Maîtrise Noyau **Contrôle Continu** **25/11/2003**	**- Durée : 2 heures** **- Toute documentation autorisée** **- Barème indicatif** ***Luciana Arantes***

Pour toutes les questions, nous considérons un **noyau unix non-préemptif**.

EXERCICE 1 – SOCKET UDP, INTERRUPTIONS et SIGNAUX (10 points)

Nous considérons le squelette d'un client UDP, comme nous avons vu en TME.

Le processus client reçoit une requête de l'usager, l'envoie au serveur UDP, attend la réponse du serveur à sa requête et la traite après l'avoir reçue :

```
int main(int argc, char *argv[]){

  struct sockaddr_in dest,from;
  int sock;
  int fromlen = sizeof(dest);
  char  message[TAILMSG];

  /* ouvrir socket: sock = socket(AF_INET,SOCK_DGRAM,0) */
   sock=ouvrir_socket ();

  /* Remplir la structure dest avec adresse et port du serveur UDP*/
  prepare_adresse_serveur (&dest);

  while (1) {
    obtenir_requete (message);

     /* Envoyer la requête */
     if (sendto(sock, message, strlen(message),0,(struct sockaddr*)&dest,
            sizeof(dest)) == -1)
        perror ("sendto"); exit (1); }

     /* attendre la reponse */
      if (recvfrom(sock,message,strlen(message),0,&from,&fromlen) == -1){
           perror ("recfrom"); exit (1); }

     traiter_reponse (message);
  }

  /* fermer socket : close (sock) */
  fermer_socket(sock);

  return(0);
}
```

Q1.1 (1,5 points) - En recevant la réponse, le processus client ne peut pas garantir si elle a été vraiment envoyée par le bon serveur. Pourquoi? Modifiez le code en conséquence.

Réponse :

(0,5 points):

Il s'agit d'un client UDP (connexion avec le serveur n'est pas établie). En principe, n'importe quel processus qui connaît l'adresse IP du client et le numéro du port (temporaire) qui lui a été affecté lors du *sendto* peut lui envoyer un message.

(1 point) code modifié:

```
if ((fromlen != sizeof (dest)) || memcmp (&dest,,&from, fromlen))
     {printf ("message n'a pas été envoyé par le bon serveur \n");
else {
     if (recvfrom(sock,message,strlen(message),0,&from,&fromlen) ==
     -1){
          perror ("recfrom"); exit (1);
     }
    traiter_reponse (message);
}
```

****

Tant que le client ne reçoit pas un message, la fonction *recvfrom* endort le processus client (*sleep*) sur l'adresse du descripteur de la socket (*sock*) avec la priorité PSOCK (voir table des priorités d'Unix 4.4 BSD en annexe). Dans ce cas, c'est la routine de traitement d'interruption de la carte réseau qui aura la responsabilité de réveiller le processus client.

Q1.2 (2,5 points)

- Est-ce que l'interruption réseau est forcement traitée dans le contexte du processus client (**0,5 points)**?

Réponse
Non

- Quand est-elle traitée ? (**1 point)**

Réponse

Dès que possible, c'est à dire, dès que le noyau ne se trouve pas à exécuter des codes non interruptible par des interruptions avec un niveau égal au supérieur à l'interruption réseau (ex.: section critique, routine d'interruption, etc.)

- Lorsqu'un message arrive du serveur et que le processus client est réveillé, sera-t-il plus prioritaire à s'exécuter que d'autres processus usagers? Justifiez votre réponse. (**1 point)**

Réponse

Le processus s'est endormi avec une priorité PSOCK. Par conséquent, il sera plus priorité à s'exécuter que tous le processus dont la priorité est inférieur à PSOCK (par exemple, ce qui sont dans "l'état prêt" avec une priorité PUSER).

****

Q1.3 (3 points) – Si un signal <crtl-C> (SIGINT) est envoyé au processus client, il se terminera (traitement par défaut). Modifiez le code du programme pour qu'il ne se termine qu'en recevant dix <crtl-C>. Observation: Il n'est pas nécessaire de traiter les autres signaux, c'est à dire, vous pouvez garder leur traitement par défaut.

```
int main(int argc, char *argv[]){

  struct sockaddr_in dest,from;
  int sock;
  int fromlen = sizeof(dest);
  char  message[TAILMSG];

int sig_cont;

void sig_INT (int sig) {
  sig_cont++;
}

action.sa_flags=0;
sigemptyset (&action.sa_mask);
action.sa_handler=sig_INT;

sigaction(SIGINT,&action, NULL);


  /* ouvrir socket: sock = socket(AF_INET,SOCK_DGRAM,0) */
   sock=ouvrir_socket ();
  /* Remplir la structure dest avec adresse et port du serveur UDP*/
  prepare_adresse_serveur (&dest);

  while (1) {
    obtenir_requete (message);

     /* Envoyer la requête */
     if (sendto(sock, message, strlen(message),0,(struct sockaddr*)&dest,
            sizeof(dest)) == -1)
        perror ("sendto"); exit (1); }
     /* attendre la reponse */
      if (recvfrom(sock,message,strlen(message),0,&from,&fromlen) == -1){
       /* <ctrl –C> ou erruer */
       if (errno == EINTR) {
         if (sig_cont == 10)
             exit (1);
       }
       else {
            perror ("recvfrom"); exit (1); }
     }
       else /* message reçu */
          traiter_reponse (message);
  }


  /* fermer socket : close (sock) */
  fermer_socket(sock);

  return(0);
}
```

