

INTRODUCTION

- ▶ Comment un processus fait-il pour passer des informations à un autre processus?
Par exemple, dans le cas de processus qui coopèrent à réaliser une tâche, l'échange d'information est indispensable.
- ▶ Comment éviter les conflits lorsque des processus s'engagent dans les activités critiques?
Exemple : manipulation de données communes
- ▶ Quel séquencement doit être respecté en cas de dépendance des opérations de manipulation?

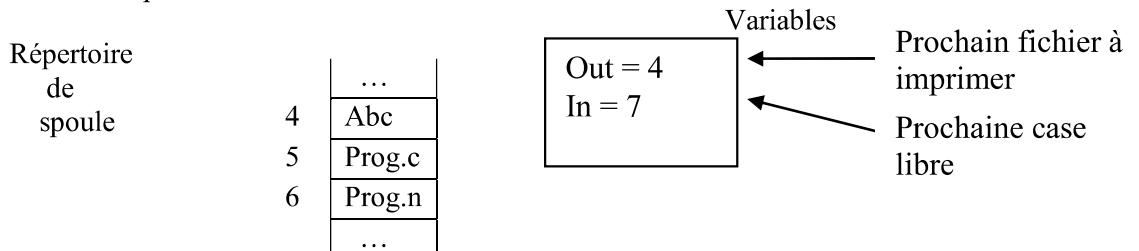
1/ CONDITIONS DE CONCURRENCE

Les processus peuvent partager un espace de stockage commun en lecture/écriture. Cet espace commun peut se trouver en M.C (partie des structures de données du noyau ou fichier partagé). Le partage d'un espace commun permet la communication mais pose des problèmes.

Exemple : le spoule d'impression.

Un processus qui veut imprimer un fichier, met le nom de ce dernier dans un répertoire de spoule spécial.

Le processus demon d'impression consulte périodiquement le spool pour vérifier s'il y a des fichiers à imprimer.



- 1 - Supposant que deux processus A et B accèdent presque simultanément à la variable **In**.
 - 2 - Les deux processus sauront que la prochaine case libre est la case 7.
 - 3 - Dans ce cas B peut stocker le nom du fichier qu'il veut imprimer dans l'entrée 7 et après il incrémente In à 8.
 - 4 - Le processus A qui avait stocké localement l'information que la prochaine entrée libre est 7, va écrire le nom du fichier à imprimer à cette entrée et écraser le nom de fichier écrit par B.
 - 5 - A ayant sauvegardé 7 pour valeur de In va incrémenter In à 8.
 - 6 - Le spoule d'impression fonctionne sans problème apparent sauf que l'utilisateur de B n'aura jamais sa sortie.
- On remarque donc que le résultat d'exécution dépend du séquencement d'exécution des opérations sur des éléments partagés. On parle de **Conditions de concurrence**.

2/ Les sections critiques :

Comment éviter ces conditions de concurrence ?

⇒ Interdire la manipulation simultanée des données partagées

Comment ?

→ Une solution : L' Exclusion Mutuelle (EM)

Qui permet d'assurer qu'un processus réalise une activité en interdisant aux autres processus d'effectuer la même activité.

Parfois l'EM assure à certains processus d'effectuer une tâche en interdisant à d'autres d'effectuer la même tâche (exemple: plusieurs lectures sur un fichier mais pas avec des écritures)

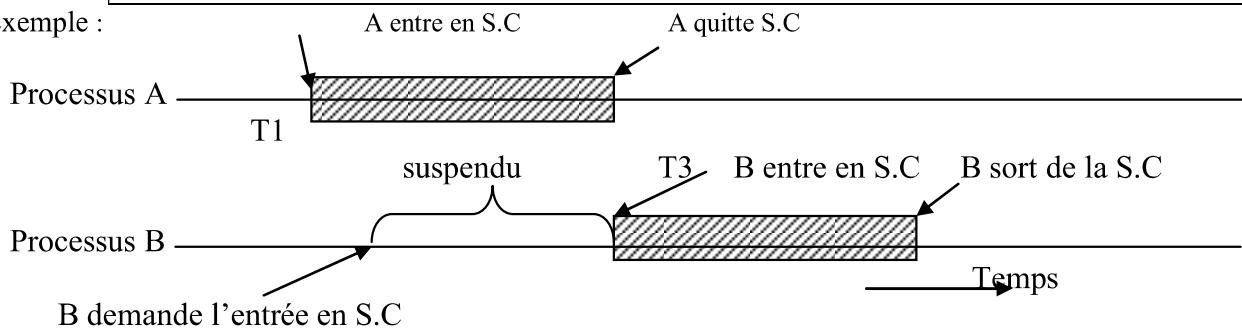
Le problème précédent est apparu car B avait commencé à travailler avec une variable partagée avant que A ne termine son travail avec la même variable.

