Module Noyau
M1 – MASTER SAR
Novembre 2013
2 heures – Tout document papier autorisé
Barème donné à titre indicatif

L. Arantes, P. Sens, J. Sopena, G. Thomas

I. SAUT - (4 POINTS)

Considérez le code ci-dessous :

```
1: jmp_buf env;

2: void func1(void) {
3: if (setjmp(env))
4:       printf("func1: deroute \n");
5:  printf("func1: fin \n");
6: }

7: void func2(void) {
8:       func1();
9:       printf("func2: fin\n");
10: }

11: void func3(int a, int b) {
12:      printf("a + b = %d\n", a+b);
13: }

14: int main (int argc, char** argv) {
15:      printf("main : avant appel func2\n");
16:      func2();
17:      printf("main : après appel func2\n");
18:      func3(1,2);
19:      printf("main : après appel func3\n");
20:      longjmp(env, 1);
21:      printf("main: fin\n");
22:      return 0;
23: }
```

I.1. (1,5 point)

Donnez l'affichage de l'exécution du programme. Expliquez le déroulement du programme.

main : avant appel func2
func1: fin
func2: fin
main : après appel func2
a + b = 3
main : après appel func3
func1: deroute
func1: fin
Erreur de segmentation

func2 fait un longjmp vers le cadre de func1 qui n'existe plus

Considérez maintenant le code ci-dessous :

```
1: static jmp_buf env;

2: void func2(void) {
3:   longjmp(env, 1);
4:   printf("func2 : fin \n");
5: }

6: void func1(void) {
7:       jmp_buf my_env;
8:       memcpy(my_env, env, sizeof(jmp_buf));
9:       if (setjmp(env)) {
10:            printf("func1 : déroute \n");
11:            memcpy(env, my_env, sizeof(jmp_buf));
12:            longjmp(env, 1);
13:      }
14:      else {
15:            printf("func1 : avant appel func2\n");
16:            func2();
17:            printf("func1 : après appel func2\n");
18:      }
19:      memcpy(env, my_env, sizeof(jmp_buf));
20:      printf("func1 : fin \n");
21:  }

22:  int main() {
23:      if (setjmp(env)) {
24:            printf("main : déroute \n");
25:      } else {
26:            printf("main : avant appel func1\n") ;
27:            func1();
28:            printf("main : après appel func1\n");
29:      }
30:      printf ("main : fin \n");
31:      return 0 ;
32:  }
```

I.2. (1,5 point)

Donnez l'affichage de l'exécution du programme. Expliquez le déroulement du programme.

main : avant appel func1
func1 : avant appel func2
func1 : déroute
main : déroute
main : fin

Le main appelle func1 (lignes 6 et 7) ; func1 exécute le « else » en appelant func2 (lignes 21 et 22) ; func2 déroute vers func1 et exécute le « if » (ligne 16), rétablie le contexte du main et fait un longjmp vers le main (lignes 17 et 18) ; Le « if » du main est exécuté (ligne 4) et la fin (ligne 10).

I.3. (1 point)

Si on enlève la ligne 11 du code du programme, quel sera l'affichage ? Expliquez le déroulement du programme.

main : avant appel func1
func1 : avant appel func2
func1 : déroute
func1 : déroute
func1 : déroute
….
Le programme boucle dans le « if » de func1 (ligne 15 à 18) parce que le « env » du main n'a pas été rétabli.

II. SIGNAUX (10 POINTS, DONT 1 BONUS)

Dans cet exercice, on souhaite adjoindre un mécanisme permettant de masquer les signaux au code du noyau Unix v6 étudié en TD. Pour cela, on définit un nouveau champ dans la structure proc appelé p_mask. Masquer le signal n est alors équivalent à activer le bit n-1 du champ p_mask.

II.1. (0,5 point)

Rappelez la différence entre un signal masqué et un signal ignoré.

Signal masqué = signal temporairement retardé, signal ignoré = réception fait NO-OP

II.2. (1,5 point)

Donnez les codes des fonctions suivantes :

a. int is_masqued(struct proc* p, int n) : renvoie vrai si le signal n est masqué dans le processus p
b. void mask_signal(struct proc* p, int n) : masque le signal n du processus p
c. void unmask_signal(struct proc* p, int n) : démasque le signal n du processus p (cette fonction ne s'occupe que de désactiver le n-1 ième bit de p->p_mask)

```
int is_masqued(struct proc* p, int n) { return p->p_mask & (1<<(n-1) ) ; }
void mask_signal(struct proc* p, int n) { p->p_mask |= (1 << (n-1)) ; }
void unmask_signal(struct proc* p, int n) { p->p_mask &= ~(1 << (n-1)) ; }
```

II.3. (2 points)

Modifiez le code de la fonction issig() vu en TD pour que cette fonction ne renvoie vrai que si le signal en attente n'est pas masqué.

```
issig()                                                                    {
 register n;
 register struct proc *p ;
 p = u.u_procp;
 while(1) {
   n = fsig(p);
  if (n == 0) return (0);
   if ((u. u_signal [n]&1) == 0) {
      if(is_masked(p, n)) continue ;
       return(n);
   }
   p–>p_sig &= ~(1<<(n–1));
 }
```

```
}
```

II.4. (1 point)

Que doit-on modifier dans la fonction psignal pour prendre en compte les signaux masqués ?

Il ne faut pas appeler setrun si le signal est masqué

II.5. (1 point)

On suppose que les seules modifications apportées au noyau sont celles des questions 3 et 4. On suppose aussi que le processus p a reçu le signal n pendant qu'il était masqué. Après avoir démasqué le signal n, à quel moment n recevra-t-il le signal ?

A la fin de l'appel système unmask_signal. Il y a un if(issig()) psig() qui s'occupera de la réception.

