Maîtrise Noyau **Contrôle Continu** **08/11/2004**	**- Durée : 2 heures** **- Toute documentation autorisée** **- Barème indicatif** ***Luciana Arantes, Pierre Sens et Gael Thomas***

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

EXERCICE 1 – SOCKET (6 points)

On considère un simple programme client / serveur fonctionnant en UDP. Le code de ce programme est donné en annexe (clientUDP.c et serveurUDP.c). On souhaite modifier le code du serveur pour que celui-ci ne traite pas directement les requêtes du client mais les redirige vers un autre serveur situé sur la machine « 192.168.0.10 ». C'est le second serveur qui répondra directement au client.

On souhaite que ce mécanisme soit totalement transparent au client (le code de clientUDP.c ne doit pas être modifié).

Q1.1 – Modifier le code du serveur initial pour rediriger le message du client vers le second serveur **(3 points).**

```
#define PORTSERV 4567
#define IP_2ND_SERVEUR "192.168.0.10"

int main(int argc, char *argv[])
{
 struct sockaddr_in sin;  /* Nom de la socket du serveur */
 struct sockaddr_in exp;       /* Nom de l'expediteur */
 int sc;
 int fromlen = sizeof(exp);

 /* creation de la socket */
 if ((sc = socket(AF_INET,SOCK_DGRAM,0)) < 0) {
  perror("creation socket");
  exit(1);
 }

 /* remplir le «nom» */
 bzero((char *)&sin,sizeof(sin));
 sin.sin_addr.s_addr = htonl(INADDR_ANY);
 sin.sin_port = htons(PORTSERV);
 sin.sin_family = AF_INET;

 /* nommage */
 if (bind(sc,(struct sockaddr *)&sin,sizeof(sin)) < 0) {
  perror("bind");
  exit(2);
 }
```

```
/* Remplir la structure identifiant le 2nd serveur */
sin.sin_addr.s_addr = inet_addr(IP_2ND_SERVEUR);

for (;;) {

  /*** Reception du message ***/
  if ( recvfrom(sc,&cpt,sizeof(cpt),0,
                        (struct sockaddr *)&exp,&fromlen) == -1) {
    perror("recvfrom");
    exit(2);
  }

  /*** Envoyer l'identifiant au second serveur la reponse ***/
  if (sendto(sc,&exp,sizeof(exp),0,(struct sockaddr *)&sin,fromlen) == -1) {
    perror("sendto");
    exit(2);
  }

  /*** "Forward" de la requete du client  ***/
  if (sendto(sc,&cpt,sizeof(cpt),0,(struct sockaddr *)&sin,fromlen) == -1) {
    perror("sendto");
    exit(2);
  }
}
close(sc);
return (0);
```

Q1.2 – Ecrivez le code du second serveur qui reçoit le message diriger, fait le traitement et renvoie directement sa réponse au client **(3 points).**

```
for (;;) {

  /*** Reception de l'identifiant du client ***/
  if ( recvfrom(sc,&client,sizeof(client),0,
                    (struct sockaddr *)&exp,&fromlen) == -1) {
    perror("recvfrom");
    exit(2);
  }

  /*** Reception du message du client***/
  if ( recvfrom(sc,&cpt,sizeof(cpt),0,
                    (struct sockaddr *)&exp,&fromlen) == -1) {
    perror("recvfrom");
    exit(2);
  }

  /*** Traitement ***/
  cpt++;
```

```
   /*** Envoyer la reponse au client initial ***/
   if (sendto(sc,&cpt,sizeof(cpt),0,(struct sockaddr *)&client,fromlen) == -1) {
     perror("sendto");
     exit(2);
   }
  }

  close(sc);

  return (0);
}
```

EXERCICE 2 – SYNCHRONISATION (8 points)

Nous supposons que le NOYAU possède une variable globale appelée *pile_partage*. C'est une variable de taille MAX_PILE partagée par tous les processus. Cependant, un processus ne peut y accéder qu'en utilisant les fonctions du système *void fs_push (int val)* et *int fs_pop ()* qui permettent à un processus respectivement d'empiler et de dépiler une valeur entière. La pile est de taille MAX_PILE mais la position 0 est utilisée pour sauvegarder le sommet de la pile. Plusieurs processus peuvent partager la pile.