La partie d'un programme qui manipule une mémoire commune ou fichier commun (globalement une ressource commune) est nommée : REGION CRITIQUE ou SECTION CRITIQUE (SC).

Empêcher deux processus de se trouver en même temps dans la même section critique

- ⇒ Eviter les conditions de concurrence
- ⇒ Une utilisation efficace des S.C qui permet une coopération des processus exige les conditions suivantes :
 - 1 - Deux processus ne doivent pas être simultanément dans une S.C.
 - 2 - Il ne faut pas faire de suppositions qu'en au nombre ou à la vitesse des processus.
 - 3 - Aucun processus s'exécutant à l'extérieur de sa S.C ne doit bloquer d'autres processus.
 - 4 - Aucun processus ne doit attendre indéfiniment pour entrer en S.C.

Exemple :



3/ EXCLUSION MUTUELLE avec attente active

3.a Masquage des interruptions

Il s'agit pour un processus de désactiver tout le système d'interruption (SI) à son entrée en S.C puisque ainsi le système ne peut basculer vers un autre processus (même l'interruption horloge est désactivée).

Toute fois, cette solution n'est pas très appropriée car cette opération (Activation/Désactivation du S.I) est souvent utilisée par le S.E.

Laisser des processus utilisateurs effectuer de telles opérations peut s'avérer **dangereux** (et si un processus ne réactive pas le système d'interruption !).

3.b Variable de Verrou

Soit une variable unique partagée dite verrou initialisée à 0.

Un processus qui désire entrer en S.C le met à 1.

Si le verrou est à 1, un processus attend qu'il passe à 0.

Ainsi,

Verrou : 0 → S.C Non occupée → accès possible

Verrou : 1 → S.C occupée → accès non possible - attente

Chaque processus qui veut exécuter la SC, doit exécuter le code ci dessous

```
while (verrou==1) {}
verrou = 1
SC();
verrou = 0
```

Mais, nous retrouvons le même problème posé, précédemment, dans le cas du spool d'impression présenté

(A lit le verrou = 0,
B lit le verrou =0,
A entre en S.C, ?!
B entre en S.C !!)

3.c Alternance stricte (Tour)

Les processus sont suivis par une variable.

Un processus qui tente d'entrer en SC, teste cette variable.

Si ce processus trouve cette variable égale à son numéro,

→ il rentre en SC, et la mis à jour à sa sorti de SC .

Un processus qui teste cette variable qui la trouve différente de son numéro va devoir attendre (Attente active : test sur la variable tour)

SPIN LOCK

```
While (TRUE)
{ while (tour !=0) /*loop */
  S.C
  Tour= 1 ;
  Région non critique
}
```

```
While (TRUE)
{ while (tour !=1) /*loop */
  S.C
  Tour= 0 ;
  Région non critique
}
```

Cependant, cette solution ne respecte pas la 3^{ème} condition énoncée au dessus.

En effet, une fois que le processus 0 a quitté sa S.C., **tour** prend la valeur 1 et ce processus est bloqué s'il désire exécuter la SC alors que le processus 1 n'est pas en S.C.

