Examen Module Noyau
Filière Système et Réseaux
Septembre 2004

3 heures - Tout document autorisé
Barème donné à titre indicatif

L. Arantes - P. Sens – G. Thomas

Exercice 1 : Socket (5 points)

1.1 : Repondez par oui ou par non. (1,5 points)

	TCP	UDP
1. on peut utiliser socket()		
2. on peut utiliser accept()		
3. on peut utiliser connect()		
4. on peut utiliser recvfrom()		
5. les envois ont une taille maximale		
6. les messages arrivent dans l'ordre d'émission		
8. peut être utilisé pour le multicast		
9. une socket de communication est toujours connectée		
10. une socket possède un port		

On souhaite écrire un serveur de nom : une machine cliente s'enregistre sur le serveur sous un nom (une chaîne de 8 caractères). Le serveur conserve alors l'association entre:

- le nom du client,
- l'adresse IP et le port du client,

Toutes les stations peuvent alors demander au serveur de convertir un nom en une adresse/port.

Le serveur doit traiter 2 types de messages : **`enregistre-nom`** qui enregistre un client et **`dereference-nom`** qui donne l'adresse et le port d'un client en fonction de son nom.

1.2 : Le client doit être capable de recevoir un message de type **nom** qui donne l'adresse et le port d'un client. On souhaite utiliser la même structure de message pour les 3 requêtes. Donnez (en C) la structure d'un message (appelée `struct msg_t`) ainsi que

les définitions de type de message (0,5 point).

1.3. : On considère que les deux fonctions : **`enregistre(struct msg_t)`** qui enregistre une machine chez le serveur et **`dereference(struct msg_t)`** qui donne l'adresse du client en fonction de son nom sont déjà écrites. Le serveur est écrit en UDP. Ecrivez le serveur. Vous utiliserez le port SERV_PORT pour la socket serveur (2 points).

1.4: Ecrivez un client qui s'enregistre sous le nom `tristan` puis cherche l'adresse et le port de `iseult` sur le serveur d'adresse 132.227.64.154 (1 point)

Exercice 2 : Threads (4 points)

Une barrière est un mécanisme de synchronisation. Elle permet que N threads prennent rendez-vous à un point donné de leur exécution. Quand un des threads atteint la barrière, il reste bloqué jusqu'à ce que tous les autres N-1 soient aussi arrivés à la barrière. Autrement dit, lorsque tous les N threads arrivent à la barrière, chacun des threads peut alors reprendre son exécution.

Le programme ci-dessous utilise le mécanisme de barrière pour synchroniser une N threads (N est passé en paramètre).

- La variable globale *bar_var* du type barrière (*struct barrier*) est utilisée pour synchroniser les threads. Elle est initialisée dans la fonction *thread_open_barrier* avec le nombre total de threads qui doivent se synchroniser sur la barrière.
- La fonction *thread_close_barrier ( )* remet cette variable à zéro tandis que la fonction *thread_sync_barrier ( )* permet qu'un thread se synchronise sur la barrière. Lorsque les *n* threads ont appelé cette fonction, les threads reprennent leur exécution.
- La fonction *pthread_open_barrier* et *pthread_close_barrier* sont appelées par le *thread main* qui ne participe pas à la synchronisation.

Nous considérons que les threads appellent correctement les trois fonctions et qu'il n'est donc pas nécessaire de renvoyer des codes de retour.

```
#define MAXTHREAD 10

/* structure barrière */
struct barrier {
  unsigned nb_proc; /*nombre total de threads total à synchroniser
*/
} bar_var;

/* initialiser la variable barrière bar_var*/
void thread_open_barrier (int n) {
     bar_var.nb_proc = n;
}

/* fermer la variable barrière */
void thread_close_barrier () {
     bar_var.nb_proc = 0;
}

void thread_sync_barrier () { /* a programmer }

void *fonc_thread(void *arg)
{
   printf("avant barrière \n");
   thread_sync_barrier ();

   printf("après barrière \n");
   pthread_exit(0);
}

int main(int argc, char *argv[]) {
  int i, N;  pthread_t tid[MAXTHREAD];

   /* nombre total de threads à synchroniser */
   N = atoi (argv[1]);

   printf ("ouvrir la variable barrière \n");
   thread_open_barrier (N);

  /* Creation des threads */
  for (i = 0; i < N; i++)
    if (pthread_create(&tid[i], NULL , fonc_thread, NULL) != 0 ) {
      perror("pthread_create"); exit(1);
    }

  /* Attendre la fin des threads */
   for (i= 0; i <  N; i++)
    if (pthread_join(tid[i], NULL) == -1){
      perror("pthread_join"); exit(1);
```

```
    }

   printf ("fermer la variable barrière \n");
   thread_close_barrier ();
   return (0);
}
```

