5.c) Primitives SLEEP et WAKEUP

SLEEP() est un appel système qui suspend l'appelant en attendant qu'un autre le réveille. WAKEUP(processus) est un appel système qui réveille un processus. Par exemple, un processus H qui veut entrer dans sa section critique est suspendu si un autre processus B est déjà dans sa section critique. Le processus H sera réveillé par le processus B, lorsqu'il quitte la section critique.
Problème : Si les tests et les mises à jour des conditions d'entrée en section critique ne sont pas exécutésen exclusion mutuelle , des problèmes peuvent survenir. Si le signal émis par WAKEUP() arrive avant que le destinataire ne soit endormi, le processus peut dormir pour toujours.
Exemples

```
// Processus P1
{
        If (cond ==0 ) SLEEP()
        ; cond = 0 ;
        section_critique ( ) ;
        cond = 1 ;
        WAKEUP(P2) ;
        ...
}

// Processus P2
{
        if (cond ==0 ) SLEEP() ;
        cond = 0 ;
        section_critique ( ) ;
        cond = 1 ;
        WAKEUP(P1) ;
        ...
}
```

Cas 1 : Supposons que :
– P1 est exécuté et il est suspendu juste après avoir lu la valeur de cond qui est égale à 1.
– P2 est élu et exécuté. Il entre dans sa section critique et est suspendu avant de la quitter.
– P1 est ensuite exécuté. Il entre dans sa section critique.
– Les deux processus sont dans leurs sections critiques.

Cas 2 : Supposons que :
– P1 est exécuté en premier. Il entre dans sa section critique. Il est suspendu avant de la quitter ;
– P2 est élu. Il lit cond qui est égale à 0 et est suspendu avant de la tester.
– P1 est élu de nouveau. Il quitte la section critique, modifie la valeur de cond et envoie un signal auprocessus P2.
– Lorsque P2 est élu de nouveau, comme il n'est pas endormi, il ignorera le signal et va s'endormir pourtoujours.

Problème du producteur et du consommateur

Dans le problème du producteur et du consommateur deux processus partagent une mémoire tampon de taille fixe, comme montré à la figure 6.3. L'un d'entre eux, le producteur, met des informations dans la mémoire tampon, et l'autre, les retire. Le producteur peut produire uniquement si le tampon n'est pas plein. Le producteur doit être bloqué tant et aussi longtemps que le tampon est plein. Le consommateur peut retirer un objet du tampon uniquement si le tampon n'est pas vide. Le consommateur doit être bloqué tant et aussi longtemps que le tampon est vide. Les deux processus ne doivent pas accéder en même temps au tampon.
Exemple 1. Une solution classique au problème du producteur et du consommateur [?] est montré sur le programme schema-prod-cons.c avec l'utilisation des primitives SLEEP et WAKEUP.

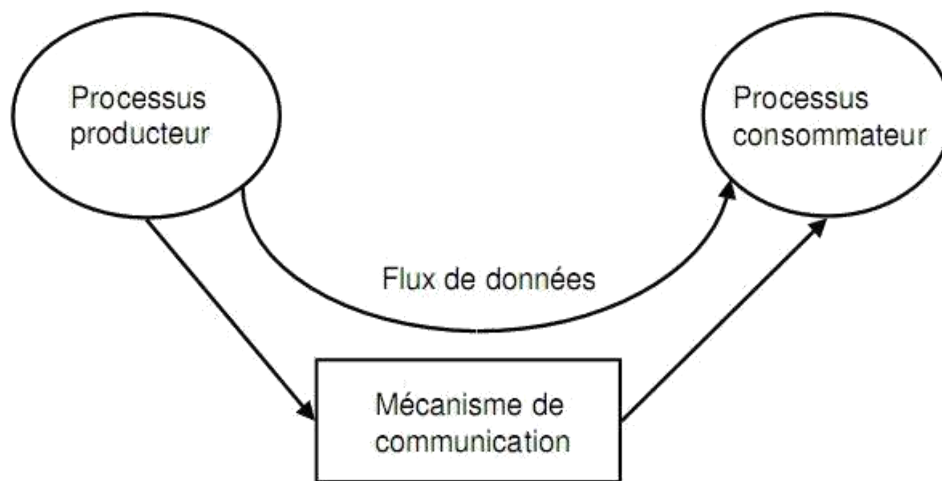


FIG. 2.3 – Problème du producteur/consommateur.

