Contrôle Module Noyau
M1 – Master SAR
Novembre 2006

2 heures – Tout document papier autorisé
Barème donné à titre indicatif

L. Arantes, P. Sens, Gaël Thomas

1. SYNCHRONISATION (55 MIN. 10 POINTS)

Considérez le code du Noyau **non préemptif** vu en TD. Nous voulons y ajouter un mécanisme de sémaphore.

Un sémaphore permet de contrôler l'accès à une ressource ayant *N* exemplaires. Cela dit, au maximum *N* processus peuvent accéder à cette ressource concurremment.

Les appels systèmes à offrir aux utilisateurs sont : *int Init_sem (int sem, int N), int Lib_sem(sem*), int *P(int sem)* et *int V(int sem)*. Ils permettent d'accéder à un objet sémaphore identifié par *sem* qui contrôle les exemplaires d'une ressource quelconque. Ces appels systèmes appellent respectivement les fonctions du noyau *int sys_Init_sem(int sem, int N), int sys_Lib_sem(sem*), int sys_*P(int sem)* et *int sys_V(int sem)* que nous voulons programmer.

Un sémaphore est un objet composé par :

(1) Un compteur *compt* qui contrôle le nombre d'exemplaires disponibles ou le nombre de processus en attente pour la ressource : si *compt >=0*, *compt* indique le nombre d'exemplaires courants disponibles ; si *compt <0*, *-compt* indique le nombre de processus en attente pour la ressource ;

(2) Une file *File* où sont mis les processus en attente pour un exemplaire de la ressource. La file *File* possède la taille **MAX_FILE** et est gérée de façon **circulaire** (**FIFO)**. La file est nécessaire parce que les **demandes pendantes doivent être satisfaites dans l'ordre de demande**.

Les fonctions pour manipuler un sémaphore identifié par un entier *sem* sont :

- ***int Init_sem (int sem,int N)*** : alloue le sémaphore identifié par *sem* en l'initialisant son compteur à N (nombre d'exemplaires de la ressource qui se trouvent disponibles). Une fois qu'un sémaphore a été initialisé, les autres processus qui veulent l'utiliser n'ont pas besoin (et ne doivent pas) l'initialiser (N >=0).

- **int *Lib_sem(int sem)* :** libère le sémaphore *sem* alloué précédemment. Pour simplifier nous considérons initialement que lorsque cette fonction est appelée, aucun exemplaire de la ressource n'est utilisé par un processus.

- ***int P(int sem)*** : permet à un processus de demander un exemplaire de la ressource gérée par *sem.* S'il y au moins un exemplaire de la ressource disponible (*compt >0*), un exemplaire est réservé au processus et la fonction se termine. Sinon, s'il y a de la place dans *File*, le processus est mis **à la fin** de cette file et il s'endort en attente d'un exemplaire disponible. La priorité de réveil est **PRISEM** dans ce cas. En se réveillant, le processus a le droit à un exemplaire de la ressource et la fonction alors se termine.

Code de renvoi : -1 : si le sémaphore n'avait pas été initialisé ou s'il n'y a pas de place dans *File* ;

0 : succès.

- ***int V (int sem)*** **:** libère **un** exemplaire de la ressource gérée par *sem*. De plus, s'il y a des processus dans *File* en attente pour un exemplaire, le **premier** processus de *File* est réveillé.

Code de renvoi : -1 : le sémaphore n'avait pas été initialisé ;

0 : succès.

Observation : les possibles codes de renvoi d'une fonction du noyau sont les mêmes qui ceux de son appel système correspondant.

Exemple : Trois processus P_1, P_2 et P_3 utilisent le sémaphore 0 afin de contrôler l'accès à une ressource ayant 2 exemplaires. Le processus ***Init,*** qui initialise le sémaphore, est exécuté avant les 3 autres.

```
Init :
 main  ( ) {
    init_sem (0,2) ;
 }
```

```
  P1 :                          P2 :                          P3 :
main ( ) {                    main () {                     main () {
   …                             ….                            ….
  P(0);                         P(0) ;                        P(0);
   Utiliser la ressource         Utiliser la ressource         Utiliser la ressource
  V(0);                         V(0);                         V(0);
   …..                          }                             }
 Lib_sem (0) ;
}
```

Observation : Si les deux exemplaires de la ressource sont réservés, le troisième processus sera bloqué tant qu'un exemplaire n'est pas libéré.

La valeur de ***sem*** est un entier entre ***0*** est ***NB_SEM -1.***

Pour implanter les sémaphores, nous avons créé la structure de données *sem* :

```
struct sem {
        int compt ; /* compteur du sémaphore */
        int File [MAX_FILE]; /* File d'attente gérée de façon circulaire */
        int first ; /* première case occupée dans File */
 }
```

Nous avons aussi ajouté à la variable globale ***v*** du noyau le vecteur ***vect_sem*** de taille ***NB_SEM :***

```
struct var {
….
struct sem vect_sem [NB_SEM];
} v ;
```

A l'initialisation du Noyau, le champ *first* de toutes les entrées de *v.vect_sem* est initialisé à -1 pour indiquer qu'aucun sémaphore est réservé.