Lorsque le programme est appelé avec N = 3, la sortie suivante est produite :

```
ouvrir la variable barrière
avant barrière
avant barrière
avant barrière
après barrière
après barrière
après barrière
fermer la variable barrière
```

2.1 : Programmez la fonction *thread_sync_barrier ( ).* Si vous désirez, vous pouvez ajouter d'autres champs à la structure *struct barrier.* Vous pouvez aussi modifier la fonction *thread_open_barrier (int n)* pour y ajouter d'autres initialisations .

Exercice 3 : Programmation dans le noyau (6 points)

Remarque : la question 3.5 est indépendante des autres

On souhaite implémenter un système de notifications qui prévient un processus dès qu'un fichier est ouvert.

Ce mécanisme repose sur un nouvel appel système sys_waitopen qui prend en paramètre le nom du fichier. L'appelant reste bloqué jusqu'à la prochaine ouverture du fichier.

Le prototype de la fonction est le suivant :

```
void sys_waitopen(char *filename);
```

3.1: Programmez la fonction noyau `sys_waitopen` et modifiez open (donné en annexe) (2 points).

3.2: On souhaite maintenant que `sys_waitopen` ne soit pas bloquant si le fichier est déjà ouvert. Modifier votre programme en conséquence (1 point).

3.3: Plusieurs processus peuvent attendre une notification d'ouverture sur le même

fichier. On souhaite que seul un des processus en attente soit réveillé en cas d'ouverture. Modifier le programme de la question 3.1 pour prendre en compte cette contrainte (1 point).

3.4: Votre programme fonctionne-t-il si l'inode du fichier dont on attend l'ouverture est déchargée sur disque après l'appel à `sys_waitopen` (justifier) ? Si ce n'est pas le cas, modifier le programme pour prendre en compte ce phénomène (1 point).

3.5: Brièvement, donnez le principe de fonctionnement d'une fonction qui afficherait la liste des noms de fichier ouverts par le processus courant (il n'y pas de code demandé dans cette question) (1 point).

Exercice 4 : Etude de codes dans le noyau (4 points)

On veut analyser la fonction `core` qui permet de sauvegarder le processus courant sur disque (ce qui est fait dans le cas d'un "core dumped" par exemple)

