Les sémaphores et leur implémentation

Rappel :Le concept de sémaphore

Un sémaphore est une variable entière partagée. Sa valeur est positive ou nulle et elle est uniquement manipulable à l'aide de deux opérations `wait(s)` et `signal(s)`, où `s` est un identificateur désignant le sémaphore.

- `wait(s)` décrémente `s` si `s > 0` ; si non, le processus exécutant l'opération `wait(s)` est mis en attente.
- `signal(s)` incrémente `s`. L'exécution de `signal(s)` peut avoir pour effet (pas nécessairement immédiat) qu'un processus en attente sur le sémaphore `s` reprend son exécution.
- L'exécution d'une opération `wait(s)` ou `signal(s)` se fait sans interaction possible (de façon atomique).

L'exclusion mutuelle à l'aide de sémaphores

Les sémaphores permettent une implémentation très simple de l'exclusion mutuelle.

```
semaphore s = 1;
```

Processus 1 :

```
while (True)
{   nc1: /* section
            non critique */ ;
         wait(s);
  crit1: /* section critique */ ;
         signal(s);
}
```

Processus 2 :

```
while (True)
{   nc2: /* section
            non critique */
         wait(s);
  crit2: /* section critique */
         signal(s)
}
```


L'implémentation des sémaphores

- Les sémaphores sont implémentés dans le noyau du système.
  - Les valeurs des sémaphores se trouvent dans une table conservée dans la mémoire du noyau. Un sémaphore est caractérisé par son numéro qui correspond à une position dans cette table.
  - Il y a des appels systèmes pour créer ou libérer un sémaphore, ainsi que pour exécuter les opérations `wait` et `signal`. Ces Opérations sont exécutées en mode superviseur et donc de façon indivisible (les interruptions sont interdites en mode superviseur).
- Dans ULg03, pour exécuter par exemple une opération `wait`, on place sur la pile les arguments de l'appel système, à savoir le numéro du sémaphore et un code `WAIT` identifiant l'opération. Supposant que le numéro du sémaphore a été placée en `r0`, cela se réalise comme suit.

```
PUSH(r0)        | 2ième argument
CMOVE(WAIT,r1)  | 1er argument
PUSH(r1)
SVC()           | appel système
```

Le handler de SVC

Avant de détailler les handler des opération sur les sémaphores, examinons le handler général nécessaire pour l'instruction SVC. On y accède par le "stub" suivant.

```
h_stub:  ST(r0, User, r31)     | sauver
         ST(r1, User+4, r31)   | les registres
              . . .
         ST(r30, User+30*4, r31)
         CMOVE(KStack, SP)     | Charger le SP du système
         BR(Svc_handler,LP)    | Appel du Handler
         LD(r31, User, r0)     | restaurer
         LD(r31, User+4, r1)
         LD(r31, User+30*4, r30)
         JMP(XP)               | retour à l'application
```

Il faut noter que l'adresse sauvée dans XP est celle de l'instruction qui suit SVC.

Le handler de SVC (suite)

- Le programme Svc_handler identifie la nature de l'appel et passe la main à la routine correspondante.
- Une difficulté est que les arguments de SVC se trouvent sur la pile du processus ayant fait l'appel, dans son *espace virtuel*.
- Il faut donc traduire ces adresses avant de les utiliser.

Le handler de SVC : les structures de données

Les structures de données utilisées par le noyau sont les suivantes.

```
struct Mstate { int R0; ..., R30;} User;
       /* l'état sauvé du processus courant */

struct PTentry {short valid, resid ; int PhysPage;};
      /* Une entrée dans la table des pages */

struct PTentry PageMap[262144];
      /* La table des pages du processus courant, 2^18 entrées */

struct PD {struct Mstate state ; int status, semwait ;
           struct PTentry PageMap[262144];} Proctbl[N];
      /* La table des processus avec table des pages, statut
         et adresse d'un semaphore bloquant éventuel */

int sem[K]; /* la table des valeurs des sémaphores */

int Cur;  /* l'index du processus courant */
```