Le code des fonctions *sys_Init_sem (int sem, int N), sys_Lib_sem* (*int sem*), *sys_P(int sem)* et *sys_V(int sem)* est respectivement :

```
    int sys_Init_sem (int sem, int N) {
      if  ( (sem > NB_SEM-1) || (sem <0) )
            return -1;
      if  (v.vect_sem[sem].first != -1)
            /*semaphore déjà en utilisation */
            return -1 ;
       v.vect_sem[sem].first=0;
       v.vect_sem[sem].compt=N;
       return 0;
     }

int sys_Lib_sem (in sem) {
     v.vect_sem[sem].first=-1;
     return 0;
}

int sys_P (int sem) {
    if ( (v.vect_sem[sem].first == -1) || (v.vect_sem[sem].compt == - MAX_FILE) )
      /* sémaphore n'est pas initialisé ou file pleine*/
       return -1;
   /* code à compléter - question 1.2 */
   ……
   return 0 ;
}

int sys_V (int sem) {
    if (v.vect_sem[sem].first == -1)
         /* sémaphore n'est pas initialisé */
        return -1;
  /* code à compléter - question 1.2*/
   ……
   return 0 ;
 }
```

1.1

En considérant que les opérations sur un sémaphore doivent être **atomiques**, pourquoi n'est-t-il pas nécessaire de masquer les interruptions dans le corps des fonctions décrites ci-dessus?

Ce n'est pas nécessaire parce que le noyau est non préemptif et aucune interruption manipule de variables globales du noyau concernant les sémaphores.

1.2

Complétez les fonctions *int sys_P(int sem)* et int sys_V(sem).

Observation : Si vous le jugez nécessaire, vous pouvez ajouter d'autres champs à *struct sem*.

```
int sys_P (int sem) {
     if (v.vect_sem[sem].first == -1)
        /* semaphore pas initialise */
     return -1;

     if (v.vect_sem[sem].compt == - MAX_FILE)
       /* file pleine*/
          return -1;

      v.vect_sem[sem].compt--;
     if (v.vect_sem[sem].compt <0)
      sleep( &(v.vect_sem[sem].File[(first–compt-1) %MAX_FILE]),PRISEM);
     return 0;
}


int sys_V (int sem) {
     if (v.vect_sem[sem].first == -1)
        /* semaphore pas initialise */
     return -1;


      v.vect_sem[sem].compt++;
     if (v.vect_sem[sem].compt >=0) {
        wakeup(&(ve.vect_sem[sem].File[first]));
         first = (first+1)%MAX_FILE;
      }
     return 0;
}
```

Supposons maintenant que nous ajoutons le champ *int compt_init* à *struct sem* afin de sauvegarder la valeur d'initialisation du sémaphore. Cette sauvegarde a été ajoutée au code de la fonction sys_*Init_sem* :

```
int sys_Init_sem (int sem, int N) {
…. /* même code que la version précedente */
   v.vect_sem[sem].compt_init=N;
   return 0;
}
```

1.3

Modifiez la fonction *int sys_Lib_sem (int sem)* pour qu'elle soit bloquante. Tant qu'il y a des processus utilisant un exemplaire de la ressource ou en attente pour un exemplaire, le processus qui a appelé *Lib_sem* restera bloqué et le sémaphore ne sera pas libéré.

Quelle fonction va réveiller ce processus ? Donnez le nouveau code de cette fonction (seulement les lignes ajoutée).

Observation : vous pouvez ajouter des champs à *struct sem*, si nécessaire. La priorité de réveil est PRISEM.

On ajoute le champs ***int flag*** dans *struct sem*;

```
int sys_Lib_sem (int sem) {
     while (v.vect_sem[sem].compt != v.vect_sem[sem].compt_init) {
        v.vect_sem[sem].flag |= I_WANT;
        sleep(&v.vect_sem[sem].flag,PRISEM);
      }
    ve.vect_sem[sem].first =-1;
}

int sys_V (int sem) {
     if (v.vect_sem[sem].first == -1)
        /* semaphore pas initialise */
     return -1;


      ve.vect_sem[sem].compt++;
     if (v.vect_sem[sem].compt >=0) {
        wakeup(&(v.vect_sem[sem].File[first]));
        first = (first+1)%MAX_FILE;
      }
    if ((v.vect_sem[sem].flag | I_WANT) &&
     (v.vect_sem[sem].compt == v.vect_sem[sem].compt_init)) {
       v.vect_sem[sem].flag |= ~I_WANT;
       wakeup(&(v.vect_sem[sem].flag));
    }
      return 0;
}
```

2. SIGNAUX (25 MIN – 6 POINTS)

Considérez le code du Noyau **non préemptif** vu en TD.

Nous voulons offrir aux utilisateurs l'appel système *int tsigwait (long ens)* qui renvoie le plus petit numéro du signal compris dans la liste de signaux pendants du processus appelant, appartenant aussi à l'ensemble *ens*. Le signal en question est retiré de la liste de signaux pendants. Cependant, s'il n'existe aucun signal pendant qui est compris dans *ens*, le processus appelant est bloqué jusqu'à réception d'un signal.

