Conservatoire National des Arts et Métiers
FOAD Ile De France

Introduction au Noyau Linux

Exercices et Travaux Pratiques
20 septembre 2020

Tous droits réservés.

Ce document est un support de cours à l'usage exclusif des auditeurs du Cnam dans le cadre de leur formation. Tout autre usage est interdit sans l'autorisation écrite du Cnam.

Sommaire

Première partie : Processus & Ordonnanceurs

Exercice 1 : Espace mémoire des Processus

Soit le programme C suivant :

```
int    indice = 1;
char   *ptn;
...

void sprog(int i, int j)
{
  char str[5] = "abcde";
  int k = 3;
  ...
  ptn = (char*) malloc (5 * sizeof (char));
  strncpy(ptn, str, 5);
  return;
}

int main(int argc, char *argv) {
  ...
  sprog(128, 456);
  ...
}
```

1) Dans quelle zone de l'espace processus vont se ranger les variables suivantes :

argv
argc
indice
ptn
i,j
str
k

2) Sur quelle zone de l'espace processus va pointer le pointeur `"ptn"`

- Réponses :

```
1)
argv -> Environnement
argc -> Environnement
indice -> Data
ptn -> Bss
i,j -> pile
str -> pile
k -> pile

2) ptn pointe sur le Tas.
```

TP 1 : Consultation Mémoire

- Pour connaitre la mémoire disponible dans le système il faut utiliser la commande **`free`** :

```
free -m
              total        used        free      shared  buff/cache   available
Mem:           3907         569        2661          19         676        3083
Swap:          3907           0        3907
```

** `m` pour mega-octet*

- La taille des 3 zones Code, Données et BSS est fixée à la compilation, l'option **-m64** du compilateur C permet d'afficher ces tailles.

```
$ cc -m64 toto.c -o toto
$ size toto
```

Exercice 2 : Création de Processus

C'est l'appel système `fork()` qui permet à un processus de créer un processus fils.

Le processus fils reçoit une copie exacte de l'espace mémoire de son père.

Le père et le fils se partagent le même code source, c'est le retour du `fork()` qui va permettre de déterminer si on est le père (retour = `PID` du fils crée) ou si on est le fils (retour = `0`) pour départager les traitements.

Si l'on veut séparer les codes du père et du fils il faut utiliser une primitive `exec()` laquelle permet de transférer le processus courant vers un autre code source.

Programme processus.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>

main() {
int pere;
int fils;

pere = getpid();
printf ("Processus : Je suis le pere, mon PID est %d\n",pere);

// Création de processus
fils = fork();

// PERE
if (fils != 0) {
  printf ("Processus : Je viens de creer le fils : %d\n", fils);
  while (1) {};
}

// FILS
else {
  printf ("Processus : je suis le fils : %d\n", getpid());
  execlp ("./mini",NULL);
}}
```

Script mini:

```
echo "Mini : je suis le nouveau code du processus $$"
sleep 100
echo FINI
```

Exercice 3 : Création de Thread

C'est l'appel système `pthread_create()` qui permet à un processus de créer un thread. Le code du nouveau thread est donné dans une fonction. Lors de la création du premier thread le code du processus se transforme en code de thread (le thread originel).

Un thread est identifié par le numéro de son processus `PID` + un numéro `LWP` (light-weight process), en parallèle un thread se voit attribuer un `identifiant Posix` (cf. Normes). C'est la commande `ps -L` qui permet d'afficher les identifiants de processus + les identifiants de processus légers `LWP`.

Les appels système relatifs aux threads sont délicats à manipuler et en particulier :
1) Il faut compiler les programmes avec l'option `-pthread`
2) L'appel système `gettid()` (obtenir `LWP`) a besoin de `<sys/syscall.h>` lequel fait appel à `<asm.unistd.h>`, il faut donc disposer de `/usr/include/asm.unistd.h` (sous Suse en particulier il faut installer le package `linux-kernel-headers`)

Programme thread.c :

```
#define _GNU_SOURCE         /* or _BSD_SOURCE or _SVID_SOURCE */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/syscall.h>
int pere;

/**********************************/
/* THREADS                        */
/**********************************/
static void *code_thread (void* arg) {
pid_t tid;
pthread_t pthr;
  tid = syscall(SYS_gettid);
  pthr = pthread_self();
  printf( "Je suis le thread %u (ID Posix %u) créé par le processus
%d \n",(unsigned int)tid, (unsigned int)pthr, pere);
  while(1) {}
}

