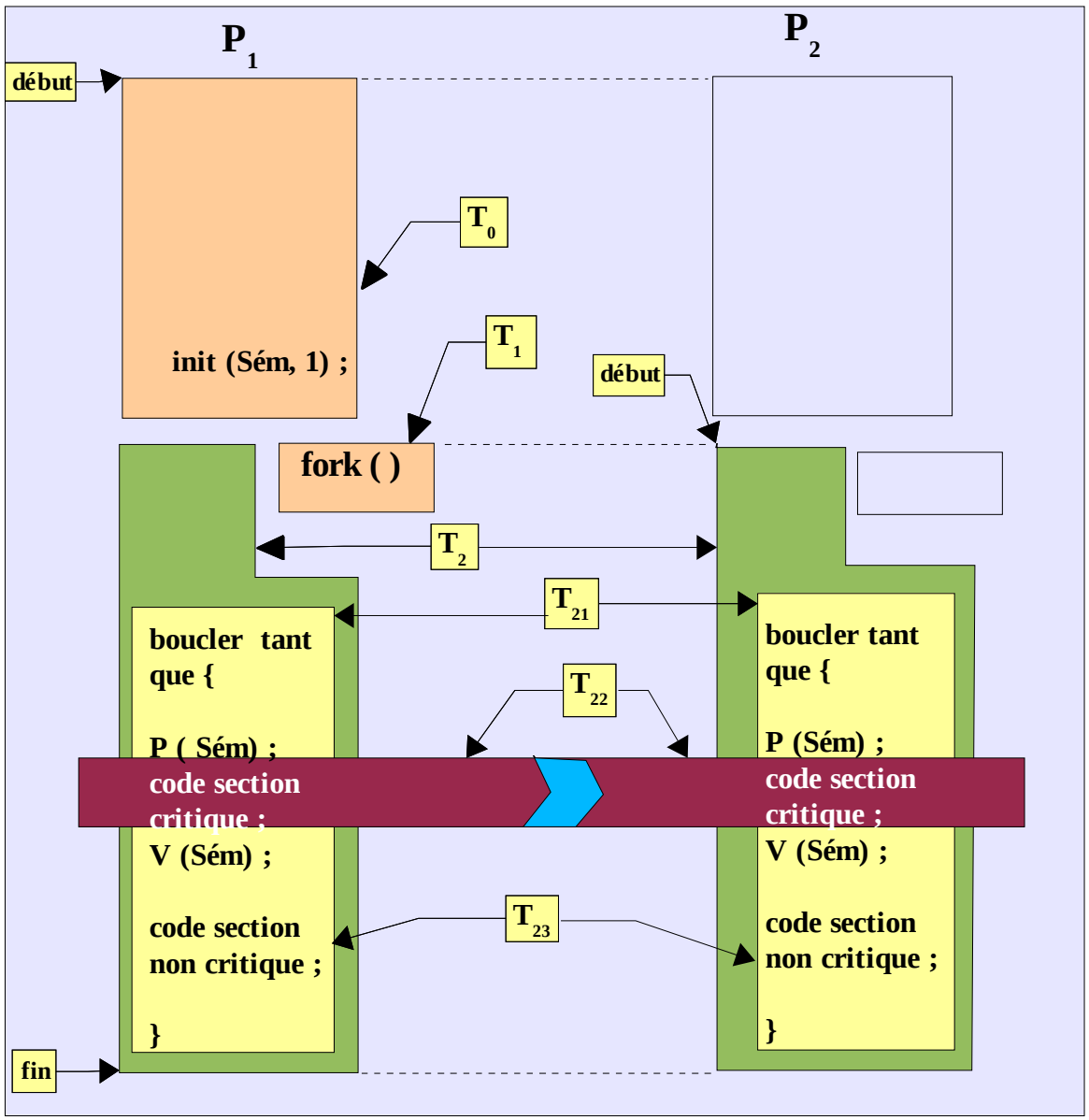
Toute implantation de ce mécanisme repose sur :

✉ l'**atomicité** des opérations **P** et **V**, c'est−à−dire sur le fait que la suite d'opérations les réalisant (section critique) est non interruptible (afin d'éviter les conflits entre processus).