On souhaite maintenant écrire une fonction sys_sigsuspend(int mask) qui met en place le masque de signaux mask avant d'endormir le processus. Ce dernier ne peut-être réveillé que par la réception d'un signal qui n'est pas masqué.

II.6. (0,5 points)

Lors d'un sys_sigsuspend, le processus doit-il s'endormir avec une priorité supérieure ou inférieure à PZERO ? Pour quelle raison ?

Supérieur, le but est bien de pouvoir réveiller le processus sur un signal

II.7. (2,5 points)

Donnez le code de sys_sigsuspend.

```
sys_sigsuspend(int mask) {
  int old_mask = u.u_procp->p_mask ;
  u.u_procp->p_mask = mask ;
  sleep(-1, 90) ; /* -1 est une adresse bidon jamais utilisée par un wakeup, 90 est la prio > 0 */
  u.u_procp->p_mask = old_mask ;
}
```

II.8. (1 point, bonus)

Que faut-il modifier d'autre dans le noyau pour pouvoir réveiller un processus qui dort pendant un sys_sigsuspend ?

Question difficile. Il faut modifier le if(issig()) goto psig du sleep de façon à ne pas sortir brutalement du noyau après la réception d'un signal. Pour éviter ce phénomène, le plus simple est d'ajouter un flag is_insigsuspend et ne faire le goto psig que si le is_insigsuspend est faux.

III. SYNCHRONISATION (6 POINTS)

Dans cet exercice on veut permettre à un programme d'obtenir un verrou sur un, et un seul, fichier parmi une liste de fichiers donnée. Lors de vos réponses, vous veillerez à respecter le nom des paramètres fournis dans le sujet.

III.1. (2 points)

Pour commencer il nous faut compléter les fonctionnalités de base du noyau. Donner l'implémentation de la fonction "`sleep_list`" qui endormira le processus jusqu'à ce **qu'au moins un wakeup** ait été fait **sur l'une des adresses** d'une liste donnée. Cette fonction devra avoir trois paramètres :

`size` : nombre d'adresses de la liste.
`list` : un tableau d'adresses
`pri` : priorité au réveil

Vous pouvez ajouter des champs aux différentes structures du noyau et modifier le code de la fonction wakeup en conséquence.

Pour simplifier, les processus endormis par cette nouvelle fonction ne pourront être réveillés par un signal (quelle que soit la priorité indiquée) mais ils seront toujours swappés si besoin.

REPONSE :

Dans la structure proc on ajoute deux champs :

```
struct proc
{
   .....
   caddr_t* p_wlist ;   /* liste d'adresses */
   int ps_wlist ;       /* nombre d'adresses de la liste */
   .....
}
```

Code du sleep_list :

```
int sleep_list (int size, caddr_t[] list, int pri) {
   s = gpl();
   spl(CLINHB);
   rp−>p_wlist = list ;
   rp−>ps_wlist = size ;
   rp−>p_stat = SSLEEP ;
   rp−>p_pri = pri ;
   spl(s);
   if (runin != 0 ) {
      runin = 0;
      wakeup(& runin) ;
   }
   swtch();
}
```

Modification du wakeup dans la boucle do/while on ajoute :

```
      ...
      ...
   if (p->p_wlist) {
      for (i=0;p->ps_wlist;i++) {
         if ( p−>p_wlist[i] == c ) {
            p->p_wlist=null;
            setrun (p) ;
            break ;
         }
      }
```

```
}
...
```

III.2. (2 points)

Donnez maintenant l'implémentation des appels système :
- `flock_list` : utilisant la fonction sleep_list pour verrouiller exactement l'un des fichiers d'une liste passée en paramètre (`size`: nombre de descripteurs, `list` : tableau de descripteurs). Cet appel système retournera le descripteur verrouillé.
- `funlock_list` : permettant de déverrouiller le fichier passer en paramètre (`file` : un descripteur)

Vous ferez attention au passage de paramètre des deux appels système ainsi qu'au retour du descripteur verrouillé.

REPONSE :

```
flock_list () {
   int size = u.u_arg[0] ;
   int[] desc = u.u_arg[1] ;
   struct inode * ip;
   int i = size ;
   caddr_t* list=malloc(coremap, size);
   while (true) {
      s = gpl() ;
      spl(BDINHB);
      for (i=0;i<size;i++) {
         ip=u.u_ofile[desc[i]]->f_inode ;
         if(~ip->i_flags & ILOCK) goto lock;
         ip->i_flags |= IWANT;
         list[i]=ip;
      }
      spl(s);
      sleep_list(size,list,PINOD);
   }

   lock :
      ip->i_flag |= ILOCK ;
      u.u_ar0[R0] = i;
      // On laisse les autres IWANT car retirer les autres IWANT peut être fatal a un autre processus.
}

funlock_list () {
   int file = u.u_arg[0] ;
   struct inode *ip =u.u_ofile[file]->f_inode ;
   ip->i_flags &= ~ILOCK ;
   if (ip->i_flags & IWANT) {
      ip->i_flags &= ~IWANT;
      wakeup ( (caddr_t) ip ) ;
   }
}
```

III.3. (2 points)

Donnez maintenant l'implémentation d'un appel système timed_flock_list qui prend un paramètre supplémentaire (`sec`: nombre de secondes) indiquant un maximum d'attente en secondes. Passé ce délai, l'appel système réveillera le processus et retournera -1.

REPONSE :
On doit modifier un peu le while du flock_list :

```
timed_flock_list () {
   ...
   ...
   int time_max = time + u.u_arg[3] ;
   while (time < time_max) {
      ...
      ...
      timeout (wakeup, list, (time_max-time)*HZ ) ;
   }
   u.u_ar0[R0] = -1;
   return ;
   lock :
      ...
}
```