3.d/ Solution de Peterson

Dans cette solution on essaye de remédier au problème précédent.
Deux primitives sont définies enter_region et leave_region.

```
# define FALSE 0
#define TRUE 1
#define N 2      /* nb de processus*/
Int turn ;    /* à qui le tour */
Int interested [N] ; /* toutes les variables initialement à 0 (false)*/

Void enter-region (int process) /* le processus est 0 ou 1*/
{
    Int other ;                      /* nombre des autres processus */
    Other= 1-process;                /* à l'opposé du processus*/
    Interested [process] = TRUE ;    /* n=montre que vous êtes intéressé*/
    Turn= process ;
    While (turn==other && interested [other]==TRUE) /* instruction null*/
}

Void leave-region (int-process) /* processus qui quitte*/
{
    Interested[process]=FALSE;       /* indique le départ de la S.C */
}
```

Ainsi

```
Un processus i
{
    Enter-region(i);
    S.C
    Leave-region(i);
}
```

L'appel de la primitive enter-region, met le processus en attente si la S.C est occupée.

Les deux variables turn et interested permettent d'alterner l'entrée en S.C pour les processus en évitant qu'un processus soit empêché d'entrer en S.C alors que l'autre processus n'est pas intéressé ou et loin de la S.C. Par exemple, le processus 0 ne rentre pas en SC dès que le processus 1 émet le vœux de rentrer en SC même si c'est le tour de 0.

Solution : difficile à généraliser
De plus, Attente active (==> CPU gaspillé)

Exemple d'exécution sur monoprocesseur :

1/

Process P0	Process P1
■ Other= 1;	
■ Interested [0] = TRUE ;	
■ Turn= 0 ;	
■ While (turn=0 && interested [1]==TRUE) //F	
■ SC	■ Other= 0;
■ Interested[0]=FALSE	■ Interested [1] = TRUE ;
	■ Turn= 1;
	■ While (turn=1 && interested [0]==TRUE) //True-Boucle
	Attente Active
	■ SC
	■ Interested[1]=FALSE

Process P0	Process P1
■ Other= 1;	
■ Interested [0] = TRUE ;	
■ Turn= 0 ;	
■ While (turn=0 && interested [1]=TRUE) //F	
■ SC	■ While (turn=1 && interested [0]=TRUE) //True-Boucle
■ Interested[0]=FALSE	////////Attente Active
	■ SC
	■ Interested[1]=FALSE

Dans ce dernier cas, turn est à 1 en dernier mais le processus 1 n'entrera pas en SC, car P0 (l'autre) a exprimer le fait qu'il veut rentrer en SC

3.e Instruction TSL (Test and Set Lock)

C'est une solution qui fait appelle au matériel et qui se trouve sur certains ordinateurs et notamment les multiprocesseurs. Cette instruction lit le contenu du mot mémoire locale dans un registre RX, puis stocke une valeur différente de 0 dans le mot mémoire Lock.

Les opérations : lire le mot + stocher la valeur sont **INDIVISIBLES** (aucun processus ne peut accéder au mot mémoire tant que l'opération n'est pas terminée)

Lorsque lock est à 0 n'importe quel processus peut le positionner à 1 via l'instruction TSL, puis lire ou écrire dans la mémoire partagée. Après, le processus repositionne lock à 0.

TSL RX, Lock Manipulation mémoire partagée Move, Lock, #0

Un processus qui désire entrer en S.C

- 1) copie la valeur de lock dans registre,
- 2) la fait passer à 1 et compare l'ancienne valeur à 0.
- 3) Si elle est égale à 0, il entre en S.C
- 4) sinon cela signifie que lock était déjà positionnée,
le programme boucle alors sur le test en attendant que lock repasse à 0.

Entrer-region	Leave-region
TSL register, LOCK, CMP, register !=0, JNE enter-region RET /* retourne à l'appel L, entre en S.C */	MOVE LOCK , !=0 /* stocke 0 dans LOCK *// RET /* retourne à l'appelant */

4/ EXCLUSION MUTUELLE avec attente passive (sleep & wakeup)

Toutes les solutions précédentes font appel à l'attente active

Problème 1 ==> !!Consommation de temps CP

Problème 2 ==> Cas de deux processus H et L avec Priorité (H)> priorité(L) !!!

