Examen 1ere Partie Module Noyau
M1 – MASTER SAR
Novembre 2009
2 heures – Tout document papier autorisé
Barème donné à titre indicatif
Corrigé indicatif

L. Arantes, E. Encrenaz, S. Monnet, P. Sens, G. Thomas

I. SETJMP/LONGJMP (7 POINTS)

Lors de son exécution le programme ci-dessous présente une erreur de segmentation:

```
jmp_buf a_buff, c_buff;

1:  void c () {
2:    printf ("debut c \n");
3:    if (setjmp (c_buff) == 0) {
4:       printf ("longjmp a_buff \n");
5:       longjmp (a_buff, 1);
6:    }
7:    printf ("fin c \n");
8:  }

9:  void b() {
10:   c();
11: }

12: void a () {
13:   printf ("debut a \n");
14:   if (setjmp (a_buff) == 0) {
/* 15: printf ("appellant b\n"); */
16:     b();
17:   } else {
18:     printf ("longjmp c_buff \n");
19:     longjmp (c_buff, 1);
20:   }
21:   printf ("fin a \n");
21: }

22: int main (int argc, char *argv[]){
23:   a();
24:   printf ("fin programme \n");
25:   return 1 ;
26: }
```

I.1. (1,5 point)

Pourquoi le programme finit par crasher ?

Lorsque le longjmp a_buff de la ligne 5 est exécuté, le « frame » dans la pile référencé par c_buff est libéré et peut alors être invalidé par d'autres appels à fonction comme le printf de la ligne 18. Cela dit, on perd le contexte de c(). Par conséquent, lorsque le

programme fait un longjmp c_buff dans la linge 19 en déroutant vers la fonction c(), l'adresse de retour de c() sauvegardée dans la pile n'est plus valables.
Le programme se plante lors du retour de la fontion c().

I.2. (1,5 point)

Quels sont les affichages qui seront exécutés avant que le processus ne crashe ?

debut a
debut c
longjmp a_buff
longjmp b_buff
fin c
« Erreur de segmentation »

Considérez la fonction void func (int n) qui est appelée par le programme principal avec une valeur > 0 :

```
1: void func (int n){
2:  if (n != 0)
3:    func (n-1);
4 : printf ("n:%d - fin fonction \n",n);
5: }
```

Par exemple, pour le programme

```
6 : main () {
7 :  func(3) ;
8 :}
```

L'affichage sera :

```
n:0 - fin fonction
n:1 - fin fonction
n:2 - fin fonction
n:3 - fin fonction
```

I.3. (1,5 point)

Montrez l'état de la pile de ce programme (appel a func(3)) avant que « n:0 - fin fonction » ne soit affiché.

PILE :
bottom
3
@ ligne 7 : func(3)
2
@ ligne 3 : func(2)
1
@ ligne 3 : func(1)
0
@ligne 3 : func(0)
top

En utilisant les fonctions setjmp et longjmp, nous voulons modifier la fonction `func` afin que les messages « n:0 - fin fonction » et « n:1 - fin fonction » ne soient jamais affichés.

Par exemple pour le programme ci-dessus, la fonction n'affichera que « n:2 - fin fonction » et « n:3 - fin fonction ». Si le programme principal appelle `func(2)`, seul « n:2 - fin fonction » sera affiché et si il appelle `func(1)` aucun affichage ne sera fait.

Notez que le non-affichage des 2 messages doit être dû au déroutement effectué par un (des) longjmp(s) et la manipulation de la pile, et non pas dû à l'ajout de conditions sur le printf de la ligne 4, du type :

```
if (n> 1)
   printf ("n:%d - fin fonction \n",n);
```

I.4. (2,5 points)

Modifiez la fonction func (int n) en conséquence.

```
void func (int n){
 if (n==1)
  if (setjmp (buff))
   return;

 if (n != 0)
  func (n-1);
 else
  longjmp (buff,1);

printf ("n:%d - fin fonction \n",n);
}
```

II. COMMUNICATION PAR ENVOI DE SEGMENT (6 POINTS)

On souhaite construire un système de communication par envoi de messages au-dessus du noyau Unix étudié en TD. Tous les processus possèdent un segment de communication et, pour communiquer, le noyau offre à deux processus le moyen de s'échanger ces segments.

Le noyau offre donc deux nouvelles fonctions :

- void receive_segment() : endort le processus jusqu'à ce qu'un émetteur échange un segment avec lui.
- void send_segment(int pid) : attend que le processus identifié par pid soit prêt à recevoir le segment, puis échange son segment de communication avec pid.