Listing 2.3 – schema-prod-cons.c

```
# define TRUE 1
# define N 50// Nb . de places dans le tampon
int compteur = 0 ;// Nb . objets dans le tampon

void producteur ( )
{
	while (TRUE)
	{
		produire_objet ( ) ;
		i f ( compteur == N)
			SLEEP ( ) ;			// Tampon plein
		mettre_objet ( ) ;	// dans tampon
		compteur ++;
		i f ( compteur = = 1 )  // Tampon vide
			WAKEUP( consomateur ) ;
	}
}

void consomateur ( )
{
	while (TRUE)
	{
		i f ( ! compteur )
			SLEEP ( ) ;		// tampon vide
		retirer_objet ( ) ;		// du tampon
		compteur --;
		i f ( compteur == N - 1)		// réveiller producteur
			WAKEUP( producteur ) ;
		consommer_objet ( ) ;
	}
}
```

Critique des solutions précédentes

La généralisation de toutes ces solutions aux cas de plusieurs processus est bien complexe. Le mélange d'exclusion mutuelle et de suspension est toujours délicat à réaliser et à implanter.

5-d) Les sémaphores

Pour contrôler les accès à un objet partagé, E. W. Dijkstra suggéra en 1965 l'emploi d'un nouveau type de variables appelées les sémaphores. Un sémaphore est un compteur entier qui désigne le nombre d'autorisations d'accès à une section critique. Il a donc un nom et une valeur initiale, par exemple semaphore S = 10.

Les sémaphores sont manipulés au moyen des opérations P ou wait et V ou signal. L'opération P(S) décrémente la valeur du sémaphore S si cette dernière est supérieure à 0. Sinon le processus appelant est mis en attente. Le test du sémaphore, le changement de sa valeur et la mise en attente éventuelle sont effectués en une seule opération atomique indivisible.

L'opération V(S) incrémente la valeur du sémaphore S, si aucun processus n'est bloqué par l'opération P(S). Sinon, l'un d'entre eux sera choisi et redeviendra prêt. V est aussi une opération indivisible.

Un sémaphore est un compteur d'accès à une ressource partagée, pouvant prendre toute valeur entière.

Par rapport à un entier, il présente trois particularités :

Il peut être initialisé à n'importe quelle valeur, mais ensuite on peut effectuer que deux opérations : P(décrémente de un) et V() (incrémente de un). Ces opérations sont garanties atomiques par l'OS. Quand un processus décrémente un sémaphore, si le résultat est négatif, le processus passe en mode bloqué et se met en attente dans une file d'attente liée au sémaphore.

Quand un processus incrémente un sémaphore qui a une valeur négative, un des processus en attente se réveille.

```
Opération P(s) :
    S :=s-1 ;
    Si s<0 alors
        Ajouter ce processus à la file et bloquer ce processus
    Fsi
```

```
Opération V(s)
    S :=s +1
    Si s<=0 alors
        Sortir un processus de la file et réveiller ce processus
    Fsi
```