Supposant que L est en S.C et que à ce moment H devient prêt (fint opération E/S),

H entre en attente active pour entrer en S.C mais vu que H est plus prioritaire, L n'aura jamais l'occasion de s'exécuter et de sortir de sa S.C et H boucle sans fin (inversion des priorités)

Pour éviter ces attentes actives, on peut utiliser des primitives qui bloquent et réveillent le processus

4.a / Sleep & Wakcup : Problème du producteur-consommateur

Deux processus partagent un tampon commun de taille fixe, l'un deux, le producteur place des informations dans le tampon; l'autre, le consommateur, les récupère.

Le problème : Le producteur désire déposer un élément alors que le tampon est plein

➔ **Solution :**

Le producteur entre en sommeil et il est réveillé lorsque le consommateur aura consommé 1 élément

Le problème est similaire lorsque le consommateur trouve le tampon vide.

Cette solution paraît simple mais peut conduire à des conditions de concurrences.

Cette solution nécessite une variable **count** qui compte le nombre d'éléments déposés. Si count=N alors

Le tampon est plein → producteur entre en sommeil

Si count=0 → consommateur entre en sommeil.

# dfine N 100 Int count = 0	
Void producer (void) <pre>Int item ; While (TRUE) { Item= produre-item /* générer element sv*/ If (count == N) sleep(); Insert_item(item); Count=count+1; If (count==1) wakcup(consumer) }}</pre>	Void consumer(void) <pre>{ Int item ; While (TRUE) { If (count == 0) sleep(); Item= remove-item Count=count-1; If (count==N-1) wakcup(producer); Consume-item(item); }}</pre>

Conditions de concurrence:

Le consommateur

- consomme
- Met à 0
- et à cet instant, l'ordonnanceur lui enlève la main et lance le producteur.
- Ce dernier produit un élément, incrémenté count, il constate qu'il est à 1
➔ Reveille le consommateur ➔ signal wakeup PERDU

Lorsque le consommateur s'exécute, il se met en sommeil puisque count=0 pour lui.

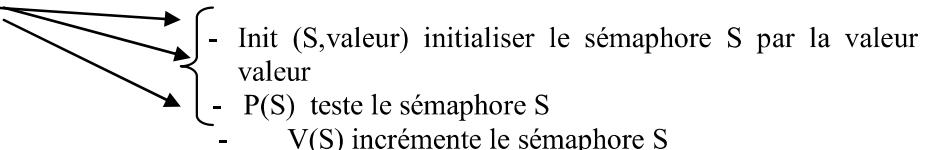
Lorsque le tampon sera plein -> le producteur en sommeil

➔ Les deux resteront en sommeil pour toujours.

4.b/ Les sémaphores

1. Un sémaphore est une variable entière qui indique le **nombre d'autorisations** d'entrée en S.C.
2. Les opérations de vérification, de modification et de mise en sommeil sont des **actions atomiques** (important).
3. A chaque sémaphore est associé une F.A qui contient les processus bloqués qui n'ont pas réussi à entrer en S.C.

Trois primitives Atomiques



P(S)	V(S)
<pre>{ S-- Si S < 0 alors { Bloquer le processus dans F.A. de S } // Sinon Continuer l'exécution }</pre>	<pre>{ S++ Si FA de S non vide Alors { réveille tête-file } }</pre>

- ♦ Les opérations **Init(S,Valeur)**, **P** et **S** sont implémenté comme des **appels systèmes**.
- ♦ Le système désactive le S. Interruption pour assurer l'atomicité de ces opérations.
Etant donné que ces opérations sont très courtes (pas d'inconvénients).
- ♦ En cas de plusieurs processus → on peut utiliser l'instruction TSL pour protéger chaque sémaphore.

Exemple :Semaphore d'EM

Mise à jour d'un compteur par plusieurs processus

Code :.....

Exemple : Semaphore compteur

Porte d'accès d'un ascenseur qui peut accepter jusqu'à 3 personnes à la fois.

Code:

Exemple : Semaphore Bloquant (privé)

P0 doit calculer $r = (x+y)$ et, par la suite, P1 doit calculer $r = r*2$.

4.b.a/ Problème du producteur consommateur

Nous considérons trois variables :

Full : compter nb d'emplacements occupés.

Empty : compter nb d'emplacements vides.

Mutex : pour assurer que le producteur et le consommateur n'accèdent pas simultanément.