Lorsqu'un processus P1 souhaite envoyer un message au processus P2, il copie son message dans son propre segment de communication. Ensuite, il appelle la fonction send_segment(P2). Cette fonction bloque l'émetteur en attendant que P2 puisse recevoir le segment. Dès que P2 exécute la fonction receive_segment, P2 s'endort et attend que P1 échange son segment de communication avec celui de P2. Lorsque l'échange est terminé, P1 et P2 sont libérés.

Pour vous aider à mettre en œuvre les fonctions d'envoi et de réception de segment, on suppose que vous disposez des fonctions suivantes :

- void swapCommunicationSegments(struct proc *p1, struct proc *p2) : inverse les segments de communication de p1 et p2
- struct proc *pid2proc(int pid) : renvoie la structure proc du processus identifié par pid

On suppose aussi que le drapeau de la structure proc d'un processus P (p_flag) peut stocker le bit suivant :

- SRECEIVE : P exécute receive_segment et attend un émetteur

II.1. (1,5 points)

Sur quelles adresses peuvent s'endormir P1 et P2 ?

P1 sur l'adresse de la structure proc de P2 + 1, et P2 sur sa structure proc +2
Observation : Cela marche aussi si P1 et P2 s'endorment sur la même adresse.

II.2. (4,5 points)

Donnez les codes de send_segment(int pid) et receive_segment(). On ne vous demande pas de gérer les erreurs.

```
void send_segment(int pid) {
 struct proc *proc  = pid2proc(pid) ;
 while( !proc→p_flag & SRECEIVE)
   sleep(proc+1) ;
 swapCommunicationSegments(u.u_procp, proc) ;
 proc→p_flag &= ~SRECEIVE;
 wakeup(proc+2) ;
}

int receive_segment() {
 u.u_procp→p_flag |= SRECEIVE ;
 wakeup(proc+1) ;
 sleep(proc+2) ;
}
```

III. SIGNAUX (7 POINTS)

On considère un système Unix dans lequel les processus s'exécutent en temps partagé. Les processus ont un quantum fixe de 1 seconde. L'interruption horloge cadence les traitements du système. Chaque interruption horloge induit un *tick*. La durée entre deux ticks consécutifs est de 100 ms.

Dans cet exercice, on s'intéresse à la manipulation des signaux. On rappelle que le vecteur des signaux postés est stocké dans le champ `p_sig` de la structure `proc`, et le vecteur des handlers associés aux signaux est stocké dans le champ `u_signal` de la structure `U`.

On considère ici que le champ `p_clk_tim` de la structure `proc` permet de décompter le nombre de **ticks** restants avant l'émission du signal `SIGALRM`.

On considère l'extrait de programme ci-dessous :

```
1      #include <unistd.h>
2      #include <stdio.h>
3      #include <stdlib.h>
```

```
4     #include <sys/signal.h>
5
6     void handler_alrm(){
7           printf(« ALARM !\n ») ;
8           kill(25683,SIGALRM) ;
9     }
10
11
12    void f(int a){
13          int b = 1 ;
14
15          while(a){
16                b *= a ;
17                …
18                a-- ;
19          }
20          return(b) ;
21    }
22    void main(){
23          int x,y;
24
25          signal(SIGALRM, handler_alrm);
26          … // affectation de x
27          alarm(2);
28          y = f(x);
29          printf("%d\n",y);
30    }
```

On considère aussi que le processus exécutant ce code a pour `pid` 25287. On suppose qu'aucun autre processus n'envoie de signaux à destination du processus 25287.

III.1. (1,5 point)

Quel est l'effet de l'instruction `signal` (ligne 25) sur les structures `p_sig` et `u_signal` du processus 25287 ? Donnez le code C de(s) l'instruction(s) affectant cette (ou ces) structure(s). Par quelle(s) fonction(s) du noyau ce traitement est-il réalisé ? Sur la pile de quel processus cet appel est-il réalisé ?

a) `signal(SIGALRM,handler_alrm)` positionne l'adresse de la fonction `handler_alrm` dans le champs `u_signal[SIGALRM]` et ne modifie pas `p_sig`.

b) Ce traitement correspond à :

```
u.u_signal[SIGALRM] = handler_alrm
```

c) Il est réalisé dans la fonction noyau `signal`

d) C'est un appel système, il est réalisé sur la pile (S) du processus ayant réalisé l'appel.

III.2. (1 point)

Quel est l'effet de l'instruction `alarm` (ligne 27) sur le champ `p_clk_tim` de la structure `proc` du processus 25287 ? Comment ce champ sera-t-il modifié par la suite ?

`alarm` affecte le champs `p_clk_tim` du processus courant au nombre de ticks à décompter avant l'échéance de 2 s ; la durée séparant deux ticks étant de 100ms, `p_clk_tim` est affecté à 20.

En fait dans les sources du TD, p_clk_tim est décrémenté à chaque secondes (fonction realtime()). Donc il est initialisé à 2s.