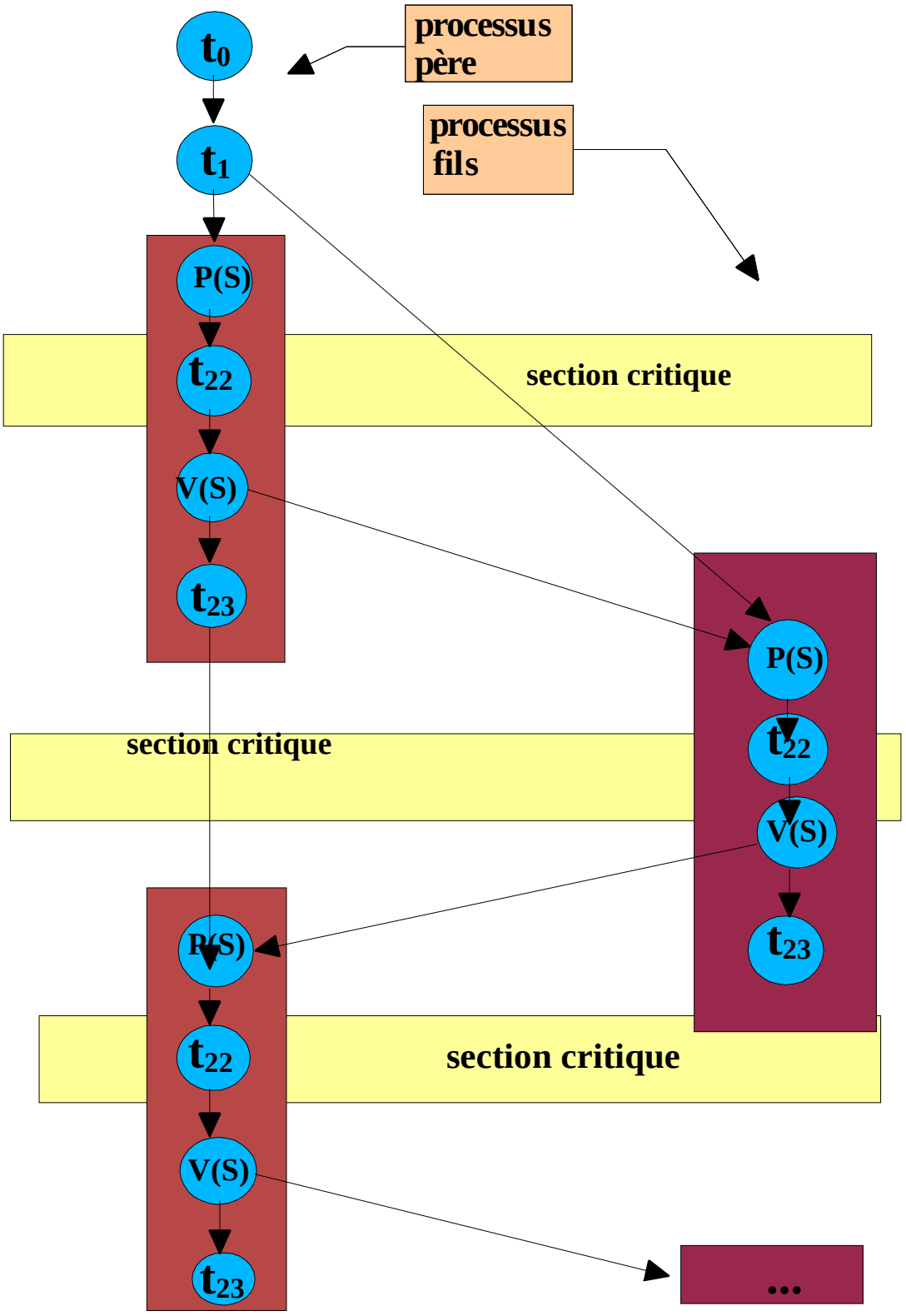
✉ et l'existence d'un mécanisme de **file d'attente**, permettant de mémoriser les demandes d'opération **P** non satisfaites et de réveiller les processus en attente.

Les sémaphores sont un outil général pour résoudre les problèmes de synchronisation.

Schématiquement, on peut bloquer un processus sur une opération **P** tant qu'une condition n'est pas réalisée. Lorsqu'elle le devient, un processus le signale aux autres en exécutant une opération **V**, qui libère l'un des processus en attente.



Graphe des tâches correspondant



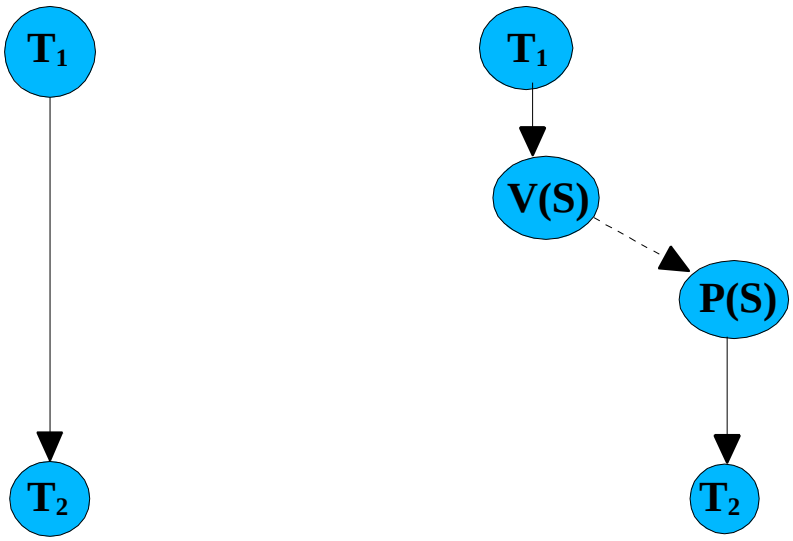
Autre exemple d'utilisation des sémaphores