//Initialisation

full = 0

mutex = 1 pour assurer qu'un seul processus puisse entrer en S.C (**SEMAPHORES BINAIRES**)

define N 100

TypeDef int semaphore;

Sémaphore mutex = 1 ; Sémaphore empty = N ; Sémaphore full = 0 ;

Void producer (void)

```
{ int item
  While (TRUE)
    {   Item = produce-item();      /* génère un élément à placer */
        P(&empty);             /* décrémente le compteur des emplacements vides */
        P(&mutex);              /* entre en section critique */
        Insert-item(item);       /* place un nouvel élément dans le tampon */
        V(&mutex);              /* quitte la section critique */
        V(&full);                /* incrémente le décompte des emplacements pleins
    }
}
```

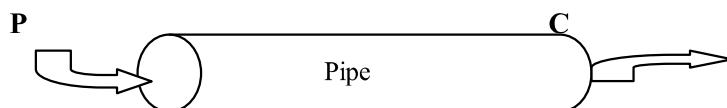
Void consumer (void)

```
{Int item ;
While (TRUE) { /* boucle sans fin */
    P(&full); /* décrémente le nb d'emplacements pleins */
    P(&mutex);
    Item = remove-item();
    V(&mutex);
    V(&empty);
    Consume-item(item);
}}
```

mutex est un semaphore binaire c.à.d une version simplifiée de sémaphore qui ne peut avoir que 2 états : déverrouillé ou verrouillé. Un seul bit est donc suffisant.

5. Communication et synchronisation par PIPE UNIX

Un pipe est utilisé comme un type particulier de fichiers pour l'échange d'information et la communication entre processus interne d'un fichier. Ainsi, un processus P peut déposer des données ou informations dans un pipe et ces informations seront récupérées par un processus C.



- Un processus qui veut lire une information à partir d'un pipe, se bloque jusqu'à la disponibilité de cette information (réception bloquante - synchrone)
 - Un pipe comporte un point d'accès en écriture et un point d'accès en lecture.
 - Une information récupérée par C est éliminée du pipe automatiquement.

Nous avons deux types de pipes :

- les pipes non nommés partagés par les processus parents.
 - les pipes nommés employés pour les communications entre processus sans liens de parenté. Ils s'apparentent à des fichiers en mémoire.

Exemple : Pipes non nommés

Primitives :

`pipe()` : primitive de création d'un pipe.

`close(p[0])` : primitive qui ferme le pipe en **lecture**.

`write(p[1])` : primitive qui permet d'écrire sur un pipe. Elle comporte trois arguments

1. la structure d'écriture soit l'entrée du pipe
 2. la chaîne à transmettre
 3. le nombre de caractères de la chaîne.

Exemple : Pipes nommés

Le pipe est utilisé comme un fichier (open, close ...).

Les pipes nommés sont utilisés de manière similaire à l'utilisation des fichiers sauf que le pipe est créé avant les deux processus.

mknod p : commande qui crée le pipe d'identificateur p.

Ecriture

```
#include <stdio.h>          /* Ecriture non bloquante dans pipe nomme */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
main()
{
    int d, i, n;
    char buf[2000];
    /* pipe créé par mknod mon_pipe pi */
    /* O_RDWR   lecture/écriture
       O_NDELAY non bloquant
       2 signifie droit d'écriture (convention Unix)
*/
    d = open("mon_pipe",O_RDWR|O_NDELAY,2);
    if (d < 0){ fprintf(stderr,"problème d'ouverture du pipe\n");
        exit(1); }
    for (i=0; i<2000; i++){ buf[i]='\n'; }      /* remplissage du pipe */
    if ((n = write(d,buf,2000))>0) fprintf(stdout,"Ecriture de %d caractères\n",n);
    sleep(20);
    exit(0);
}
```

Lecture

```
/* lecture bloquante dans pipe nomme mon_pipe */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
main()
{
    int d,i,n;
    char buf[2000];
    /* open du pipe mon_pipe lecture bloquante
       O_RDONLY  lecture seulement 4 droit de lecture (convention Unix)
*/
    d = open("mon_pipe",O_RDONLY,4);
    if (d < 0){
        fprintf(stderr,"Erreur ouverture du pipe\n");
        exit(1); }
    fprintf(stdout,"pipe ouvert\n");

    n = read(d,buf, 2000);
    fprintf(stdout,"Lecture de %d caractères\n",n);
    fprintf(stdout,"Les 20 premiers sont\n");
    for (i=0;i<20;i++){ fprintf(stdout,"%c",buf[i]); }

    fprintf(stdout,"\n");
    exit(0);
}
```

La lecture est bloquante donc le processus attendra l'arrivée de 2000 caractères avant de passer à l'instruction qui suit le read.

