

Semaphores :

A semaphore is an integer variable that is accessed only through two standard atomic operations : $P()$ (wait()) and $V()$ (signal()).

$P(S) \{$ while $s \leq 0;$ $s--;$ $\}$	$V(S) \{$ $s++;$ $\}$
--	-----------------------------

An example of the usage of semaphores, let's consider two processes P_1 and P_2 . Suppose that P_2 requires that P_1 finishes its execution:

$P_1:$ $S_1;$ $V(S)$	$P_2:$ $P(S)$ $S_2;$
----------------------------	----------------------------

Here S is a synchronisation semaphore that's initialized to 0, when P_2 invokes $P(S)$ it will be blocked since S is 0 until P_1 invokes $V(S)$ and increments the semaphore which allows P_2 to continue executing.

Implementation:

One thing to note is that the previous definition requires busy waiting (attente active). So we need to modify $P()$ and $V()$ to use **blocking** (attente passive), to do that we need to add a waiting queue (file d'attente).

The structure of semaphore according to this definition is as follows:

Type Semaphore = Record

e : integer

$f(s)$: queue of PCB

End

We put the PCB (Process Control Block) in the queue to save the context of the blocked processes.

now we can implement P() and V():

P(s):

```
e--;  
if (e < 0) {  
    Process.etat = block;  
    Enfiler(f(s), Process.PCB);  
    Call Scheduler();  
}
```

V(s):

```
e++;  
if (e <= 0) {  
    PCB_choisi = defiler(f(s));  
    PCB_choisi.Etat = PRET;  
}
```

Problème Producteur Consommateur:

Points de synchronisation:

1) Début de dépôt (DD)

3) Début de Retrait (DR)

2) Fin de dépôt (FD)

4) Fin de Retrait (FR)

Variables:

Nplein : semaphore init 0

// nombre des cas pleins à un moment donné

Nvide : semaphore init N

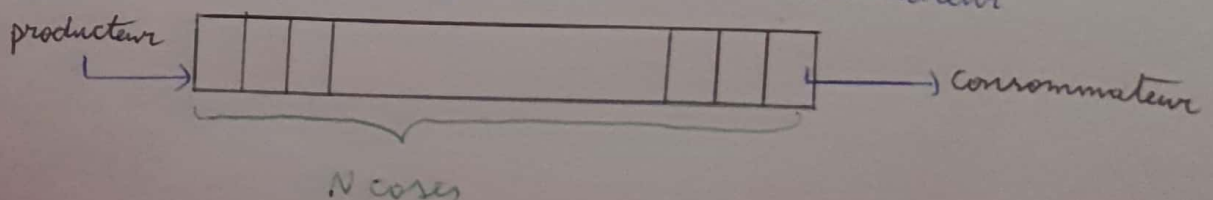
// nombre des cas vide à un moment donné

mutex : semaphore init 1

// pour assuré l'EM avant accédé au buffer

Condition de franchissement:

- 1) Le consommateur ne peut pas consommer si le buffer est vide ($N_{plein} > 0 \Leftrightarrow N_{vide} < N$)
- 2) Le producteur ne peut pas produire si le buffer est plein ($N_{plein} < N \Leftrightarrow N_{vide} > 0$)
- 3) Le consommateur et le producteur ne peut pas accéder au buffer en même temps, il y a un EM entre le producteur et consommateur



Implementation:

Producteur:

```
do {  
    P(Nvide); // attend jusqu'il  
               y a des cose vide  
    P(mutex);  
    // production (dépôt au buffer)  
    V(mutex);  
    V(Nplein); // incrimente le nbr  
               de cose plein  
} while (true)
```

Consommateur:

```
do {  
    P(NPlein); // attend jusqu'il y a  
               des cose plein  
    P(mutex);  
    // Consommation (retrait du buffer)  
    V(mutex);  
    V(Nvide); // incrimente le nbr  
               des cose vide
```

Rendez-vous de N processus avec sémaphore de synchronisation

Variables:

cpt: integer init 0; // compteur, partagé, du nbr de process qui déjà arrivés
synch: sémaphore init 0; // sémaphore de synchronisation.
mutex: sémaphore init 1;

Conditions de franchissement:

• $cpt = N$

Implementation:

$P_i, i = 1 \dots N:$

```
P(mutex);  
cpt++;  
if (cpt < N) {  
    V(mutex);  
    P(synch);  
} else {  
    while (cpt < > 1) {  
        V(synch);  
        cpt--;  
    }  
    cpt = 0; // retour à l'état initial  
    V(mutex);  
}
```

Rendez-vous de N proces en utilisant des sémaphores privés:

Variables:

cpt: integer init 0;

synch[N]: array of sémaphores init 0 // array of N private sémaphore

mutex: sémaphore init 1;

Condition de franchissement:

- $cpt = N$

Implementation:

$P_i : i = 1 \dots N :$

P(mutex);

cpt++;

if (cpt < N) {

V(mutex);

P(synch[i]);

} else {

while (cpt < N) {

V(synch[cpt-1]);

cpt--;

}

V(mutex);

}

Lecteurs / Rédacteurs: Priorité absolue aux lecteurs:

S'il y a une lecture en cours tout lecteurs peut accéder même s'il y a des rédacteurs en attente \Rightarrow possibilité de privation (starvation) des rédacteurs

Points de synchronisation:

• DL: début de lecture

• FL: Fin de lecture

• DE: début écriture

• FE: fin écriture

Variables:

nl: integer init 0 // nbr de lecteurs en cours de lecture

nlatt: integer init 0 // nbr de lecteurs en attente

nratt: integer init 0 // nbr de rédacteurs en attente

E: boolean init false // E = true : écriture en cours

mutex: semaphore init 1

Sl: semaphore init 0 // semaphore pour bloquer les lecteurs

Sr: semaphore init 0 // semaphore pour bloquer les rédacteurs

Conditions de franchissement:

- DL: (E = false)
- DE: (E = false) and (nl = 0)

Implementation:

DL:

```
P(mutex);  
if (E == true) {  
    nratt ++;  
    V(mutex);  
    P(Sl);  
} else {  
    nl ++;  
    V(mutex);  
}
```

FL:

```
P(mutex);  
nl --;  
if (nl == 0 && nratt > 0) {  
    E = true;  
    nratt --;  
    V(Sr);  
}  
V(mutex);
```

DE:

```
P(mutex);  
if (E == true || nl > 0) {  
    nratt ++;  
    V(mutex);  
    P(Sr);  
} else {  
    E = true;  
    V(mutex);  
}
```

FE:

```
P(mutex);  
E = false;  
if (nlatt > 0) {  
    while (nlatt > 0) {  
        nlatt --;  
        nl ++;  
        V(Sl);  
    }  
} else {  
    if (nratt > 0) {  
        E = true;  
        nratt --;  
        V(Sr);  
    }  
}  
V(mutex);
```

Lecteurs / Rédacteurs : pas de priorité

Si il y a une lecture en cours tout lecteur peut accéder en même temps.

Après la fin d'une écriture il n'y a pas de priorité (it is up to the CPU scheduler)

Variables:

readcount: integer init 0; // nbr de lecture en cours

wrt: semaphore init 1; // semaphore d'EM entre lecteur et rédacteur

mutex: semaphore init 1;

Condition de franchissement:

- DL: il y a aucun écriture en cours
- DE: il y a aucun lecture ou écriture

Implementation:

DL:

P(mutex);

readcount++;

if (readcount == 1) {

P(wrt);

}

V(mutex);

FL:

P(mutex);

readcount--;

if (readcount == 0) {

V(wrt);

}

V(mutex);

DE:

P(wrt);

FE:

V(wrt);

Lecteurs / Rédacteurs : Pas de priorité avec (FIFO):

Pas de priorité (FIFO), mais s'il y a une lecture en cours et la tête de file contient des lecteurs, ceux-ci doivent pouvoir accéder tout qu'il y n'y a pas de rédacteurs qui les précèdent.

Variables:

nl: integer init 0;

E: boolean init false;

fifo: semaphore init 1; // semaphore pour bloquer les process

mutex: semaphore init 1; // semaphore pour protéger les variables partagées

sync: semaphore init 0; // semaphore de synchronisation

firstblocked: liste {N, L, R} init N;

Condition de franchissement:

- DL: il n'y a pas d'écriture en cours
- DE: il n'y a pas de lecture ou écriture en cours

Implementation:

DL:

```
P(fifo)
P(mutex)
if (E) {
    firstblocked = "L";
    V(mutex);
    P(sync);
} else {
    nl++;
    V(mutex);
}
V(fifo);
```

FL:

```
P(mutex);
nl--;
if (nl >= 0) {
    if (firstblocked == "R") {
        firstblocked = "N";
        E = true;
        V(sync);
    }
}
V(mutex);
```

DE:

```
P(fifo)
P(mutex)
if (E || nl > 0) {
    firstblocked = "R";
    V(mutex);
    P(sync);
} else {
    E = true;
    V(mutex);
}
V(fifo);
```

FE:

```
P(mutex)
E = false;
if (firstblocked == "L") {
    nl++; V(sync);
} else {
    if (firstblocked == "R") {
        E = true; V(sync);
    }
}
firstblocked = "N";
V(mutex);
```

Lecteurs / Rédacteurs : priorité total au rédacteur.

Tout lecteur qui vient et trouve un rédacteur en attente doit se bloquer même s'il y a une lecture en cours.

Variables:

nl: integer init 0;
mutex: semaphore init 1;
ratt: semaphore init 1; // semaphore pour bloquer s'il y a un rédacteur en attente
resource: semaphore init 1; // pour assurer EM entre lecteurs et rédacteurs

Implementation:

DL:

```
P(ratt); // bloquer s'il y a un rédacteur en attente
P(mutex);
nl++;
if (nl == 1) {
    P(resource); // bloquer la ressource au rédacteur
}
V(mutex);
V(ratt);
```

FL:

```
P(mutex);
nl--;
if (nl <= 0) {
    V(resource);
}
V(mutex);
```

DE:

```
P(ratt); // indique qu'il y a un rédacteur en attente
P(resource); // vérifie si la ressource est libre
```

FE:

```
V(resource);
V(ratt);
```


Lecteurs / rédacteurs : Priorité partielle aux rédacteurs.

Tout lecteur qui vient et trouve qu'il y a une lecture en cours doit pouvoir accéder même s'il y a un rédacteur en attente, mais dès qu'il y a une écriture tous les rédacteurs venant après doivent être satisfait un par un.

Variables:

nl: integer init 0;

mutex: semaphore init 1;

resource: semaphore init 1;

rEnCours: semaphore init 1;

Implémentation:

DL:

P(rEnCours);

P(mutex);

nl++;

if (nl == 1) {

 P(resource);

}

V(mutex);

V(rEnCours);

FL:

P(mutex);

nl--;

if (nl <= 0) {

 V(resource);

}

V(mutex);

DE:

P(resource);

P(rEnCours);

FE:

V(rEnCours);

V(resource);