Ici on veut synchroniser les tâches T_1 et T_2 exécutées respectivement par les processus P_1 et P_2. Pour cela on initialise un sémaphore S à 0 et on remplace les tâches T_1 et T_2 par les tâches T'_1 et T'_2, avec :

$T'_1 = T_1 ; V(S)$ et $T'_2 = P(S) ; T_2$.

```
    init(S, 0);
parbegin
      début T1 ; V(S) fin;
      début P(S) ; T2 fin
parend;
```

Cela revient à remplacer le graphe de précédence initial par :



Troisième exemple d'utilisation des sémaphores

Supposons qu'il existe k exemplaires d'une même ressource ($k \geq 2$).

Pour une ressource, l'accès à cette dernière constitue une section critique, mais il peut y avoir autant de processus en section critique que d'exemplaires initialement disponibles.

On utilise un sémaphore (*nbr_de_ressources*) dont la valeur représente à chaque instant le nombre d'exemplaires disponibles de la ressource.

```
    init (nbr_ressources, k) ;
répéter
        < section restante >
        P (nbr_ressources) ;
        < section critique >
        V (nbr_ressources) ;
jusqu'à faux ;
```

Lorsque k processus sont en section critique, le sémaphore vaut 0. Si d'autres processus cherchent à acquérir un exemplaire de la ressource (P), ils sont alors placés dans une file d'attente.

Dès qu'un processus sort de sa section critique (V), l'un des processus en attente est réveillé et un exemplaire de la ressource lui est allouée. La valeur du sémaphore reste à 0.

Lorsque la file d'attente est vide et qu'un processus sort de sa section critique, l'opération V fait passer la valeur du sémaphore à 1.

Enfin si tous les processus sont sortis de leur section critique, le sémaphore a repris sa valeur initiale k, ce qui exprime que tous les exemplaires de la ressource sont à nouveau disponibles.

Quatrième exemple d'utilisation des sémaphores

Le processus P_1 doit posséder plusieurs ressources distinctes, comme par exemple un tampon de données et un segment de mémoire partagée, pour déplacer des données de l'une à l'autre.

$\Rightarrow$

2 sémaphores et 2 opérations **P** pour disposer des deux ressources recherchées.

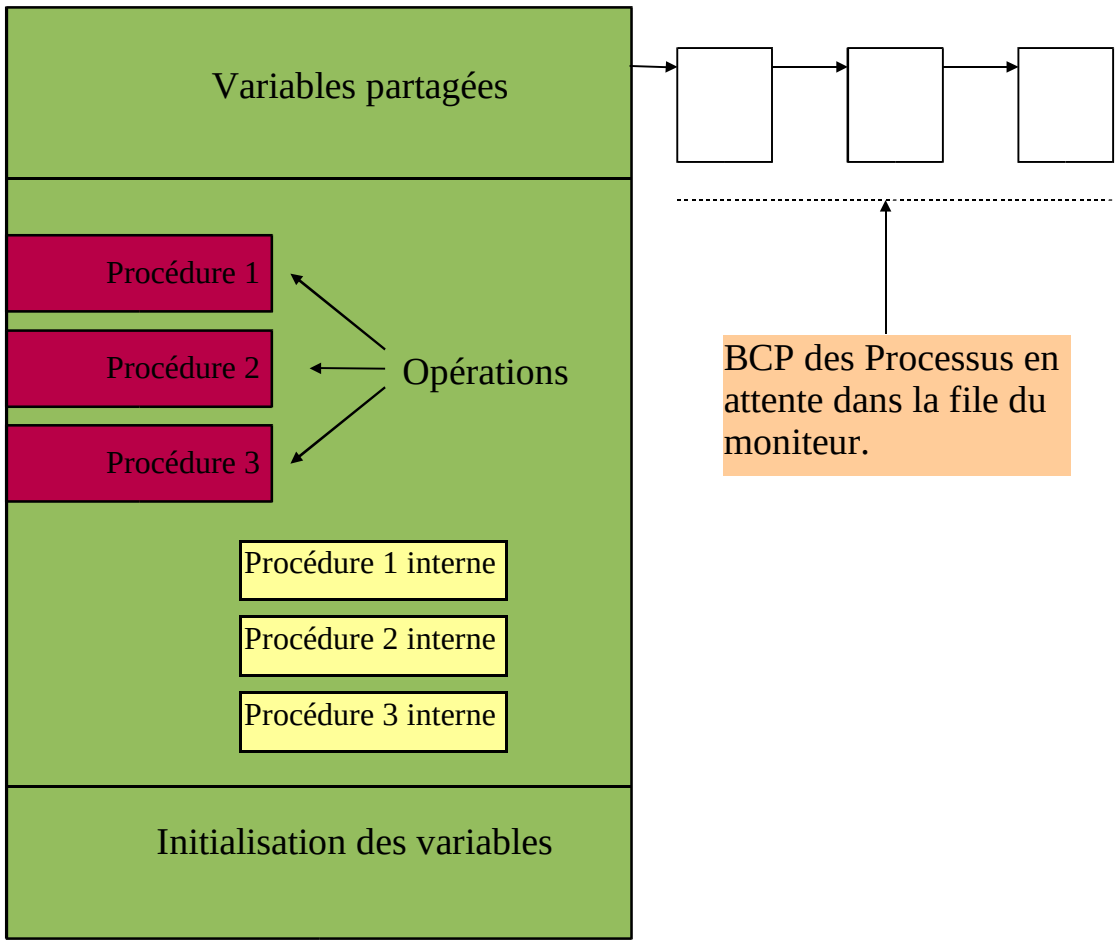
MAIS risque d'INTERBLOCAGE : Si par exemple un processus possède l'accès exclusif au tampon de données et veut accéder au segment de mémoire partagée, alors qu'un autre est en situation inverse.