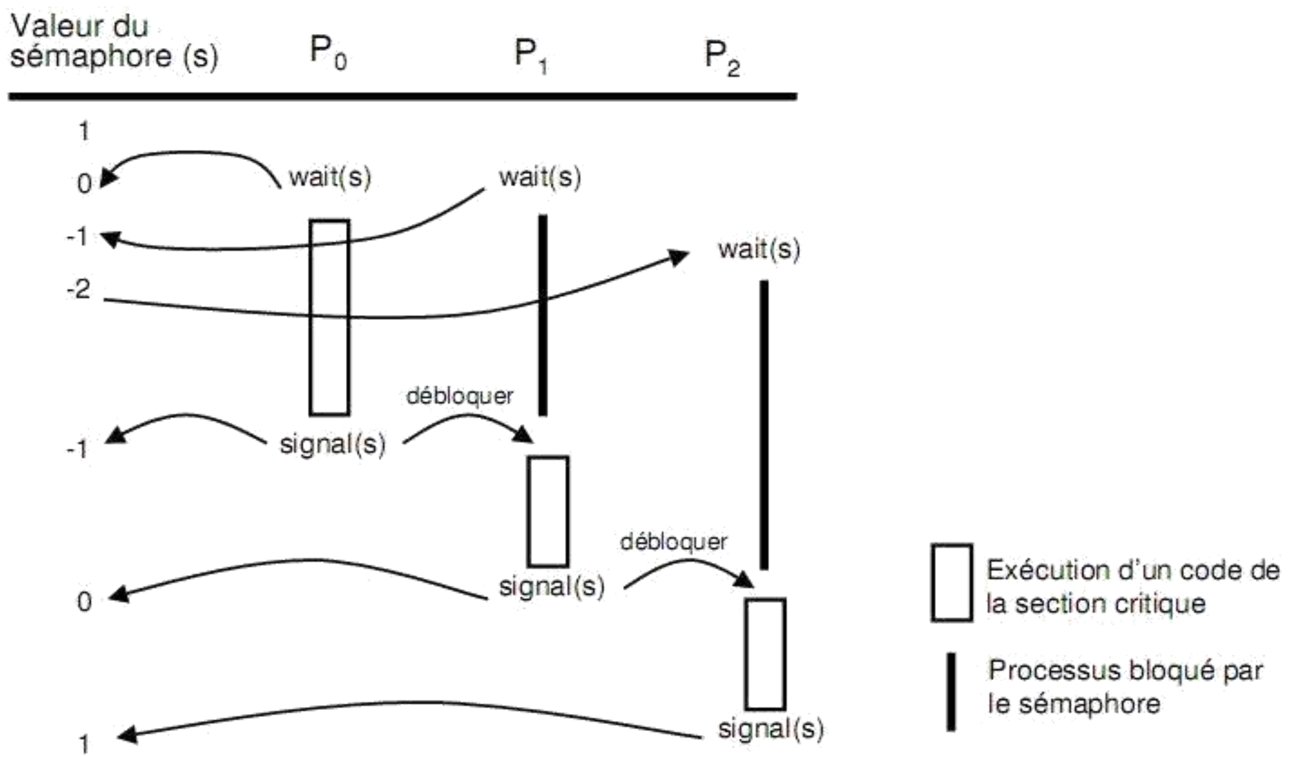
Les sémaphores permettent de réaliser des exclusions mutuelles, comme suit :

semaphore mutex=1 ;	
processus P1 : P(mutex) section critique de P1 ; V(mutex) ;	processus P2 ; P(mutex) section critique de P2 ; V(mutex) ;

Par exemple pour l'exclusion mutuelle d'une petite section critique a++, on écrit :

int a ; semaphore mutex=1 ;	
processus P1 : P(mutex) a++ ; V(mutex) ;	processus P2 : P(mutex) a++ ; V(mutex) ;

Sur la figure suivante, on montre la valeur que les sémaphores prennent lors des appels des P (wait) et V (signal), pour trois processus P0, P1 et P2.