Le handler de SVC : la traduction des adresses virtuelles

Pour avoir accès à la pile du processus ayant exécuté le SVC, le système doit traduire en adresse physique, l'adresse contenue dans r29 (SP). En effet, même si la cache de table des pages est à jour, la traduction ne se fait pas au niveau matériel puisque l'on est en mode superviseur. Cette traduction peut se faire comme suit.

```
int Vmap(int Vaddress)

 { int  VpageNo, PageNo, Offset;
   VpageNo = Vaddress>>14; Offset = Vaddress & 16383;
   if (PageMap[VpageNo].valid == 0 || PageMap[VpageNo].resid == 0)
           Pagerror(VpageNo);
   PageNo =  PageMap[VpageNo].PhysPage ;
   return((PageNo<<14) | Offset);
 }
```

Le handler de SVC : structure générale

Le handler de SVC extrait le code caractérisant l'appel et passe la main à la fonction correspondante.

```
Svc_handler()
{ int code;
  code = *Vmap(User.R29-4)
  switch (code)
       {  ....

          case WAIT   : Wait_h(*Vmap(User.R29-8)); break;
          case SIGNAL : Signal_h(*Vmap(User.R29-8)); break;
          ....
       }
}
```

Les adresses des arguments de l'appel système ne sont pas calculés exactement comme dans le cas d'un appel de procédure car, dans le cas de l'appel système rien n'est ajouté à la pile du processus appelant après l'appel.

Les handlers Wait_h et Signal_h : première version

```
Wait_h(int semno)
 { if (sem[semno] <= 0) {
      User.R30 = User.R30 - 4; /*  SVC sera réexécuté */
      Proctbl[Cur].status = WAIT; Proctbl[Cur].semwait = semno;
      scheduler();
      }  else
          sem[semno] = sem[semno] - 1;
 }
```

Lorsque le sémaphore n'est pas positif, le processus est mis en attente ; il réexécutera l'instruction SVC lorsqu'il sera réactivé.


```
Signal_h(int semno)
 { for (i = 0; i<N; i++) {
      if (Proctbl[i].status == WAIT && Proctbl[i].semwait == semno)
                 Proctbl[i].status = RUN;
   }
   sem[semno] = sem[semno] + 1;
 }
```

Les processus en attente sur le sémaphore sont réactivés et le sémaphore est incrémenté.

Tous les processus en attente réexécuteront l'appel système `wait` ; celui qui trouvera le sémaphore non nul peut être n'importe lequel.

Dans le cadre de l'exclusion mutuelle cette implémentation ne garantit pas que chaque processus aura accès à sa section critique.

Les handlers Wait_h et Signal_h : deuxième version

```
Wait_h(int semno)
 { if (sem[semno] <= 0) {
      Proctbl[Cur].status = WAIT; Proctbl[Cur].semwait = semno;
      scheduler();
      }  else
          sem[semno] = sem[semno] - 1;
 }
```

L'appel système n'est pas réexécuté lorsqu'un processus sort de son état d'attente. En effet, le handler de `signal` va se charger de réactiver un processus en attente s'il y en a un, sans passer par l'incrémentation et la décrémentation du sémaphore.

```
Signal_h(int semno)
 { int i,wokeup = 0;
   for (i = 0; i<N; i++) {
      if (Proctbl[i].status == WAIT && Proctbl[i].semwait == semno) {
          Proctbl[i].status = RUN;
          wokeup = 1; break;
      }
   }
   if (wokeup == 0) sem[semno] = sem[semno] + 1;
 }
```

Cette implémentation garantit que, si seuls deux processus utilisent le sémaphore, aucun ne restera indéfiniment en attente.

Une implémentation équitable pour plus de deux processus utilisera une file d'attente par sémaphore pour gérer les processus en attente.

L'implémentation des sémaphores sur une machine à plusieurs processeurs

- Sur une machine à plusieurs processeurs, même l'exécution en mode superviseur ne garantit pas l'exclusion mutuelle puisqu'elle peut simultanément avoir lieu sur plus d'un processeur.
- Il faut donc recourir à un autre mécanisme pour implémenter l'exclusion mutuelle.
- L'accès atomique à la mémoire en lecture ou en écriture n'est pas suffisant pour une réalisation pratique.
- On introduit donc une instruction spéciale qui permet de lire et modifier la mémoire de façon indivisible.

L'instruction "Test and Clear"

Cette instruction copie un mot mémoire dans un registre et le met à 0.