Q1.4 (2 points) – Considérez que, pour une raison quelconque, le serveur ne répond pas. Le client restera alors bloqué dans la fonction *recvfrom* en attente d'un message. Que se passe-t-il au niveau de l'exécution du programme client et du noyau si le premier SIGINT est envoyé à ce moment là ? Quand le SIGINT sera-t-il détecté ? Quand sera-t-il traité ?

Réponse :

(1,5 points):

Niveau de l'exécution du programme client: le programme va recevoir –1 comme code de renvoie de *recvfrom* et *errno* sera égal à EINTR.

Niveau noyau: le processus sera réveillé est mis dans l'état "prêt à exécuter". Lorsqu'il s'exécute (code fonction sleep) *issig* va être appelée (issig () après le switch ()). Comme un signal a été envoyé au processus, un "longjump" aura lieu et la fonction recvfrom terminera avec –1 et errno = EINTR.

Signal détecté dans sleep (voir paragraphe précédant).

(0,5 points):

Le signal sera traité (psig) lorsque le processus passe du mode système vers le mode usager (trap à la fin de recvfrom)

****

Q1.5 (1 point) – L'usager a remarqué que ce le processus client ne se termine pas toujours lorsqu'il envoie les dix <crtl-C>. Dans quel cas cela peut arriver ?

Réponse :

Le champ *p_sig* de la structure *proc* sauvegarde les informations des signaux envoyés en réservant un bit par signal. Cela entraîne une impossibilité de sauvegarder plusieurs occurrences d'un même signal non traité. Par conséquent, si deux au plus <crtl-C> ont été envoyés au processus avant qu'il ait pu s'exécuter, quelques occurrences de ce signal ne vont pas être traitées.

EXERCICE 2 – SYNCHRONISATION (10 points)

Une **barrière** est un mécanisme de synchronisation. Elle permet à N processus de prendre rendez-vous en un point donné de leur exécution. Quand un des processus atteint la barrière, il reste bloqué jusqu'à ce que tous les autres arrivent à la barrière. Lorsque les N processus sont arrivés à la barrière, chacun des processus peut alors reprendre son exécution.

Nous voulons offrir 3 fonctions système (*open_barrier, sync_barrier* et *close_barrier*) qui permettent aux programmeurs d'utiliser le mécanisme de barrière afin de synchroniser des processus concurrents en certains points du programme.

Un processus ne peut utiliser qu'une barrière à la fois. Avant de l'utiliser il faut qu'elle soit ouverte en indiquant le nombre de processus *n* qui iront se synchroniser (fonctions *open_barrier* décrite ci-dessous). Les fils héritent de la barrière ouverte.

Nous avons introduit au **niveau de noyau** :

- une structure du type barrière :

```
struct barrier {
        /* a compléter */
  };
```

- un vecteur global *barrier_vect* de type *struct barrier* dont la taille est MAX_BARRIER:

```
struct  barrier barrier_vect [MAX_BARRIER];
```

- un champ *f_barrier* du type *struct barrier** à la *zone* u :

```
struct user {
       ....
    struct barrier* f_barrier;
}u;
```

- Une priorité PBAR, définie comme la priorité de réveil pour un processus qui attend sur une barrière.

Les fonctions sont les suivantes :

- *int open_barrier (int n)* : ouvre une barrière. Le paramètre *n* indique le nombre de processus qui iront se synchroniser sur la barrière. Si le processus ne possède pas déjà une barrière ouverte, la fonction cherche une entrée libre dans le vecteur *barrier_vect* et l'initialise avec le nombre total de processus *n*. Le champ *f_barrier* de la *zone u* du processus pointera alors vers cette entrée. En cas de succès, la fonction retourne 0, sinon -1. (Obs.: un processus ne peut pas ouvrir une barrière si elle se trouve déjà ouverte).

- *int close_barrier ( )* : ferme une barrière. En cas de succès, la fonction retourne 0, sinon –1. (Obs.: un processus ne peut pas fermer une barrière s'il y a des processus qui attendent sur cette barrière) .

- *sync_barrier* (*)*: permet aux processus de se synchroniser sur la barrière, c'est à dire, d'attendre que tous les processus arrivent à la barrière. Elle retourne –1 si la barrière n'avait pas été initialisée ou 0 en cas de succès.