6. Communication et synchronisation par signaux

Un signal peut être : la fin d'un délai, l'arrivée d'une interruption,...

Un processus peut définir son comportement à partir des signaux reçus.

Un signal est traité à son arrivée tout comme une interruption logicielle :

- save CTXT
- Traitement correspondant
- Restaurer CTXT

7. Communication et synchronisation échange de message:

Les solutions précédentes ne sont pas adaptées à des systèmes distribués où des processus peuvent s'exécuter sur des machines différentes. D'autres parts, elles ne permettent pas l'échange d'information. Deux primitives sont utilisées : SEND & RECEIVE

SEND et RECEIVE sont des appels systèmes:

SEND (Destination, message)

RECEIVE (Source, message)

En absence de message, le récepteur peut se bloquer jusqu'à l'arrivée du message (Réception synchrone) ou il peut continuer (Réception asynchrone)

Parmi les problèmes posés par l'échange de messages : la perte de message

→ **Solution** : utilisation des accusés de réception (ACK)

Passé un délai l'émetteur retransmet le message s'il n'a pas reçu d'ACK

→ **Pb** : si c'est le premier ACK qui a été perdu et non le message

→ **Solution** : les messages sont numérotés pour distinguer les différents messages. 1 message comportant un numéro déjà reçu est désigné comme double

Autre problème : L'authentification

Comment assurer qu'un client communique à coup sûr avec le bon serveur de fichier et non un imposteur

Pour plus d'efficacité, la taille d'un message doit être limitée

Exemple : Producteur – consommateur avec échange de messages

Hypothèses :

- Tous les messages sont de même taille
- Les messages émis et non reçus sont mis en mémoires tampons par le S.E.
- N messages dans le système

Principe :

- Le consommateur commence par émettre N messages vides vers le producteur.
- Chaque fois que le producteur désire produire un nouvel élément pour le consommateur, il prend un message vide et renvoie un message plein. Ainsi le nombre total de message reste constant.
- Si le producteur travaille plus vite que le consommateur → le producteur bloqué
- Si c'est le consommateur qui travaille plus vite → bloqué le consommateur

<pre># define N 100 Void producer (void) {int item ; Message m ; While (TRUE) { Item = producer-item() ; Receive(producer, &m) ; Build-message(&m,item) ; Send(producer, &m) }}</pre>	<pre>Void consumer (void) {int item, i ; Message m ; For (i=0 ;i<N ;i++) send (producer,&m); /* envoie message vides */ While (TRUE) { Receive (producer, &m); Item = extrait-item(item,&m); Send (producer,&m); /* retourne mess vide*/ Consumer-item(item) ; }}</pre>
--	---

8/ Communication et Synchronisation par Moniteurs

Le concept de moniteur a été introduit par Brinch & Hansen en 1972 et Hoare en 1974.

Un moniteur est composé de :

- un ensemble de variables d'état,
- des conditions (ou variables de type **condition**),
- des procédures externes (pour l'accès)
- des procédures internes (appeler uniquement à l'intérieur du moniteur),
- des primitives de synchronisation
- et, des files d'attentes.

```
Type m : moniteur
Var // Déclaration des variables locales (ressources partagées);
      .... //par exp, c : condition ; i : int ;
//Déclaration et corps des procédures du moniteur (points d'entrée);
Procedure idfl() ;
    Debut
    ...
    Fin;
    ....
Procedure idfn() ;
    Debut
    ...
    Fin;
Début
    Initialisation des variables locales;
Fin
```

Ces éléments sont manipulés avec des règles strictes :

- Une variable d'état ne peut être manipulée que par une procédure **externe**.
- toute procédure du moniteur s'exécute en **exclusion mutuelle**.
- On ne peut avoir qu'un seul processus actif à la fois au sein d'un moniteur,