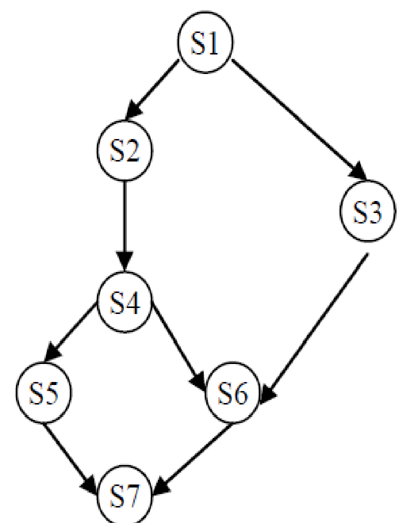
Exemples :

1) Soient deux processus p1 et p2 s'exécutent en parallèle où S1 est une instruction de p1 et s2 est une instruction de P2. Supposons qu'on désire exécuter S2 après la fin d'exécution de S1. On peut réaliser cela en utilisant un sémaphore appelé synch initialisé à 0.

P1	P2
. . S1 ; V(synch) ; .	. P(synch) ; S2 ; .

2) on peut utiliser plusieurs sémaphore pour ce graphe : Var a,b,c,d,e,f,g : sémaphore // initialisé à 0

parbegin
begin s1 ; V(a) ; V(b) ;end;
begin P(a) ; S2 ; S4; V(c) ; V(d); end; begin P(b); S3; V(e); end;
Begin P(c); S5; V(f); end; Begin P(d); P(e); s6; V(g); end Begin P(f);
P(g); s7; end; Parend



***Sémaphores privés**

Un sémaphore privé est un sémaphore dont l'opération P n'est accessible que par un seul processus, les autres pouvant exécuter uniquement la primitive V. Les sémaphores privés sont utilisés pour qu'un processus puisse se bloquer lui-même, généralement en utilisant des données de contrôle (protégées en section critique).

Exemple : schéma clients/serveur
Un processus serveur dont le rôle est d'offrir un service à des processus clients ne peut s'exécuter que sur demande des clients, il traite une seule demande à la fois. Contexte commun sémaphore spriv; (initial :0)

Processus serveur
tant que vrai faire
P(spriv); // service
...
fait;
	Processus client1
	...
	V(spriv);
	...
		Processus client2
		...
		V(spriv);
		...

Exemple : contrôle d'accès à un musée...
On considère un musée pour lequel les entrées et sorties sont réalisées par des portillons automatisés (avec le ticket acheté à l'entrée). On souhaite programmer ces portillons de manière à ce que :

- le nombre de visiteurs présents à l'intérieur du musée n'excède jamais 100 personnes,
- à tout instant, le nombre de visiteurs ayant visité le musée soit disponible.

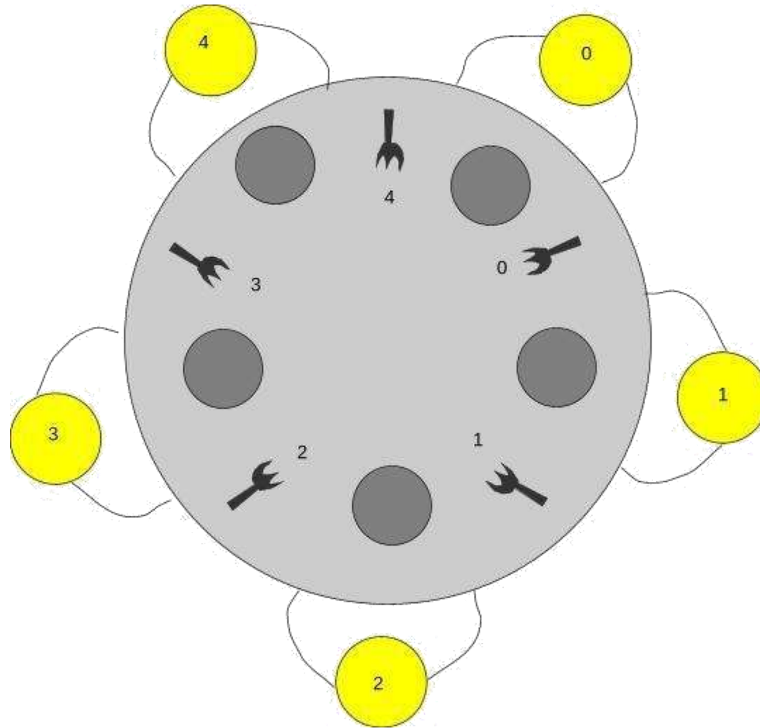
```
const Max_Visiteurs = 100;
MUTEX ( =1), SECURITE (= max_visiteurs): Semaphore;
Tot_Visites (=0) : integer;

processus ENTREE(num : entier);
while true do
        if ticket then
                P(SECURITE)
                P(MUTEX)
                Tot_Visites ++;
                V(MUTEX);

processus SORTIE(num : entier);
while true do
        if sortie then
                V(SECURITE);
```