SOLUTION : P ( ) et V ( ) ne sont pas réalisées sur un seul sémaphore S, mais sur un tableau de sémaphores. C'est le cas par défaut dans Unix (interface Linux). Un processus continue son exécution si les opérations P se sont bien déroulées sur tous les éléments du tableau. Pour l'utilisateur ceci est réalisé de façon atomique (en un seul appel système) .

Les moniteurs (de Hoare)

Ce sont des structures de synchronisation modulaires, qui regroupent des variables partagées par plusieurs processus, ainsi que les instructions qui les manipulent.

Ces structures sont plus générales et plus complexes (sémantique) que les sémaphores.



Remarque 1 : Un seul processus est actif à un instant donné dans le moniteur.

Remarque 2 : Chaque entrée (opération) est accessible en exclusion mutuelle.

Remarque 3 : Pas de variable exportée, seules les procédures du moniteur sont susceptibles de manipuler les variables du moniteur.

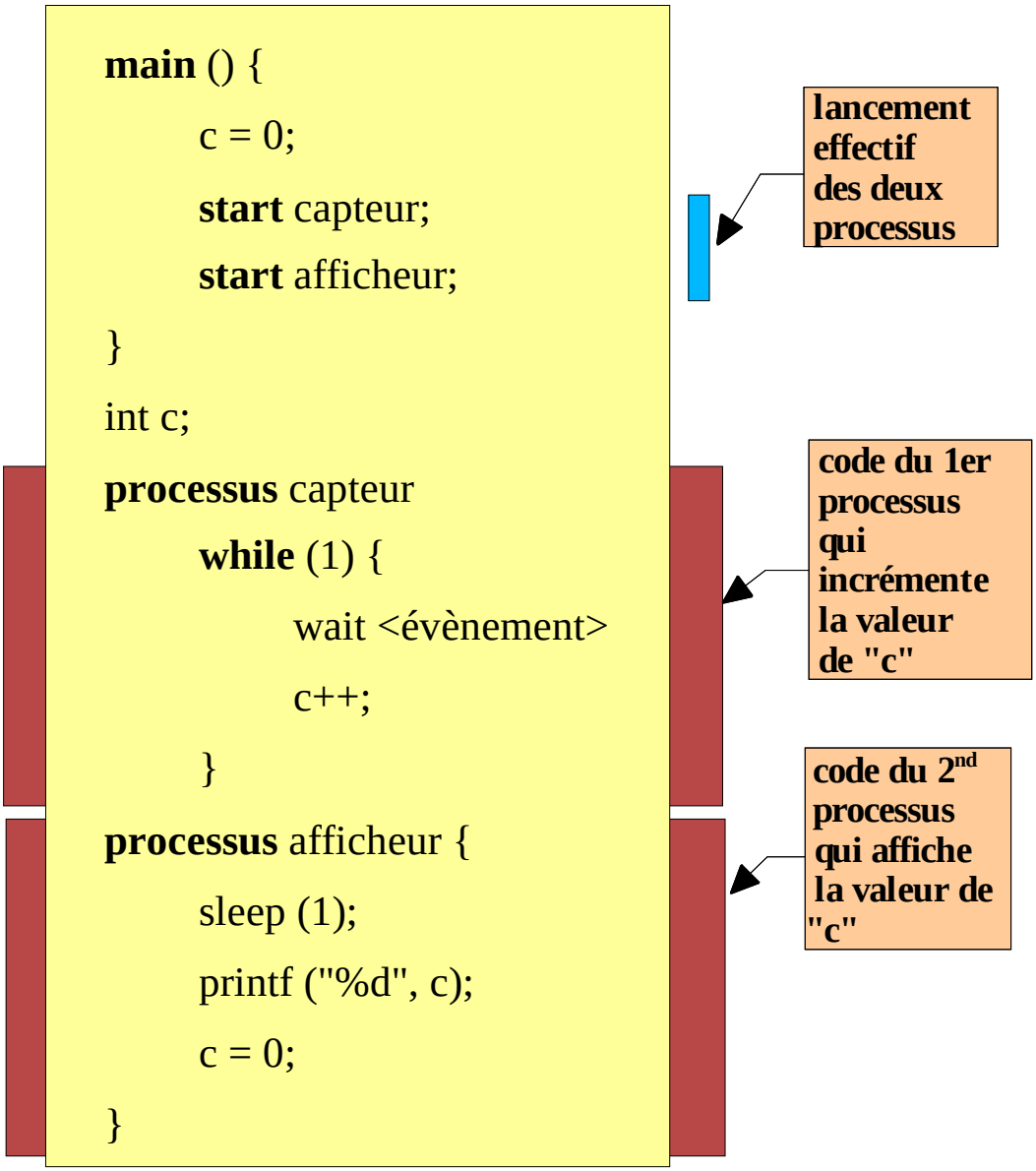
Remarque 4 : Au lieu d'être dispersées dans les différents processus, les sections critiques sont transformées en opérations (procédures) d'un moniteur.

