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<=0;
    s--;
}
```

```
V(s){
    s++;
}
```

An exemple of the usage of semaphores, lets consider two processes P_1 and P_2. Suppose that P_2 requires that P_1 finishes its execution:

```
P1 :
    S1;
    V(s)
```

```
P2 :
    P(s)
    S2;
```

Here S is a sychronisation semaphore that's initialized to 0, whe P_2 invokes P(s) it while be blocked since S is 0 until P_2 invokes V(s) and incriments 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 définition is as follows:

```
Type semaphore = Record
                    e : integer
                    f(s) : queue of PCB
                End
```

standard atom.

We put pointer to the PCB (Process Controle Bloc) in the queue to save the contex 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 = défiler (f(s));
      PCB_choisi.Etat = PRET;
    }
```

Probleme Producteur Consommateur: plusieurs producteurs et plusieurs consomateurs

Points de synchronisation:

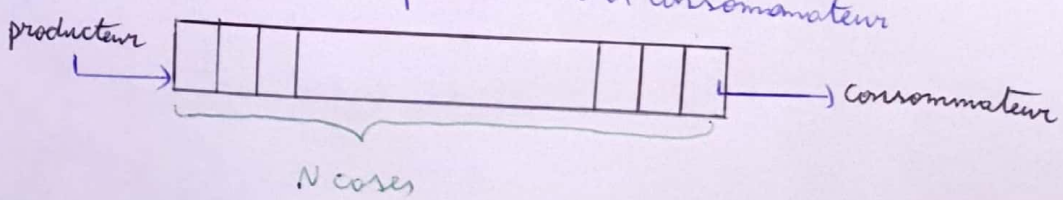
1) Début de dépôt (DD)
2) Fin de dépôt (FD)
3) Début de Retrait (DR)
4) Fin de Retrait (FR)

Variables:

```
Nplein : semaphore init 0      // nombre des cas pleines à un moment donné
Nvide : semaphore init N       // nombre des cas vide à un moment donné
mutexp : semaphore init 1      // pour assuré l'EM avant accédé au buffre pour produc
mutexC : semaphore init 1      // pour assuré l'EM avant accédé au buffer pour cons
```

Condition de franchissement:

1) Le consommateur ne peut pas consomer si le buffer est vide ($Nplein > 0 \Leftrightarrow Nvide < N$)
2) Le producteur ne peut pas produire si le buffer est plein ($Nplein < N \Leftrightarrow Nvide > 0$)
3) Le consommateur et le producteur ne peut pas accédé au buffer en même temps, il y a un EM entre le producteur et consomanateur



// semaphore is an integer ...
standard atom...

Implementation :

Producteur :

```
do {
    P(Nvide); // attend jusqu'il y a des case vide
    P(mutexp);
    // production (dépot au buffer)
    V(mutexp);
    V(Nplein); // incrimenté le nbr de case plein
} while(true)
```

Consommateur :

```
do{ P(Nplein); // attend jusqu'il y a des case plein
    P(mutexC);
    // Consomation (retrait du buffer)
    V(mutexC);
    V(Nvide); // incrementé le nbr des case vide
```

Rendez-vous de N processus avec semaphore de synchronisation

Variables :

```
cpt : integer init 0;      // compteur, portogé, du nbr de process qui déjà arrivé
synch : semaphore init 0;  // semaphore de synchronisation.
mutex : semaphore init 1;
```

Condition de franchissement :

- cpt = N

Implementation :

```
Pi , i = 1...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 process en utilisant des semaphores privés:

Variables:

cpt: integer init 0;

synch[N]: array of semaphores init 0 // array of N private semaphore

mutex: semaphore init 1;

condition de franchissement:

- cpt = N

Implementation:

```
Pi : i = 1 - N :
    P(mutex);
    cpt++;
    if (cpt < N) {
      V(mutex);
      P(synch[i]);
    } else {
      while (cpt <> 0){
        V(synch[cpt-1]);
        cpt--;
      }
    V(mutex);
  }
```

Lecteurs / Redacteur : 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 ⇒ possibilité de privation (starvation) des rédacteurs

Points de synchronisation:

- DL : debut de lecture
- FL : Fin de lecture
- DE : debut 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)

Implémentation :

DL :

```
P(mutex);
if (E == true) {
    nlatt ++;
    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 / Redacteurs : pas de priorité

S'il y a une lecture en cours tout Lecteurs peut accéder en même temps.

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

Variables :

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

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

mutex : semaphore init 1 ;

Condition de franchisement :

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

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 / Redacteurs : 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 tant 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; //semaphor pour protéger les variables partagées
sync : semaphore init 0; // semaphor de synchronisation
firstblocked: liste {N, L, R} init N;
```

Condition de franchissement :

- DL : il n'y a pas de écriture en cours
- DE : il n'y a pas de lecture ou ecriture 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 redacteur en attent
resource : semaphore init 1; // pour assuré EM entre lecteurs et rédacteurs
```

Implementation :

DL :

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

FL :

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

DE :

```
P(ratt); // indique q'il y a un redacteur en attente
P(resource); // verifi si la resource 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 rédacteur venant après doivent être satisfait un par un.

Variables :

```
nl : integer init 0;
mutex : semaphore init 1;
SL : semaphore init ;
SR : semaphore init ;
```

```
nlatt : integer init 0;
nratt : integer init 0;
E : boolean init false;
```

Implementation :

```
DL :
    P(mutex);
    if (E == true) {
        nlatt++;
        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 (nratt > 0) {
        nratt--;
        V(SR);
        E = true;
    } else {
        if (nlatt > 0) {
            while (nlatt > 0) {
                nlatt--;
                nl++; V(SL);
            }
        }
    } V(mutex);
```