TCLR(Ra,literal,Rc) : $PC \leftarrow PC + 4$
$EA \leftarrow Reg[Ra] + SEXT(literal)$
$Reg[Rc] \leftarrow Mem[EA]$
$Mem[EA] \leftarrow 0$

On lui attribue le code opératoire `0x04`

```
.macro TCLR(Ra,Lit,Rc)          ENC_LIT(0b000100,Ra,Rc,Lit)
```


Le microcode de TCLR

`TCLR(Ra, Literal, Rc)` (mode superviseur)

Opcode = 000100 IRQ = * PC31 = 1

Phase	Fl.	$\overline{\text{Latch flags}}$	UC/D	ALU $F, \overline{C_{in}}, M$	LD SEL	DR SEL	PC+	SVR	
00000	*	1	0	000000	0011	0001	0	0	SMAR <- Ra
00001	*	1	0	000000	0001	0100	0	0	A <- SRAM
00010	*	1	0	000000	0010	0010	0	0	B <- Lit
00011	*	1	0	000000	0011	0000	0	0	SMAR <- Rc
00100	*	1	0	100110	0100	0011	0	0	DMAR <- A+B
00101	*	1	0	000000	0101	0101	0	0	SRAM <- DRAM
00110	*	1	0	001101	0110	0011	0	0	DRAM <- 0
00111	*	1	0	000000	0100	0110	1	0	DMAR <- PC; PC+
01000	*	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

L'exclusion mutuelle à l'aide de TCLR

L'exclusion mutuelle se réalise très simplement à l'aide de TCLR.

```
wait: TCLR(r31,lock,r0)
      BEQ(r0,wait)

      ... section critique ...

      CMOVE(1,r0)
      ST(r0,lock,r31)
```

Cette implémentation implique une attente active et ne garantit pas l'équité entre les processus en attente.

On ne l'utilise pas telle quelle pour gérer l'exclusion mutuelle en général, mais elle est parfaitement adéquate pour implémenter les sémaphores sur une machine à plusieurs processeurs.

Le Problème de la zone tampon (buffer) ou problème du producteur et consommateur

Un processus *Producteur* envoie un flux d'informations à un processus *Consommateur*. Une zone tampon est utilisée pour faire transiter ce flux d'informations. Elle est modélisée par un tableau en mémoire partagée utilisé à l'aide de deux fonctions (append et take).

```
/* mémoire partagée */
int in = 0, out = 0;
int buf[N];
```

```
append(int v)
 { buf[in]= v;
   in = (in+1) % N;
 }
```

```
int take()
 { int v;
   v = buf[out];
   out = (out+1) % N;
   return v;
 }
```

Telles quelles, ces opérations permettent d'écrire dans une zone tampon pleine ou de lire dans un zone vide : *il faut les synchroniser*.

Le problème de la zone tampon : Synchronisation du consommateur

Pour éviter que le consommateur ne prenne d'éléments dans une zone vide, on utilise un sémaphore qui compte le nombre d'éléments dans le tampon.

```
/* mémoire partagée */
int in, out = 0;
int buf[N];
semaphore n = 0;
```

```
append(int v)
 { buf[in]= v;
   in = (in+1) % N;
   signal(n);
 }
```

```
int take()
 { int v;
   wait(n);
   v = buf[out];
   out = (out+1) % N;
   return v;
 }
```

Il faut encore limiter le producteur lorsque le tampon est plein.

Le problème de la zone tampon : Synchronisation du consommateur et du producteur

Un deuxième sémaphore, initialisé au nombre de places vides dans le tampon permet de synchroniser le producteur.

```
/* mémoire partagée */
int in, out = 0;
int buf[N];
semaphore n = 0, e = N;
```

```
append(int v)
 { wait(e);
   buf[in]= v;
   in = (in+1) % N;
   signal(n);
 }
```

```
int take()
 { int v;
   wait(n);
   v = buf[out];
   out = (out+1) % N;
   signal(e);
   return v;
 }
```

Le problème de la zone tampon : Exclusion mutuelle des opérations

S'il y a plusieurs producteurs ou consommateurs, il faut encore assurer l'exclusion mutuelle des opérations de manipulation du tampon. Un sémaphore additionnel le permet.

```
/* mémoire partagée */
int in, out = 0; int buf[N];
semaphore n = 0, e = N, s = 1;
```

```
append(int v)
 { wait(e);
   wait(s);
   buf[in] = v;
   in = (in+1) % N;
   signal(s);
   signal(n);
 }
```

```
int take()
 { int v;
   wait(n);
   wait(s);
   v = buf[out];
   out = (out+1) % N;
   signal(s);
   signal(e);
   return v;
 }
```