/**********************************/
/* PROCESSUS PERE                 */
/**********************************/
int main() {
int i;
pthread_t p; // identification Posix du thread
pere = getpid();

  for (i = 0; i < 3; i++) {
    if (pthread_create(&p, NULL,&code_thread, NULL) != 0)
      printf( "Je suis le processus %d, faute lancement thread \n",
pere );
    else printf( "Je suis le processus %d, j'ai lancé le thread Posix
%u \n", pere,(unsigned int)p );
  }
  while(1) {}    // Boucle sans Fin
}
```

TP 2 : Consulter et modifier le quantum `nice`

La commande **`ps -l`** permet d'afficher le **`nice`** des processus (colonne **`NI`**).

Un utilisateur peut modifier positivement le **`nice`** d'un de ses processus en cours d'exécution à l'aide de la commande **`renice`**, seul le super utilisateur peut modifier négativement le **`nice`** d'un processus.

Reprenez le script mini :

```
echo "Mini : je suis le nouveau code du processus $$"
sleep 100
echo FINI
```

Lancez le et utilisez la commande **`ps -l | grep mini`** pour connaitre son quantum et son PID.

Augmentez son quantum : **`renice 10 PID`** et vérifiez le résultat obtenu avec la commande **`ps`**.

Exercice 4 : Rattacher un processus à un ordonnanceur temps réel

C'est l'appel système `sched_setscheduler()` qui permet de rattacher un processus à un ordonnanceur ; il faut fournir à `sched_setscheduler()` un pointeur sur une structure `sched_param` contenant les caractéristiques souhaitées pour le processus (priorité, quantum ...).

Programme scheduler.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sched.h>

int main() {
int i = 0;
int fils = 0;
int pere;
int politique;
struct sched_param param;

  // PROCESSUS PERE
  pere = getpid();
  printf("Je suis le pere mon numero est : %d\n", pere );
  politique = sched_getscheduler(pere);
  if (politique == SCHED_FIFO) printf("Politique père FIFO\n");
  else if (politique == SCHED_RR) printf("Politique père
Tourniquet\n");
  else if (politique == SCHED_OTHER) printf("Politique père Temps
partagé\n");
  else printf("Politique père Inconnue %d\n", politique);

  // PROCESSUS FILS
  if ((fils = fork()) == 0){
  fils = getpid();
  printf( "Je suis le fils %d. Mon pere est : %d\n", fils , pere );
  param.sched_priority = 10;

  // PASSAGE AU SCHEDULER FIFO
  if (sched_setscheduler(fils,SCHED_FIFO,&param) == -1) printf("Faute
setscheduler\n");
  else {
    politique = sched_getscheduler(fils);
    if (politique == SCHED_FIFO) printf("Nouvelle Politique fils
FIFO\n");
    else if (politique == SCHED_RR) printf("Nouvelle Politique fils
Tourniquet\n");
    else if (politique == SCHED_OTHER) printf("Nouvelle Politique
fils Temps partagé\n");
    }
  }

  // CODE COMMUM
  //while(1){};         //Boucle sans fin
}
```

Attention : Ce programme est à lancer en tant que root.

Exercice 5 : Ordonnancement

On considère 4 processus Linux temps réel A, B, C et D dont les caractéristiques sont les suivantes :

Processus	Temps d'exécution	Priorité
A	6 unités	1
B	4 unités	1
C	14 unités	1
D	2 unités	1

Les 4 processus sont à l'état "prêt" à l'instant "t0" et dans l'ordre A (premier arrivé) puis B puis C puis D (dernier arrivé) dans la file d'attente.

1) Les processus sont rattachés à l'ordonnanceur SCHED_FIFO : Donnez l'ordre d'exécution des processus et leur temps de réponse.

2) Les processus sont rattachés à l'ordonnanceur SCHED_RR, avec un quantum de 2 unités chacun : Représentez l'exécution des processus et donnez leur temps de réponse.

3) Les processus sont rattachés à l'ordonnanceur SCHED_FIFO et on attribue des priorités différentes aux processus : Donnez l'ordre d'exécution des processus et leur temps de réponse.

Processus	Temps d'exécution	Priorité
A	6 unités	2
B	4 unités	3
C	14 unités	1
D	2 unités	4

4) Les processus sont rattachés à l'ordonnanceur SCHED_RR avec toujours un quantum de 2, avec des priorités différentes et les 4 processus parviennent à des temps différents dans la file d'attente des processus prêts : Représentez l'exécution des processus et donnez leur temps de réponse.

"t0" démarre l'unité de temps 0, on suppose qu'un processus qui entre dans la file d'attente à "t0 + x" arrive juste à temps pour être prêt à l'unité "x".