Remarque 5 : La gestion de la section critique est à la charge du moniteur et non pas de l'utilisateur. Le moniteur est implanté tout entier comme une section critique.

Remarque 6 : Si le moniteur appelé par un processus pour manipuler une donnée partagée, est occupé, le processus est placé dans la file d'attente du moniteur. Dès qu'il est libéré, l'un des processus de la file est choisi et la procédure invoquée est exécutée.

Exemple simple : Compteur Geiger (toutes les secondes il affiche le nombre de désintégration/sec.)

Naïvement, sans moniteur



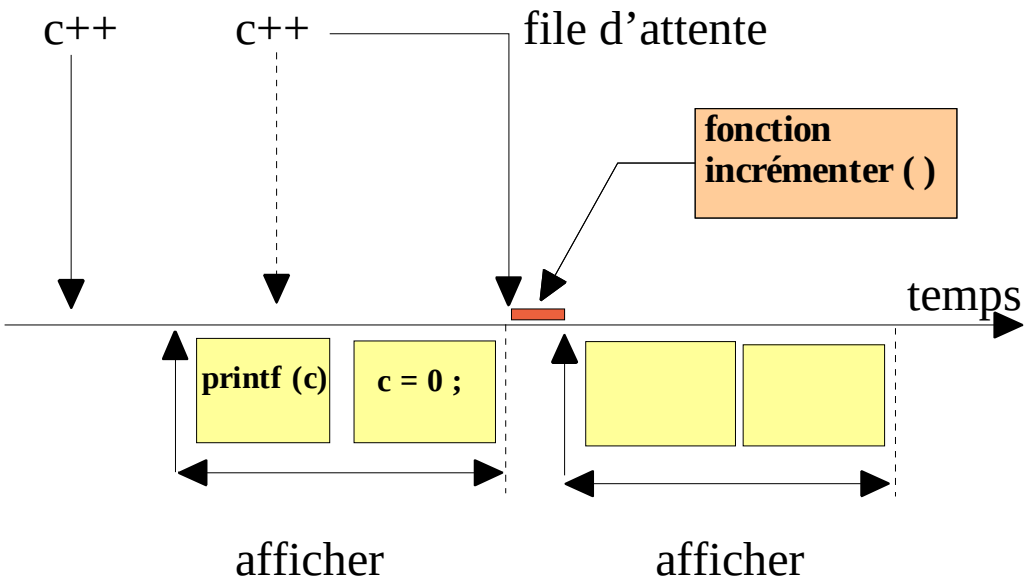
On a un risque d'entrelacement causé par l'instruction **" c = 0;" après l'instruction "c++;",** qui peut conduire à **un nbre d'événements affichés ≤ au nbre d'événements réels.**

Avec moniteur de Hoare

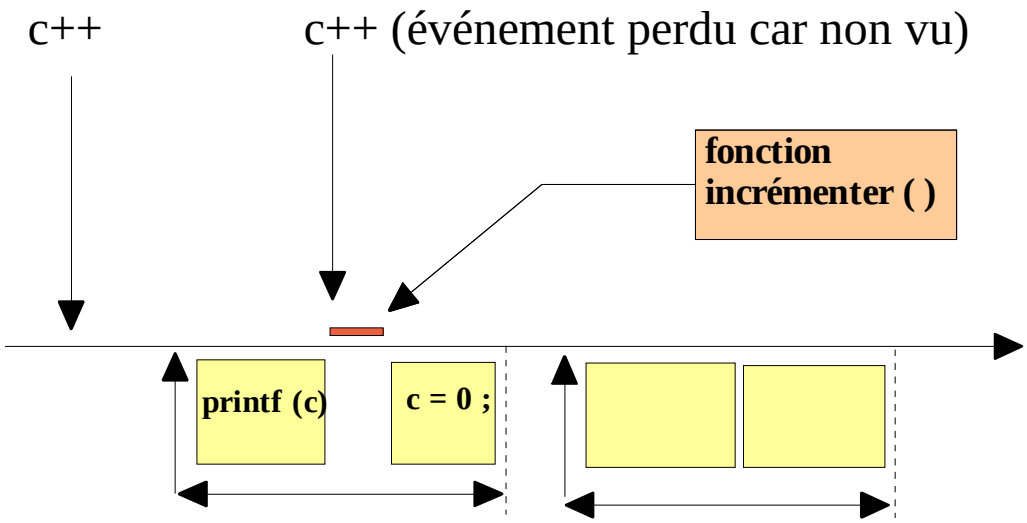
```
moniteur compteur ;          /* déclaration du moniteur
                             "compteur" */
export incrémenter, afficher;      // opérations accessibles
int c;
Procédure incrémenter;
        begin       c++;        end;
Procédure afficher;
        begin       printf ("%d", c);       c = 0;       end;
init;   begin       c = 0       end;

/* module de surveillance  */
import compteur
processus capteur
                while (1) {
                        wait <évènement>
                        compteur.incrementer;  }
processus afficheur {
                while (1) {
                        compteur.afficher;  }

/* Programme principal  */

start capteur
start afficheur
```