Ensuite, à chaque tick, `p_clk_tim` est décrémenté de 1.

Par rapport aux sources du TD, il est décrémenté à chaque seconde (realtime ())

III.3. (1,5 point)

Quand le signal `SIGALRM` sera-t-il émis ? Quelle est la suite d'actions réalisées par le noyau à l'instant de l'émission de `SIGALRM` : Vous préciserez les éventuelles modifications sur les structures `p_sig` et `u_signal` du (ou des) processus concerné(s). Donnez le code C de(s) l'instruction(s) affectant cette (ou ces) structure(s). Par quelle(s) fonction(s) du noyau ce traitement est-il réalisé ? Sur la pile de quel processus cet appel est-il réalisé ?

a) le signal `SIGALRM` sera émis sur le tick annulant le champs `p_clk_tim` du processus 25287. Le traitement correspondant à cette émission est le positionnement à 1 du bit correspondant au signal `SIGALRM` dans `p_sig` (du processus 25287) ; `u_signal` n'est pas modifié.

b) Ce traitement correspond à :

```
u.u_procp->p_sig |= 1 << (SIGALRM -1)
```

(le décalage de 1 entre le n° du signal et sa position dans le vecteur de bit est anecdotique)

En fait, realtime () parcourt proc[] est si p->p_clk_tim !=0 il le décremente et si = 0 il le signale : `p->p_sig |= 1 << (SIGALRM -1) (appel à psignal())`

c) Il est réalisé dans la fonction noyau `psignal (kill ())`

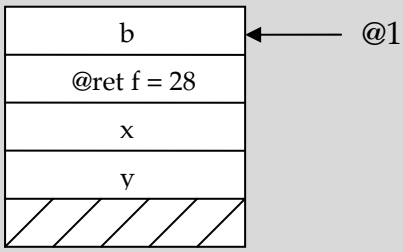
d) Ici, l'émission du signal par `psignal` est une conséquence d'un tick (donc interruption), il est réalisé sur la pile (S) du processus ayant subi l'interruption horloge.

On suppose que la durée d'exécution de la fonction `f` est de 5 secondes. Lors de sa première commutation, le processus 25287 est en train d'exécuter `f`, dans la boucle `while`.

III.4. (1 point)

Décrivez le contexte du processus 25287 lorsqu'il est évincé du CPU lors de sa première commutation : vous représenterez la pile d'exécution avant l'éviction (pile U), et indiquerez les valeurs des registres pointeur de code et pointeur de pile sauvegardées lors de son entrée en mode S.

Lors de la première commutation du processus 25287, le contexte sauvegardé est le suivant :



IP : 16 (ou 17, ou 18)
SP : @1

L'adresse de retour de `f` est sur la ligne 28, mais correspond aux instructions assembleur de récupération du résultat de `f` et affectation dans `y`.

On suppose que le signal `SIGALRM` destiné au processus 25287 survient **après** qu'il ait quitté le CPU suite à sa première commutation et **avant** qu'il y accède une deuxième fois.

III.5. (1 point)

Du point de vue de la gestion des signaux, quelle est la suite d'actions réalisées par le noyau lorsque le processus 25287 accède au CPU pour la seconde fois ? Vous décrirez précisément les modifications du contexte d'exécution du processus avant son retour en mode U, en décrivant sa pile d'exécution, ainsi que les valeurs des registres pointeur de code et pointeur de pile qui seront restaurés lors de son retour du mode S.

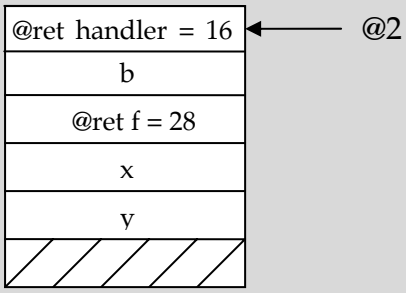
Lorsque le processus 25287 accède au CPU la seconde fois, son contexte est restauré par `swtch`, puis juste avant le passage du mode S à U, la séquence

```
if (issig()) psig()
```

est réalisée.

Le champs `p_sig` contient au moins un signal à traiter (`SIGALRM`), et donc `psig`() va modifier le contexte du processus pour faire *comme si* le handler du signal à traiter avait été appelé juste avant la commutation ; de plus `psig()` remet à 0 le bit correspondant au signal traité dans le champ `p_sig`.

Le contexte créé est le suivant :



IP = @handler_alrm
SP = @2

III.6. (1 point)

A quelles conditions l'appel système `kill` exécuté ligne 8 induira-t-il l'émission d'un signal `SIGALRM` ?

Si le processus 25683 existe et que les deux processus appartiennent au même utilisateur.