Les sémaphores binaires

- Un sémaphore binaire est un sémaphore dont la valeur est limitée à 0 et 1.
- Ce type de sémaphore permet de gérer la mise en attente de processus, mais pas de compter.
- Un sémaphore général peut toutefois être simulé par un sémaphore binaire et un compteur.
- Il peut toutefois y avoir quelques difficultés liées au découplage du sémaphore et du compteur.

La zone tampon avec sémaphores binaires

```
/* mémoire partagée */
int in = 0, out = 0, count = 0;
int buf[N];
semaphore n = 0, e = 0, s = 1;
```

```
append(int v)
 { wait(s);
   if (count == N) {
      signal(s);wait(e);wait(s);}
   buf[in] = v; in = (in+1)%N;
   count = count+1;
   if (count == 1) signal(n);
   signal(s);
 }
```

```
int take()
 { int v;
   wait(s);
   if (count == 0) {
    signal(s);wait(n);wait(s);}
   v = buf[out];out = (out+1)%N;
   count = count-1;
   if (count == N-1) signal(e);
   signal(s); return v;
 }
```

Solution incorrecte car on peut exécuter `append`; `take`; `take`. Le problème est que l'on signale deux fois la présence d'un élément : en mettant `count` à 1 et par l'opération `signal(n)`.

La zone tampon avec sémaphores binaires : Solution pour un producteur et un consommateur

```
/* mémoire partagée */
int in = 0, out = 0, count = 0;
int buf[N]; int ewait = 0 ,nwait = 0;
semaphore n = 0, e = 0, s = 1;
```

```
append(int v)
 { wait(s);
   if (count == N) {
     ewait = 1; signal(s);
     wait(e);wait(s);ewait = 0;}
   buf[in] = v; in = (in+1)%N;
   count = count+1;
   if (nwait > 0) signal(n);
   signal(s);
 }
```

```
int take()
 { int v;
   wait(s);
   if (count == 0) {
     nwait = 1; signal(s);
     wait(n);wait(s);nwait = 0;}
   v = buf[out];out = (out+1)%N;
   count = count-1;
   if (ewait > 0) signal(e);
   signal(s); return v;
 }
```

- Pour éviter la double signalisation, on n'exécute `signal(n)` ou `signal(e)` que s'il y a un processus en attente.

- On peut imaginer d'étendre cette solution à plusieurs producteurs et consommateurs en comptant le nombre de processus en attente.

- Toutefois, il faut éviter que, par exemple, un deuxième consommateur ne dépasse un consommateur qui vient d'être libéré et ne consomme l'élément destiné à ce dernier.

- Cela peut se faire en transférant l'exclusion mutuelle au processus libéré.

La zone tampon avec sémaphores binaires : Solution pour plusieurs producteurs et consommateurs

```
/* mémoire partagée */
int in = 0, out = 0, count = 0;
int buf[N]; int ewait = 0 ,nwait = 0;
semaphore n = 0, e = 0, s = 1;
```

```
append(int v)
 { wait(s);
   if (count == N) {
     ewait++;signal(s);wait(e);
     ewait--;}
   buf[in] = v;in = (in+1)%N;
   count = count+1;
   if (nwait > 0) signal(n);
            else  signal(s);
 }
```

```
int take()
 { int v;
   wait(s);
   if (count == 0) {
     nwait++; signal(s);wait(n);
     nwait--;}
   v = buf[out];out = (out+1)%N;
   count = count-1;
   if (ewait > 0) signal(e);
             else signal(s);
   return v;
 }
```


Conclusions sur les sémaphores

- Les sémaphores sont un excellent mécanisme pour gérer la mise en attente de processus.

- Lorsqu'il faut gérer la mise en attente en combinaison avec la manipulation d'un compteur, les sémaphores sont très bien adaptés.

- L'utilisation des sémaphores binaires indique quelles difficultés apparaissent lorsque l'on combine l'attente implémentée par un sémaphore avec la manipulation d'une autre structure de donnée.

- Il serait intéressant d'avoir une solution systématique pour gérer ce type de problème.