La séquence **"c++;" et "c = 0;"** est impossible.

avec moniteur



sans moniteur



3.5 Problèmes classiques de synchronisation

Les producteurs / consommateurs

Des activités productrices produisent des informations consommées par des activités consommatrices.

> L'exécution concourante des producteurs et des consommateurs repose sur l'utilisation d'un ensemble de N *buffers* intermédiaires remplis par les producteurs et vidés par les consommateurs.

Les producteurs sont en compétition entre eux, les consommateurs aussi.

Exemples

- Le processus clavier produit des caractères qui sont consommés par le processus d'affichage à l'écran.
- Le pilote de l'imprimante produit des lignes de caractères, consommées par l'imprimante.
- Un compilateur produit des lignes de code consommées par l'assembleur.

Il existe toute une classe de problèmes selon les règles de fonctionnement choisies :

- l'information produite et consommée est de taille fixe/variable,
- une information produite est consommée une seule/plusieurs fois,
- une information produite doit nécessairement être consommée avant qu'un producteur puisse réutiliser le buffer qui la contient,

- une opération impossible est bloquante/non bloquante ; par exemple lors d'une tentative de production, si le nombre de *buffers* libres est insuffisant ou lors d'une tentative de consommation, si le nombre de *buffers* pleins est insuffisant.

- les objets sont consommés dans l'ordre où ils ont été produits (FIFO).

- l'accès au tampon constitue une session critique pour chacun des processus :

  dans ce cas soit le tampon n'est pas de taille bornée ; seul le consommateur doit attendre quand le tampon est vide ;

  dans le cas contraire (*buffer* borné) il faut en plus garantir que le producteur attend lorsque le tampon/*buffer* est plein.

Les lecteurs / rédacteurs

Un objet (par ex. un fichier, un enregistrement dans un fichier ou une base de données toute entière) est partagé entre plusieurs activités concurrentes. Certaines activités (les lecteurs) ne modifient pas le contenu de l'objet contrairement à d'autres (les écrivains).

> Les lecteurs peuvent donc accéder simultanément au fichier. Un écrivain au contraire doit accéder seul au fichier. Si le fichier est disponible, lecteur et rédacteur ont la même priorité.

Il existe plusieurs versions classiques du problème.

variante n°1 : ***Priorité aux lecteurs***.

S'il existe des lecteurs sur le fichier, toute nouvelle demande de lecture est acceptée.

- **risque** : le rédacteur peut ne jamais accéder au fichier (famine).

variante n°2 : ***Priorité aux lecteurs, sans famine des rédacteurs.***

S'il existe des lecteurs sur le fichier, toute nouvelle demande de lecture est acceptée sauf s'il y a un rédacteur en attente.

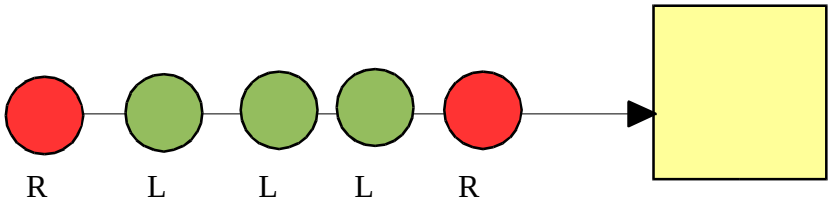
variante n°3 : ***Priorité aux rédacteurs***.

Un lecteur ne peut lire que si aucun rédacteur n'est présent ou en attente.

- **risque** : famine des lecteurs.

variante n°4 : ***FIFO***.

Les demandes d'accès à l'objet sont servies dans l'ordre d'arrivée. S'il y a plusieurs lecteurs consécutifs, ils sont servis ensemble.



- **risque** : Le regroupement des lecteurs est inefficace si les demandes sont lecteur/écrivain en alternance.