2.1

Programmez la fonction du noyau *int sys_tsigwait (long ens)* appelée par l'appel système *tsigwait* décrit ci-dessus. Si nécessaire, le processus doit s'endormir avec la priorité ***PRISIG*** (interruptible par des signaux) sur l'adresse de la variable du noyau qui sauvegarde les signaux pendants du processus appelant.

```
/* ma solution ressemble beaucoup à fsig.

int sys_tsigwait(long ens) {
struct proc *p= u.u_procp;
long n;

   n= (p->p_sig & ens);
   /* n= signaux qui se trouvent dans les deux ensembles */
```

```
   for (i=1; i<NSIG ; i++){
     if (n& 1L)
       {  /* mettre à zero le bit correspondant au signal dans p_sig*/ {
         p->p_sig &= ~(1L<<(i-1));
         return (i);
       }
      n>>1;
     }

   /* aucun signal trouvé */
   sleep (&(p->p_sig),PRISIG); /* la function sortira avec -1*/
   return 0;
}
```

2.2

Supposons que le processus en appelant *tsigwait* s'endorme. Quand est-ce qu'il sera réveillé ? Quelle fonction va le réveiller ? Quel sera le code de renvoie de la fonction *tsigwait*?

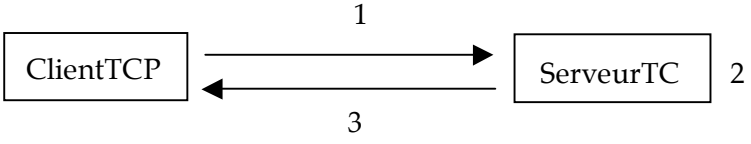
Le processus sera réveillé lorsqu'un signal lui est envoyé. La fonction *psignal* (appelée par kill) vérifiera que le processus se trouve dans l'état SLEEP et appellera alors *setrun* qui mettra le processus dans l'état prêt. Lorsque le processus aura la CPU, la fonction *sleep*, appellera *issig ( )*. Comme il y aura un signal pendant dans ce cas, le *sleep* fera un *longjmp* vers le *syscall*. Le code de renvoi sera -1 *(errno =EINTR)*.

3. UTILISATION D'UN SERVEUR TCP AVEC UDP (25 MIN. 4 POINTS)

Le but de cet exercice est de réutiliser le serveur de variable TCP écrit en TME, pour gérer des clients qui communiquent en UDP. Contrairement à la solution proposée dans le TME, on ne souhaite pas modifier le serveur écrit en TCP. On définit la structure des messages de la façon suivante :

```
#define MSGSET 0
#define MSGGET 1
strcut msg {
  int type ; /* MSGSET ou MSGGET
 char var[256] ;
 char val[256] ;
}
```

Le protocole original peut être symbolisé de la façon suivante.



Avec 1, 2 et 3 symbolisant les actions suivantes :

- 1 : msg avec msg.type = MSG**GET**, 2 : consultation de la variable, 3 : msg avec val la valeur de la variable.
- 1 : msg avec msg.type = MSG**SET**, 2 : modification de la variable, 3 : msg d'origine.

De manière à utiliser le serveur écrit en TCP avec des clients écrits en UDP, on interpose un nouveau serveur, appelé mandataire, entre les deux. Le mandataire s'occupe de traduire les requêtes UDP en TCP. Le mandataire permet donc de réutiliser le serveur TCP sans modification. La figure suivante symbolise le fonctionnement.

3.1

Brièvement, que se passe-t-il si le mandataire tombe en panne ? Est-ce que le service peut toujours être rendu ? Et si le serveur TCP tombe en panne ?

Si le mandataire tombe en panne, marche toujours en TCP, si le serveur tombe en panne, ne marche plus du tout.

3.2

Combien de sockets doivent être créées sur le mandataire ? Peut on utiliser le même numéro de port côté client et côté serveur ?

2 sockets (une en TCP, une en UDP). Oui car port TCP et port UDP n'ont rien à voir.

3.3

Programmez le mandataire sachant que le serveur ne ferme pas sa socket de communication TCP après le traitement d'une requête.

```
void main(int argc, char **argv) {
 int sclient = socket(AF_INET, SOCK_DGRAM, 0) ;
 int sserver = socket(AF_INET, SOCK_STREAM, 0) ;
 struct sockaddr_in saddr,exp ;
 struct hosten *h;
 int len = sizeof(exp);
 h = gethostbyname(argv[1]);
 memcpy(saddr.sin_addr, h->h_addr, h->h_lenght) ;
 saddr.sin_port = htons(argv[2]) ;
 saddr.sin_family = AF_INET ;
 int scom = connect(sserver, saddr) ;
 saddr.sin_addr = IN_ADDR_ANY ;
 bind(sclient, saddr) ;

 while(1) {
   stuct msg m;
   recvfrom(sclient, &m, sizeof(m), exp, &len) ;
   write(scom, &m, sizeof(m)) ;
   read(scom, &m, sizeof(m)) ;
```

```
    sendto(sclient, &m, sizeof(m), ex, len) ;
  }
}
```