Nous introduisons au **niveau du noyau** :

- la variable *pile* globale :

int pile_partage [MAX_PILE] ;

int *flag_pile* est une variable globale qui sauvegarde l'état de la pile. Il peut être affecté avec les valeurs P_PLEINE et P_VIDE.

- à la zone u :
  - le champ *p_pile* qui pointe vers la variable *pile_partage* ;

```
struct user {
        ....
        int* p_pile;
 }u;


 int flag_pile;
```

- Une priorité *PPILE*, définie comme la priorité de réveil pour un processus qui attend sur la pile.
- Les constantes:
  - *P_PLEINE* : pile pleine. Aucun processus ne peut empiler une valeur sur la pile.
  - *P_VIDE* : pile vide. Aucun processus ne peut dépiler une valeur de la pile.

Les fonctions de manipulation de la pile sont les suivantes :

- *void fs_push (int val):* permet à un processus d'empiler la valeur *val* au sommet de la pile, si la pile n'est pas pleine. Dans ce cas, la position pile_partage[0] qui sauvegarde ce sommet est mis à jour . Si la pile est pleine le processus s'endort en attendant qu'un autre processus dépile une valeur.

- *int fs_pop (void):* permet à un processus de dépiler une valeur entière du sommet de la pile, si la pile n'est pas vide. Cette valeur est renvoyée par la fonction. Dans ce cas, la position pile_partage[0] qui sauvegarde ce sommet est mis à jour . Si la pile est vide le processus s'endort en attendant qu'un autre processus empile une valeur.

Q2.1 – A votre avis, la priorité PPILE doit-elle être interruptible aux signaux ? Justifiez votre réponse **(1 point).**

Oui parce que se sont des fonctions appelées par le programme de l'usager. Le programme endormis dans un *push* ou un *pop* doit pouvoir être interrompu par un signal (ex. <ctrl>C).

Q2.2 – Programmez les fonctions *void fs_push(int val) et int fs_pop(void)* en utilisant les primitives *sleep*/*wakeup* ainsi que les données et constantes décrites ci-dessus. C'est à vous de choisir le (les) adresse(s) pour endormir les processus (**7 points**).

La position 0 de la pile contient la valeur ***1*** si la pile est vide et ***MAX_PILE +1*** si elle est pleine.
Pour la pile pleine le processus s'endort sur ***p*** et la pile vide sur ***(p+1)***

Obs. : le pointeur *p_pile* n'est pas vraiment nécessaire. On peut accéder directement à la variable *pile_partage*.

```
fs_push (int val) {
  int* p = u.p_pile ;

   while (*p == (MAX_PILE+1)) {
         /* pile pleine */
    flag_pile |= PILE_PLEINE ;
    sleep (ip,PPILE) ;
   }

   /* ajouter val au sommet de la pile */
    *(p + *p) = val;
   (*p)++;

   if ( flag_pile & PILE_VIDE) {
        /* réveiller les processus endormis sur pile vide  ( adresse p+1) */
       flag_pile &= ~PILE_VIDE ;
       wakeup (p+1) ;
   }

}
```

```
int fs_pop ( ) {

        int* p = u.p_pile ;
        int ret;

         while (*p == 1)) {
                /* pile vide */
           flag_pile |= PILE_VIDE ;
           sleep (ip+1,PPILE) ;
         }

         /* ajouter val au sommet de la pile */
          ret = *(p + *p);
          (*p)--;

         if ( flag_pile & PILE_PLEINE) {
            /* réveiller les processus endormis sur pile pleine  ( adresse p) */
            flag_pile &= ~PILE_PLEINE ;
            wakeup (p) ;
   }
       return ret ;

}
```

EXERCICE 3 – SIGNAUX : fonction SIGSUSPEND (3 points)

La fonction *int sigsupend (const sigst_t *pEns)* permet :

- l'installation du masque de signaux pointé par *pEns* jusqu'au retour de l'appel. Au retour le masque d'origine est réinstallé.
- La mise en sommeil du processus avec priorité PSLEP jusqu'à l'arrivée d'un signal non masqué.