Processus	Temps d'exécution	Priorité	Entrée dans la file d'attente
A	6 unités	1	t0 + 4 unités
B	4 unités	2	t0
C	14 unités	3	t0 + 2 unités
D	2 unités	3	t0 + 6 unités

- Corrigé

1)

Processus	Ordre d'exécution	Temps de réponse
A	1	6
B	2	10
C	3	24
D	4	26

2)

P	1									10										20							Temps de réponse
A	x	x							x	x					x	x											16
B			x	x							x	x															12
C					x	x							x	x			x	x	x	x	x	x	x	x	x	x	26
D							x	x																			8

3)

Processus	Ordre d'exécution	Temps de réponse
A	2	20
B	3	24
C	1	14
D	4	26

4)

P	1									10										20							Temps de réponse
A					x	x	x	x	x	x																	10
B	x	x	x	x																							4
C											x	x			x	x	x	x	x	x	x	x	x	x	x	x	26
D													x	x													14

Seconde partie : Interruptions & Signaux

Compléments de cours : Interruptions microprocesseur x86

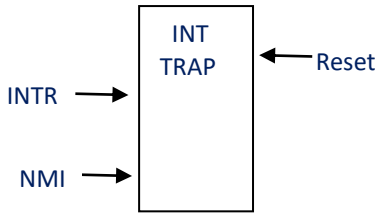
- Reset : Réinitialisation du processeur, non masquable :
  - RAZ registres
  - Compteur ordinal à l'adresse FFFFFFF0 (adresse de boot)

- NMI : Interruptions périphériques non masquables (coupure secteur, faute de parité mémoire ...)

- INTR : Interruptions périphériques masquables (périphériques, cf. contrôleur d'interruption ci-dessous)

- TRAP : Exceptions ou "trap" (division par zéro, changement de page ...), non masquable, générée par le microprocesseur.

- INT : Interruption logicielle masquable, générée par programmation et donc par le microprocesseur (instruction int).



Rappel : Contrôleur d'interruption

Les interruptions périphériques (toutes masquables) passent par un composant "contrôleur d'interruption" lequel gère 16 entrées d'interruptions :

- IRQ0 – Programmable Interval Timer
- IRQ1 – keyboard controller
- IRQ2 –
- IRQ3 – 8250 UART serial port COM2 and COM4
- IRQ4 – 8250 UART serial port COM1 and COM3
- IRQ5 – Intel 8255 parallel port LPT2
- IRQ6 – Intel 82072A floppy disk controller
- IRQ7 – Intel 8255 parallel port LPT1
- IRQ8 – real-time clock (RTC)
- IRQ9 –
- IRQ10 –
- IRQ11 –
- IRQ12 – Intel 8042 PS/2 mouse controller
- IRQ13 – math coprocessor
- IRQ14 – hard disk controller 1
- IRQ15 – hard disk controller 2

- L'interruption levée par le contrôleur est dirigée vers l'entrée INTR du microprocesseur.
- Le microprocesseur doit acquitter une interruption pour pouvoir recevoir la suivante (instruction EOI[1]), en cas d'interruptions en attente, le contrôleur remonte les interruptions vers le microprocesseur dans l'ordre de leur priorité.

- Pour des raisons historiques[2] l'ordre décroissant des priorités est :
  IRQ0, IRQ1 puis IRQ8 à IRQ15 puis IRQ3 à IRQ7

[1] End Of Interrupt
[2] Jusqu'à récemment il y avait 2 contrôleurs d'interruptions en cascade sur les plateformes PC.

Liste complète des interruptions Linux

INT 0-31 : TRAP (Exceptions) et NMI (Interruption non-masquable matérielle) :

Int	Masquable	Ring	Source
0	non	0	Division Overflow
1	non	0	Single Step (debug)
2	non	0	NMI (Interruption non-masquable matérielle)
3	non	3	Breakpoint (debug)
4	non	0	Overflow
5	non	0	Bound (out of range)
6	non	0	Invalid Machine Code
7	non	0	Coprocessor Not Available
8	non	0	Double Fault
9	non	0	Coprocessor Segment Overrun
10	non	0	Invalid Task State Segment
11	non	0	Segment Not Present
12	non	0	Stack Overflow
13	non	0	General Protection Error
14	non	0	Page Fault
15	non	0	reserved
16	non	0	Coprocessor Error
17	non	0	Instruction non alignée
18	non	0	Faute CPU ou BUS mémoire
19	non	0	Faute opération de types "flottants"
20-31	non	0	Réservées à des nouvelles fonctionnalités ...