Le programme ci-dessous montre un exemple d'utilisation du mécanisme de barrière afin de synchroniser 3 processus fils. La barrière est ouverte par le processus *main* qui ne

participe pas à la synchronisation. Les fils héritent de la barrière ouverte. Le résultat affiché est aussi montré.

```
# define PROC_FILS 3

int main (int argc, char** argv)
{ int i;

  printf ("ouvrir barriere \n");
  if (! open_barrier (PROC_FILS)) {
       printf ("erreur \n"); exit (1);}

  for (i=0; i < PROC_FILS; i++)

    if (fork () == 0) {

      /* processus fils - hérite de la barrière*/

      printf("avant barriere \n");
      sync_barrier ();
      printf("apres barriere \n");

      exit (0);
    }

      /* pere */
      /* attendre la fin de ses fils */
      for (i= 0; i <  PROC_FILS; i++)
         wait (NULL);

    printf ("fermer barriere \n");
    if (! close_barrier ()) {
        printf ("erreur \n"); exit (1);}

    exit (0);
}
```

```
ouvrir barriere
avant barriere
avant barriere
avant barriere
apres barriere
apres barriere
apres barriere
fermer barriere
```

Q 2.1 (1 point) – A votre avis, la priorité PBAR doit-elle être interruptible aux signaux ? Justifiez votre réponse.

Réponse

Oui, elle doit être interruptible, vu que *sync_barrier* est une fonction offerte au programmeur.

Si le programme de l'usager n'utilise pas la fonction correctement (par exemple, le nombre de processus qui se synchronisent en appelant *sync_barrier* est inférieur au numéro passé comme paramètre dans la fonction *open_barrier*), les processus qui ont appelé *sync_barrier* resteraient bloqués pour toujours sans que l'utilisateur ait la possibilité de les terminer par un envoie d'un signal (ex. <crtl-C>).

****

Q.2.2 (8 points) – Programmez les fonctions *open_barrier, close_barrier* et *sync_barrier*. Vous pouvez ajouter des champs à la structure *struct barrier.* Pour la fonction *sync_barrier*, vous devez utiliser les primitives *sleep*/wakeup .

Réponse

Dans *struct barrier* ajouter les champs: **(0,5 points)**

```
struct barrier {
     int nb_proc; /* nombre total de processus qui doivent se
                  synchroniser à la barrier */
     int nb_cont;  /* compteur des processus qui se trouve déjà à la
                   barrière */
  };
```

```
int open_barrier (int n) { (2,5 points)
  int i=0 ;
      if (u.f_barrier != NULL)
      /* barrière déjà ouverte */
       return (-1) ;

      /* chercher une entrée libre dans barrier_vect */
      while ((i < MAX_BARRIER) && (barrier_vect [i].nb_proc !=0) ) i++ ;

      if (i == MAX_BARRIER)
       /* il n'y a plus d'entrée libre dans barrier_vect */
        return (-1);

      /* initialiser l'entrée du vecteur affectée au processus */
      barrier_vect[i].nb_proc = n;
      barrier_vect[i].nb_cont = 0;

      /* initialiser le pointeur de la zone u vers sa variable barrière */
      u.f_barrier = &(barrier_vect[i]);

     return (0);
}
```

```
int sync_barrier () { (3 points)
     if ((u.f_barrier == NULL) || (u.f_barrier ->nb_proc == 0))
       /* barrier n'a pas été ouverte ou a été fermé par un autre
        processus*/
       return (-1);
     u.f_barrier -> nb_cont++;
     if (u.f_barrier ->nb_cont == u.f_barrier ->nb_proc) {
        /* tous les processus sont arrivés à la barrière */
        wakeup (u.f_barrier);
        u.f_barrier ->nb_count =0;
     }
      else
        sleep (u.f_barrier,PBAR);

   return (0);
}
```

```
int close_barrier () {
```
(2 points)

```
      if ((u.f_barrier == NULL) ||(u.f_barrier->nb_cont != 0))
          /* barrière n'est pas ouverte ou elle est en train d'être
             utilisée par un processus*/
       return (-1) ;

      u.f_barrier -> nb_proc = 0;
      u.f_barrier = NULL;

      return (0);
}
```

****

Q 2.3 (1 point) - Serait-il possible d'utiliser ces fonctions système pour synchroniser des threads? Justifiez votre réponse.

Réponse

Oui, s'il s'agit des **threads noyau**. Sinon, lorsqu'un thread usager appèle la fonction *sys_barrier*, le processus sera bloqué et les autres threads qui pourraient, en arrivant à la barrière, libérer le premier thread (et le processus en question) ne seront jamais exécutées.

Cependant, pour que les fonctions puissent être utilisées, elles ne peuvent pas utiliser u.f_barrier pour sauvegarder la variable barrière vu que la zone u du processus n'est pas partagée par les threads. On pourrait, par exemple, utiliser une variable du processus (tous le threads pourront alors y accéder).

Annexe

Priorité 4.4 BSD