La fonction renvoie toujours -1, avec la variable errno positionnée à la valeur EINTR.

Q3.1 – Programmez la fonction *int sigsuspend (const sigset_t *pEns).*

On ajoute la variable *sigset_t p_anc_hold* pour sauvegarder le masque des signaux original. Le programme va s'endormir sur une adresse (ex. *p_anc_hold*). On considère qu'aucun processus fera un *wakeup* sur cette adresse.

```
int sigsuspent (const sigset_t *pEns)
{
        p = u.u_proc ;

        /* sauvegarder masque de signaux courant */
        p->p_anc_hold = p->hold ;

        /* masque pour le sigsuspend */
        p->p_hold = pEns;
```

```
        sleep (&(p->p_anc_hold, SLEP);
        /* restaurer ancie masque */
        p_p_hold = p->p_anc_hold ;

        return -1;
}
```

Observations :

- Nous considérons les structures de données et fonctions données en cours pour le traitement signaux. Vous les trouverez aussi en annexe.
- Vous pouvez ajouter des nouveaux champs dans la structure PROC et USER.
- Le code d'erreur est affecté dans le champ *u_error* de la structure USER.

EXERCICE 4 – QUESTIONS DIVERSES (3 points)

Q4.1 – Structures PROC et USER :
Pourquoi les champs *p_sig* et *p_pri* sont déclarés dans la structure PROC et non pas dans la structure USER **(1 point)?**

Parce que se sont des informations qui doivent être toujours présentes en mémoire. Le vecteur PROC n'est pas « swappable. » Par contre, il se peut que la zone U (structure USER) ne se trouve pas en mémoire si le processus n'est pas en exécution.

Donnez des exemples des fonctions et/ou mécanismes du NOYAU qui ne marcheraient pas si ces champs étaient déplacés vers la structure USER. **(1 point)**

. p_sig :

fonction *kill* utilise *p_sig*. Un processus peut envoyer un signal à un deuxième même si celui-ci se trouve « swappé ». Donc la fonction *kill* doit pouvoir trouver le champ *p_sig* de n'importe quel processus.

. **p_pri**
fonction *psignal* (appelée par *kill*) : cette fonction a besoin du champ *p_pri* du processus que va recevoir le signal afin de le réveiller s'il est plus prioritaire. Il faut alors que le champ *p_pri* de ce processus soit un mémoire même si la zone U de ce processus se trouve swappée.

Fonction *setrun* (appelée par *wakeup*) : vérifie le champ *p_pri* pour savoir si le processus à réveiller est plus prioritaire que le processus courant. Il faut alors que le champ *p_pri* du processus à réveiller soit un mémoire même si la zone U de ce processus se trouve swappée.

Ordonnanceur (*switch ()*): parcourt du vecteur PROC en cherchant le processus plus prioritaire, même s'il se trouve swappé.

Q4. 2 - SIGNAUX (1 pont):
Nous savons que la réception des signaux n'est vérifiée (fonction issig) qu'au retour du mode système vers le mode utilisateur (retour appel système ou interruption). Est-il possible qu'au retour d'une interruption cette vérification ne soit pas faite? Justifiez votre réponse.

Oui, dans le cas des traitements d'interruptions imbriquées. Le retour d'une interruption ne va pas nécessairement basculer vers le mode usager. Par exemple, si une interruption horloge arrive pendant le traitement d'une interruption disque. Dans ce cas, le traitement de l'interruption disque est suspendu et le traitement de l'interruption horloge (plus prioritaire) a lieu. Au retour du traitement de l'interruption horloge le mode va continuer en système vu que le traitement de l'interruption disque sera repris.