Problème des philosophes avec les sémaphores

Introduction : N philosophes (ici 5) sont assis a une table circulaire, ils aimeraient manger leur plat de spaghettis avec 2 fourchettes chacun mais il n'y a que N fourchettes. Evidemment ils ne pourront pas tous manger en même temps. Il s'agit d'écrire un outil de synchronisation pour ce problème.




```
Philo(i){
        while(1){
                pense( ); prendre_fourchette( i );
                manger ( ); poser_fourchette( i );
        }
}
```

Il s'agit donc d'écrire les procédures prendre_fourchette et poser_fourchette.

Première solution :On prend un sémaphore par fourchetteChaque sémaphore est initialisé à 1 sem

fourch[N] = {1,........1}

```
prendre_fourchette ( int i ){
        P(fourch[ i ]);                 //demande/prend de la fourchette de gauche
        P(fourch[ (i + 1) % N ]);       //demande/prend la fourchette de droite
}
poser_fourchette ( int i ){
        V(fourch[ i ]);                 //pose de la fourchette de gauche
        V(fourch[ (i + 1) % N ]);       //pose la fourchette de droite
}
```

Cette implémentation pose un problème d'interblocage dans le cas où les N philosophes décident de manger donc d'appliquer la procédure prendre fourchette. En effet, tous les philosophes détiennent la fourchette qui est à leur droite et attendent que celle qui est a leur gauche se libère.

Deuxième solution :

On se centre ici sur les philosophes. Un sémaphore est attribué à chaque philosophe.

Etat = { PENSE , MANGE , A_FAIM}

Un philosophe qui veut prendre les fourchettes (donc manger) déclare qu'il a faim. Si l'un de ses deux voisins est en train de manger, il ne peut donc pas prendre les deux fourchettes pour l'instant et donc se met en attente. Si les deus philosophes a coté ne mangent pas alors il peut prendre les deux fourchettes et déclarer qu'il mange. Quand le philosophe a fini de manger, il déclare donc qu'il pense et regarde si ses deux voisins qui forcément ne pouvaient pas manger, en ont dès lors l'opportunité. Si c'est le cas il les réveille.

```
sem philo_prive [ N ]
Etat Etat_Philo[ N ] = { PENSE, ………, PENSE}
Semaphore mutex = 1                  //mutex = 1 pour que l'écriture dans le tableau   Etat_Philo se
                                       //fasse en exclusion mutuelle
        test_mange( int i ){
                if (Etat_Philo[ i ] == A_FAIM
                        &&  Etat_Philo[ (i+1)%N ] != MANGE
                        &&  Etat_Philo[ (i-1+N)%N ] != MANGE ) {

                        Etat_Philo[ i ] = MANGE;
                        V(philo_prive[ i ]);
                }
        }

        prendre_fourchette (int i){
                P(mutex);
                Etat_Philo[ i ] = A_FAIM;
                test_mange( i );
                V(mutex);
                P(philo_prive[ i ]);
        }

        poser_fourchette (int i){
                P(mutex);
                Etat_Philo[ i ] = PENSE;
                test_mange((i+1)%N );
                test_mange((i-1+N)%N );
                V(mutex);
        }
```