Ces règles nécessitent de distinguer un processus effectuant une **nouvelle demande** pour une opération de lecture ou d'écriture d'un processus en attente **ayant déjà fait cette demande.**

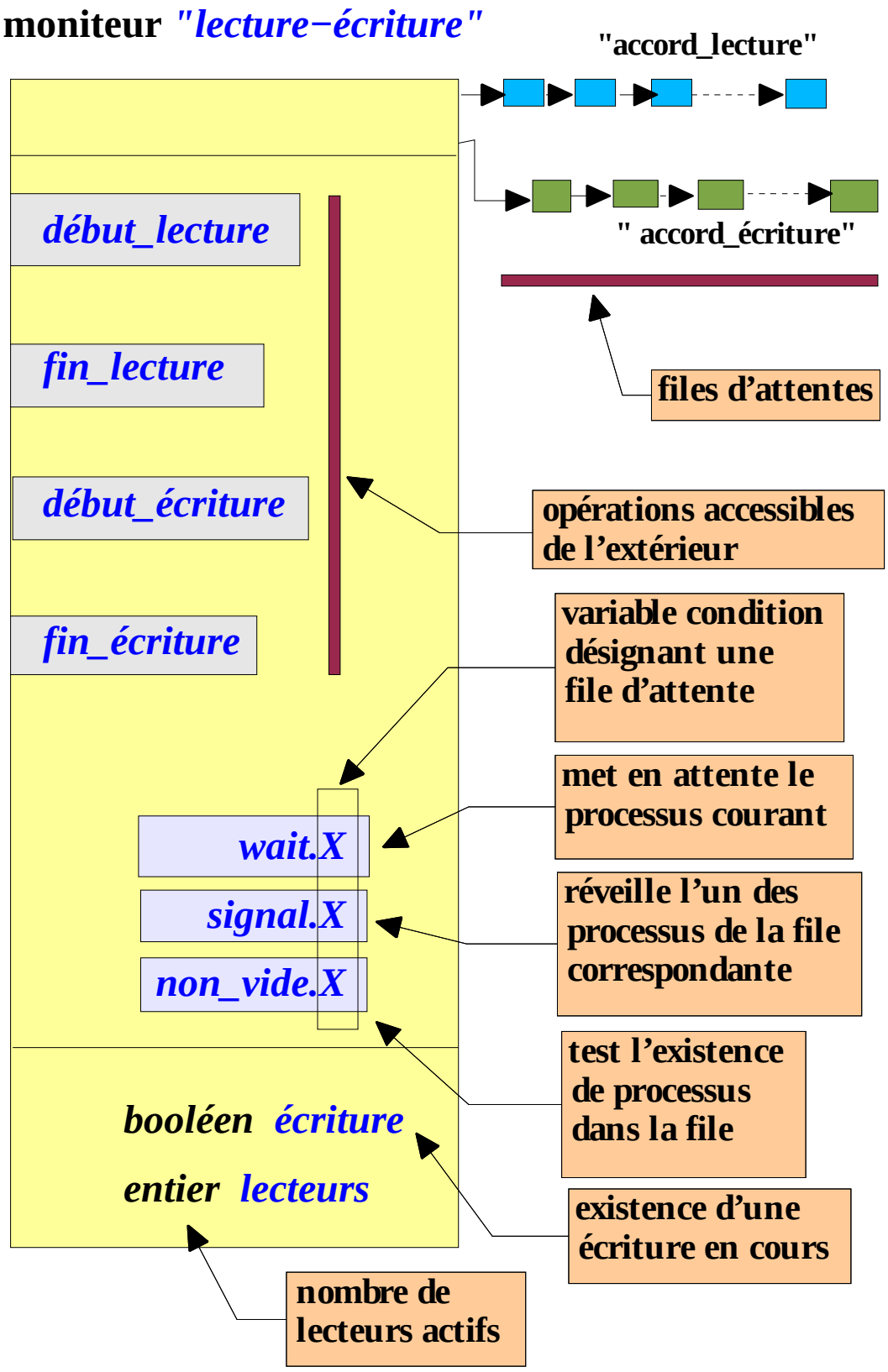
Pour contrôler la transmission des priorités le moniteur contient quatre opérations : ***début_lecture***, ***fin_lecture***, ***début_écriture*** et ***fin_écriture***.

Les files d'attente de lecteurs et de rédacteurs sont deux variables de type condition, désignées par ***accord_lecture*** et ***accord_écriture***.

L'existence d'une écriture en cours est repérée par un booléen ***écriture***, tandis que le nombre de lecteurs actifs est un entier ***lecteurs***.

Une variable condition ***X*** désigne une file de processus en attente, manipulée exclusivement dans le moniteur par les opérations ***wait*** et ***signal***. Si un processus ayant appelé une procédure d'un moniteur, exécute une instruction ***X.wait*** il est mis en attente dans la file ***X***. L'exécution de ***X.signal*** par un processus déclenche le réveil d'un autre processus en attente dans la file ***X***.