ANNEXE

Socket

Code ClientUDP.c

```
…
#define PORTSERV 4567 /* Port du serveur */
#define IP_SERVEUR "192.168.0.5"

int main(int argc, char *argv[])
{
  int reponse;
  struct sockaddr_in dest;
  struct hostent *hp;
  int sock;
  int fromlen = sizeof(dest);


  if ((sock = socket(AF_INET,SOCK_DGRAM,0)) == -1) {
    perror("socket");
    exit(1);
  }
  /* Remplir la structure dest */

  bzero((char *)&dest,sizeof(dest));
  dest.sin_addr.s_addr = inet_addr(IP_SERVEUR);;
  dest.sin_family = AF_INET;
  dest.sin_port = htons(PORTSERV);


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

  /* Recevoir la reponse */
  if ( recvfrom(sock,&reponse,sizeof(reponse),0,0,&fromlen) == -1) {
    perror("recvfrom");
    exit(1);
  }
```

```
  printf("Reponse : %d\n", reponse);
  close(sock);
  return(0);
}
```

Code serveurUDP.c :

```
...
#define PORTSERV 4567
int main(int argc, char *argv[])
{
  struct sockaddr_in sin;  /* Nom de la socket du serveur */
  struct sockaddr_in exp;     /* Nom de l'expediteur */
  struct hostent *hp;
  int sc;
  int fromlen = sizeof(exp), cpt;

  /* creation de la socket */
  if ((sc = socket(AF_INET,SOCK_DGRAM,0)) < 0) {
    perror("creation socket");
    exit(1);
  }

  /* remplir le «nom» */
  bzero((char *)&sin,sizeof(sin));
  sin.sin_addr.s_addr = htonl(INADDR_ANY);
  sin.sin_port = htons(PORTSERV);
  sin.sin_family = AF_INET;

  /* nommage */
  if (bind(sc,(struct sockaddr *)&sin,sizeof(sin)) < 0) {
    perror("bind");
    exit(2);
  }
  for (;;) {

     /*** Reception du message ***/
     if ( recvfrom(sc,&cpt,sizeof(cpt),0,
                    (struct sockaddr *)&exp,&fromlen) == -1) {
       perror("recvfrom");
       exit(2);
     }

     /*** Traitement ***/
     cpt++;

     /*** Envoyer la reponse ***/
     if (sendto(sc,&cpt,sizeof(cpt),0,
              (struct sockaddr *)&exp,fromlen) == -1){
```

```
      perror("sendto");
      exit(2);
    }
  }
  close(sc);
  return (0);
}
```

Gestion des Signaux

Structures

- **Dans la zone U (struc user):**
  - u_signal[] : routines de traitements
  - u_sigmask[] : masque associé à chaque routine
  - ...
- **Dans struct proc :**
  - p_cursig : masque des signaux “pendants”
  - p_sig : signal en cours de traitement
  - p_hold : masque des signaux bloqués
  - p_ignore : masque des signaux ignorés

Obs : Ces champs sont de vecteurs de bits, où chaque bit correspond à un type de signal.

Lors d’un kill

- Chercher la structure proc du processus cible
- Tester *p_ignore*, si signal ignoré retourner directement
- Ajouter le signal dans *p_cursig*
- Si le processus est bloqué dans le noyau, le réveiller (rendre prêt)

⇨ 1 seul traitement pour plusieurs instances du même signal

appel à issig

- Vérifier les signaux positionnés dans p_cursig
- Vérifier si le signal est bloqué (test de *p_hold*)
- Si non bloqué mettre le numéro de signal dans *p_sig*
- renvoyer TRUE

⇨ Si issig retourne TRUE traiter le signal : appel de *psig*

psig

- Trouver la routine de traitement dans *u_signal* du processus courant
- Si aucune routine exécuter le traitement par défaut
- ...p_hold |= ...u_sigmask
- Appel de sendsig qui exécute la routine lors du retour en mode utilisateur.