4.1: Donner l'algorithme de la fonction core

4.2: Donner l'algorithme de la fonction writei

ANNEXE

```
/*
 * open system call
 */
open()
{
   register struct inode *ip;
   register struct a {
      char  *fname;
      int   rwmode;
   } *uap;

   uap = (struct a *)u.u_ap;
   ip = namei(uchar, 0);
   if(ip == NULL)
      return;
   open1(ip, uap->rwmode, 0);
}


/*
 * common code for open and creat.
 * Check permissions, allocate an open file structure,
 * and call the device open routine if any.
 */
open1(ip, mode, trf)
register struct inode *ip;
register mode;
{
...
}


#define BSIZE   512            /* size of secondary block (bytes)
*/
#define BMASK   0777           /* BSIZE-1 */
#define BSHIFT  9              /* LOG2(BSIZE) */

core()
{
        register struct inode *ip;
        register unsigned s;
        extern schar();

        u.u_error = 0;
        u.u_dirp = "core";
```

```
        ip = namei(schar, 1);
        if(!access(ip, IWRITE) &&
           (ip->i_mode&IFMT) == IFREG &&
           u.u_uid == u.u_ruid) {
                itrunc(ip);
                u.u_offset = 0;
                u.u_base = (caddr_t)&u;
                u.u_count = ctob(USIZE);
                u.u_segflg = 1;
                writei(ip);
                s = u.u_procp->p_size - USIZE;
                u.u_base = 0;
                u.u_count = ctob(s);
                u.u_segflg = 0;
                writei(ip);
        }
        iput(ip);
        return(u.u_error==0);
}



/*
 * Write the file corresponding to
 * the inode pointed at by the argument.
 * The actual write arguments are found
 * in the variables:
 *      u_base          core address for source
 *      u_offset        byte offset in file
 *      u_count         number of bytes to write
 *      u_segflg        write to kernel/user/user I
 */
writei(ip)
register struct inode *ip;
{
        struct buf *bp;
        dev_t dev;
        daddr_t bn;
        register n, on;
        register type;

        if(u.u_offset < 0) {
                u.u_error = EINVAL;
                return;
        }
        dev = (dev_t)ip->i_un.i_rdev;
        type = ip->i_mode&IFMT;
        if (type==IFCHR || type==IFMPC) {
                ip->i_flag |= IUPD|ICHG;
                (*cdevsw[major(dev)].d_write)(dev);
```

```
                return;
        }
        if (u.u_count == 0)
                return;

        do {
                bn = u.u_offset >> BSHIFT;
                on = u.u_offset & BMASK;
                n = min((unsigned)(BSIZE-on), u.u_count);
                if(n == BSIZE)
                        bp = getblk(dev, bn);
                else
                        bp = bread(dev, bn);
                if(u.u_error != 0)
                        brelse(bp);
                else
                        bdwrite(bp);
                if(u.u_offset > ip->i_size &&
                   (type==IFDIR || type==IFREG))
                        ip->i_size = u.u_offset;
                ip->i_flag |= IUPD|ICHG;
        } while(u.u_error==0 && u.u_count!=0);
}
```

Structures de données utiles

```
struct  inode
{
   char    i_flag;
   char    i_count;     /* reference count */
   dev_t   i_dev;       /* device where inode resides */
   ino_t   i_number;    /* i number, 1-to-1 with device address */
   unsigned short  i_mode;
   short   i_nlink;        /* directory entries */
   short   i_uid;          /* owner */
   short   i_gid;          /* group of owner */
   off_t   i_size;         /* size of file */
   union {
      struct {
         daddr_t i_addr[NADDR];  /* if normal file/directory */
         daddr_t i_lastr;        /* last logical block read (for
read-ahead) */
      };
      struct  {
         daddr_t i_rdev;                /* i_addr[0] */
         struct  group   i_group;       /*  multiplexor group fi
le */
      };
   } i_un;
};
```

```
/* flags */
#define  ILOCK 01    /* inode is locked */
#define  IUPD  02    /* file has been modified */
#define  IACC  04    /* inode access time to be updated */
#define  IMOUNT 010   /* inode is mounted on */
#define  IWANT 020   /* some process waiting on lock */
#define  ITEXT 040   /* inode is pure text prototype */
#define  ICHG  0100     /* inode has been changed */


/*
 * The user structure.
 */


struct  user
{
...
        char    u_segflg;/* IO flag: 0:user D; 1:system; 2:user I
*/
        char    u_error;                /* return error code */
        short   u_uid;                  /* effective user id */
        short   u_gid;                  /* effective group id */
        short   u_ruid;                 /* real user id */
        short   u_rgid;                 /* real group id */
        int     *u_ap;                  /* pointer to arglist */
        caddr_t u_base;                 /* base address for IO */
        unsigned int u_count;           /* bytes remaining for IO
*/
        off_t   u_offset;               /* offset in file for IO
*/
        struct inode *u_cdir;           /* pointer to inode of
current directory */
        struct inode *u_rdir;           /* root directory of
current process */
        caddr_t u_dirp;                 /* pathname pointer */
…
};
```