Les primitives de synchronisation associées à un moniteur sont : les primitives **wait** et **signaler** permettent de bloquer ou de réveiller un processus sur une condition. Une condition est une variable qui n'a pas de valeur mais qui est de type condition. Soit **c** une condition.

c.wait	bloque le processus et le place en attente de la satisfaction de la condition c . c.wait bloque automatiquement le processus alors que la primitive P() vue pour les semaphores ne le fait pas nécessairement.
kc.vide	vrai si aucun processus n'est en attente de c , faux sinon.
c.signal	Si c.vide non vraie alors réveiller un processus en attente de c sinon elle n'a aucun effet. La primitive V() vue pour les semaphores a toujours un effet puisqu'elle exécute au moins une opération d'incrémentation.

Exemples / Exercices

Ex1 : On veut offrir des procédures de mise à jour d'une variable avec la condition que la variable doit être manipulée en EM. Proposez un moniteur pour ces procédures.

Ex2 : On veut traiter le problème de Rendez-vous de N processus en utilisant les moniteurs.

Ex3 : Reprendre le problème des Producteurs Consommateurs avec les moniteurs. Nous supposons N tampons.

Ex4 : Problème des lecteurs-rédacteurs

Donner une solution à base de moniteur pour le problème des Lecteurs /Rédacteurs. Traiter les cas, où la priorité est donnée aux lecteurs.

Corrigé

Ex1 : Opérations de manipulation d'une variable

<pre> incr_decr : moniteur; var i : entier; procédure incrément; début i := i + 1 ; fin; </pre>	<pre> procédure décrémente; début i := i - 1 ; fin ; début i := 0 ; fin fin incr_decr </pre>
--	---

Ex 2: Rendez-vous entre N processus

<pre> rendez_vous : moniteur ; var n : entier ; tous_là : condition ; . .</pre>	<pre> procédure arriver ; début n := n + 1 ; si n < N alors tous_là.wait ; tous_là.signal; fin ; </pre>	<pre> début n := 0 ; fin fin rendez_vous. </pre>
--	--	--

Ex 3: Producteur et consommateurs avec N tampons

<pre> tampon : moniteur ; var n : 0..N ; NonPlein, NonVide : condition ; type message : ... ; var file: tableau[0..N-1] message ; tête, queue : 0..N-1 ; </pre>	<pre> procédure déposer(m : message) ; début si n = N alors NonPlein.wait ; n := n + 1 ; entrer(m) ; NonVide.signal; fin ; procédure retirer(var m:message) ; début si n = 0 alors NonVide.wait; sortir(m) ; n := n - 1 ; NonPlein.signal ; fin ; </pre>	<pre> procédure entrer(m : message) ; début file[queue] := m ; queue := (queue + 1) mod N ; fin ; procédure sortir(var m:message); début m := file[tête] ; tête := (tête + 1) mod N ; fin ; </pre>	<pre> début n := 0 ; tête := 0 ; queue := 0 ; fin fin tampon. </pre>
--	--	--	--

Ex4 : Problème des lecteurs-rédacteurs

Donner une solution à base de moniteur pour le problème des Lecteurs /Rédacteurs. Traiter les cas, où la priorité est donnée aux lecteurs.

<pre> lecteur_rédacteur : moniteur ; var ecr : booléen ; nl : entier ; //nl : Nb Lecteurs c_ecr, c_lect : condition ; </pre>	<pre> procédure début_lire ; début nl := nl + 1; si ecr ou nl > 0 alors c_lect.wait ; c_lect.signal ; fin ;procédure fin_lire ; début nl := nl - 1; si nl = 0 alors c_ecr.signal ; fin ; </pre>	<pre> procédure début_écrire ; début si ecr ou nl > 0 alors c_ecr.wait ; ecr := vrai ; fin ; procédure fin_écrire ; début ecr := faux ; si nl > 0 alors c_lect.signal ; sinon c_ecr.signal ; fin ; </pre>	<pre> début ecr := faux ; nl := 0; fin fin lecteur_rédacteur. </pre>
---	---	--	--