INT 32-47 : INTR (Interruptions masquables matérielles) :

Int	Masquable	Ring	Correspondance Contrôleur PC
32	oui	0	IRQ0 System Timer
33	oui	0	IRQ1 Keyboard
35	oui	0	IRQ3 Serial Port #2
36	oui	0	IRQ4 Serial Port #1
37	oui	0	IRQ5 Parallel Port #2
38	oui	0	IRQ6 Floppy Controller
39	oui	0	IRQ7 Parallel Port #1
40	oui	0	IRQ8 Real Time Clock
41	oui	0	IRQ9 available
42	oui	0	IRQ10 available
43	oui	0	IRQ 11 available
44	oui	0	IRQ 12 Mouse
45	oui	0	IRQ13 Coprocessor ERROR line
46	oui	0	IRQ 14 Hard Drive Controller
47	oui	0	IRQ15 available

INT 48-255 : INT (Interruptions masquables logicielles)

Int	Masquable	Ring	Source
...			
128	oui	3	Appel système (le numéro du service est passé dans le registre A)
...			

Liste complète des signaux Linux

Numéro	Mnémonique	Signification
1	SIGHUP	Fin de session (fermeture terminal)
2	SIGINT	Arrêt par utilisateur (CTRL/C)
3	SIGQUIT	Arrêt et dump mémoire
4	SIGILL	Instruction illégale
5	SIGTRAP	Trace
6	SIGABRT	Instruction IOT ou abort
7	SIGEMT	Instruction EMT
8	SIGFPE	Exception arithmétique
9	SIGKILL	Arrêt non masquable
10	SIGBUS	Bus error
11	SIGSEGV	Violation de mémoire
12	SIGSYS	Erreur appel système
13	SIGPIPE	Ecriture dans un pipe sans lecteur
14	SIGALRM	Alarme de l'horloge
15	SIGTERM	Arrêt
16	SIGURG	Information urgente sur socket
17	SIGSTOP	Demande de suspension
18	SIGTSTP	Demande de suspension depuis le terminal
19	SIGCONT	Demande de reprise de process
20	SIGCHLD	Terminaison d'un process fils
21	SIGTTIN	Lecture au terminal en background
22	SIGTTOU	Ecriture au terminal en background
23	SIGIO	Occurence d'événement scruté
24	SIGXCPU	Temps CPU maximal écoulé
25	SIGXFSZ	Taille maximale de fichier atteinte
26	SIGVTALRM	Horloge virtuelle
27	SIGPROF	Horloge
28	SIGWINCH	Changement de taille de fenêtre
29	SIGINFO	
30	SIGUSR1	Signal 1 utilisateur
31	SIGUSR2	Signal 2 utilisateur

** Attention : La numérotation des 16 premiers signaux est standard, pour les autres signaux cela dépend des implémentations. Il faut donc utiliser les mnémoniques par sécurité.*

TP 3 : Signaux

Signaux de suspension, reprise et arrêt des processus

Pour envoyer un signal il faut utiliser la commande kill, par défaut kill PID envoie le signal 15 autrement il faut préciser, par exemple kill -9 PID pour envoyer le signal 9 au processus PID.

Pour masquer ou récupérer un signal (se protéger contre l'action par défaut) utilisez la commande trap, le signal 9 ne peut pas être masqué ou récupéré, c'est l'arme absolue pour tuer n'importe quel processus.

Les appels systèmes correspondantes sont kill() et signal().

Ci dessous un processus boucle qui est une boucle sans fin qui affiche une lettre A toutes les secondes.

Script "boucle" :

```
echo $$      #afficher PID
while true   #boucle sans fin
do
       echo A
       sleep 1
done
```

Script "boucle" avec masquage du signal 15 :

```
trap "" 15
echo $$
while true
do
       echo A
       sleep 1
done
```

Script "boucle" avec traitement du signal 15 :

```
int15() {    #traitement du signal 15
       echo "arrêt refusé"
       exit
}
trap "int15" 15 #début du programme
echo $$
while true
do
       echo A
       sleep 1
done
```

- Lancez le processus boucle dans un terminal et ensuite testez les signaux à partir d'un autre terminal :

```
$kill PID          # SIGTERM ARRET
$kill -9 PID       # SIGKILL ARRET ABSOLU
$kill -17 PID      # SIGTSTOP SUSPENSION ABSOLUE
```

```
$kill -18 PID      # SIGTSTP SUSPENSION
$kill -19 PID      # SIGCONT REPRISE
```

- Vous pouvez aussi arrêter votre processus boucle à partir de son propre terminal avec CTRL/C (équivalent SIGINT) ou en fermant son terminal (équivalent SIGHUP)