On utilise une nouvelle primitive pour les variables de type condition, destinée à vérifier l'existence de processus dans la file d'attente : il s'agit d'une fonction booléenne qui a pour résultat vrai si la file d'attente est non vide. Pour une variable ***X***, on écrit : ***X.non_vide***


moniteur "lecture−écriture"
"accord_lecture"
" accord_écriture"
début_lecture
fin_lecture
début_écriture
fin_écriture
wait.X
signal.X
non_vide.X
booléen écriture
entier lecteurs
files d'attentes
opérations accessibles de l'extérieur
variable condition désignant une file d'attente
met en attente le processus courant
réveille l'un des processus de la file correspondante
test l'existence de processus dans la file
existence d'une écriture en cours
nombre de lecteurs actifs

```
moniteur  lecture_écriture ;  // déclaration du moniteur

export      début_lecture, fin_lecture,
            début_écriture, fin_écriture ; /* opérations
                                                accessibles  */

import      lecture_écriture
```

```
processus  lecteur  {

            lecture_écriture.début_lecture  ;
                        < lecture >
            lecture_écriture.fin_lecture  ;
      }
```

```
processus  écrivain  {

            lecture_écriture.début_écriture  ;
                        < écriture >
            lecture_écriture.fin_écriture  ;
      }
```

Le moniteur est le suivant :

```
type lecture_écriture = moniteur
    var  écriture: booléen;
         lecteurs: entier;
         accord_lecture, accord_écriture: condition;
    procédure début_lecture;
         début
              si écriture ou accord_écriture.non_vide
                   alors accord_lecture.wait
              findesi;
              lecteurs:= lecteurs + 1;
              accord_lecture.signal
         fin;
```

```
    procédure fin_lecture;
         début
              lecteurs:= lecteurs − 1;
              si lecteurs = 0 alors accord_écriture.signal
              findesi
         fin;
```

(suite)

```
procédure début_écriture;
        début
                si lecteurs > 0 ou écriture
                        alors accord_écriture.wait
                findesi;
                écriture:= vrai
        fin;
```

```
procédure fin_écriture;
        début
                écriture:= faux;
                si accord_lecture.non_vide
                        alors accord_lecture.signal
                        sinon accord_écriture.signal
                findesi
        fin;
```

```
début {du moniteur}
        écriture:= faux;
        lecteurs:= 0
fin;
```

Un tel moniteur n'est efficace que si tout processus désirant effectuer une écriture, utilise la séquence suivante :

```
début_écriture
<écriture>
fin_écriture;
```

idem pour les opérations de lecture.

Il existe un phénomène de *réactions en chaîne*. Lorsqu'un processus rédacteur exécute la procédure ***fin_écriture*** du moniteur et que des processus lecteurs sont en attente, l'un d'entre eux est réveillé par l'instruction ***accord_lecture.signal***.

Celui−ci reprend la procédure ***début_lecture*** à l'instruction qui incrémente le nombre de lecteurs actifs, puis réveille (s'il existe) un second processus lecteur en attente. Ces réveils en cascade se poursuivent jusqu'à ce que tous les processus lecteurs en attente soient devenus actifs.

Pendant toute la durée de ces opérations, un processus quelconque, lecteur ou rédacteur, est bloqué à l'entrée du moniteur.

Ensuite, s'il n'y a pas de rédacteur en attente, des processus lecteurs peuvent à nouveau entrer dans le moniteur et se joindre aux opérations de lecture, jusqu'à l'arrivée d'un rédacteur.

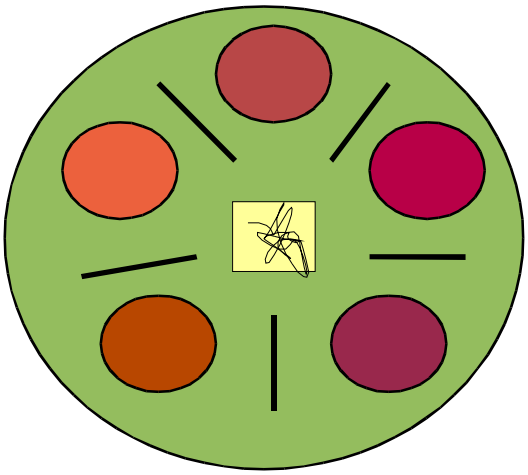
Le problème des philosophes (Dikkstra 1965)

5 philosophes sont assis autour d'une table pour parler et manger. Sur la table il y a 5 assiettes (une par philosophe), 5 baguettes (une entre deux assiettes voisines) et un plat de riz jamais vide.

Quand un philosophe parle, il ne tient aucune baguette.

Si un philosophe désire manger il doit prendre séquentiellement les 2 baguettes bordant son assiette. Dès qu'il a fini de manger, il repose les 2 baguettes.

Lorsqu'un philosophe veut manger et que l'un de ses deux voisins est en train de manger, il ne peut prendre les



deux baguettes qui bordent son assiette. L'opération devient impossible.

Si chacun des philosophes prend la baguette située à sa gauche et attend que l'autre baguette soit libre, ils sont **tous bloqués**. Si deux philosophes entourant un philosophe, s'entendent pour manger alternativement, le philosophe entouré **meurt de faim.**