Systèmes d'exploitation
TD 2: Processus et Threads

Frédéric Vivien, bureau 306, e-mail: fvivien

11 février 1997

1 Processus

1.1 Fonctions de gestions des processus

- **int getpid(void)**: retourne l'identificateur du processus appelant.
- **int getppid(void)**: retourne l'identificateur du processus père.
- **int sleep(int sec)**: provoque la suspension du processus pendant *sec* secondes.
- **int fork(void)**: crée un nouveau processus, copie conforme du processus appelant, et en retourne l'identificateur. Les deux processus partagent le même segment de code et le segment de données statiques est dupliqué dès que le processus fils le modifie.

  Le processus fils hérite d'un double de tout les descripteurs de fichiers ouverts du père: si le fils en ferme un, la copie du père n'est pas changée, par contre les pointeurs de fichiers associés sont partagés (si le fils se déplace dans un fichier, la prochaine manipulation du père se fera à la nouvelle adresse).

  Exercice: faire tourner le programme *fork1.c* que vous récupérerez à l'URL:
  http://www.ens-lyon.fr/~fvivien/Enseignement/SE/fork1.c.
- **int execl(char * path, char * arg0, ..., char * argn, NULL)**: permet d'exécuter un programme dont le nombre d'arguments est connu. Le programme en train de s'exécuter est remplacé par le programme dont le nom est pointé par *path*, *arg0*, ..., *argn* étant les paramètres passés au programme. L'habitude sous **UNIX** veut que l'on utilise le nom du programme comme premier paramètre (*arg0*).
- **int execv(char * path, char ** argv)**: même chose avec un nombre quelconque d'arguments.
- autres variantes de **exec**: se référer au *man*.

1.2 Exercice

Écrire un petit interpréteur de commandes. Ce programme lira un caractère au clavier codant pour une fonction: *A* pour afficher l'heure; *B* pour afficher les personnes connectées à la machine; *C* pour afficher la liste des processus; *D* pour arrêter le processus. Une fois le caractère codant lu, le processus créera un processus fils dont il remplacera le code par celui de la fonction désirée.

1.3 Signaux

Un signal est une interruption. Si on ne précise rien, la réception d'un signal par un processus provoque la fin de ce processus.

La fonction *signal* permet de demander d'ignorer un signal ou d'indiquer un sous-programme qui sera exécuté en cas d'arrivée d'un signal. Dans ce dernier cas, lorsqu'un signal arrive pour le processus en cours, ce dernier est interrompu et le sous-programme indiqué est exécuté. La fin du sous-programme provoque le retour à l'endroit où le processus avait été interrompu.

Lorsqu'un signal a été traité, l'action qui lui est associé n'est pas ré-initialisée à l'action par défaut qui est la fin du processus, ceci doit être fait explicitement (voir la fonction *signal*). Les signaux sont:

SIGHUP	1	hangup
SIGINT	2	interrupt (rubout)
SIGQUIT	3	quit (ASCII FS)
SIGILL	4	illegal instruction (not reset when caught)
SIGTRAP	5	trace trap (not reset when caught)
SIGIOT	6	IOT instruction
SIGABRT	6	used by abort, replace SIGIOT in the future
SIGEMT	7	EMT instruction
SIGFPE	8	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10	bus error
SIGSEGV	11	segmentation violation
SIGSYS	12	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal from kill
SIGUSR1	16	user defined signal 1
SIGUSR2	17	user defined signal 2
SIGCLD	18	child status change
SIGCHLD	18	child status change alias (POSIX)
SIGPWR	19	power-fail restart
SIGWINCH	20	window size change
SIGURG	21	urgent socket condition
SIGPOLL	22	pollable event occured
SIGIO	*SIGPOLL*	socket I/O possible (SIGPOLL alias)
SIGSTOP	23	stop (cannot be caught or ignored)
SIGTSTP	24	user stop requested from tty
SIGCONT	25	stopped process has been continued
SIGTTIN	26	background tty read attempted
SIGTTOU	27	background tty write attempted
SIGVTALRM	28	virtual timer expired
SIGPROF	29	profiling timer expired
SIGXCPU	30	exceeded cpu limit
SIGXFSZ	31	exceeded file size limit
SIGWAITING	32	process's lwps are blocked
SIGLWP	33	special signal used by thread library
SIGFREEZE	34	special signal used by CPR
SIGTHAW	35	special signal used by CPR
SIGCANCEL	36	thread cancellation signal used by libthread

Fonctions de manipulations des signaux:

- **void (*signal (int sig, void (*disp)(int)))(int)**: définit le traitement effectué à la réception du signal *sig*. Trois cas sont possibles:
  - Si *disp* est *SIG_DFL*, l'action par défaut pour le signal *sig* est mise en place. Le processus termine.
  - Si *disp* est *SIG_IGN*, le signal *sig* est ignoré par le processus (les signaux *SIGKILL* et *SIGSTOP* ne peuvent être ignorés).
  - Une autre valeur de *disp* donne l'adresse d'un sous-programme qui sera exécuté lors de la réception du signal *sig*. Ce sous-programme à un argument qui est celui du signal pour lequel il est activé.

Lorsqu'un signal a été reçu, le traitement d'une autre réception de ce signal n'est pas remis à la valeur par défaut. Si on désire revenir au traitement standard, il faut rappeler la fonction ***signal*** avec le paramètre ***disp*** positionné à ***SIG_DFL***.

L'utilisation de la fonction ***signal*** dans un programme nécessite l'insertion du fichier ***signal.h***.

- **int pause(void)**: met le processus en attente de l'arrivée d'un signal.
- **int kill(int pid, int sig)**: envoie le signal ***sig*** au processus d'identificateur ***pid***. Si ***sig*** est nul, la fonction se contente de vérifier l'existence d'un processus d'identificateur ***pid***.

1.4 Exercices

1. Modifier le programme de la partie 1.2 pour que les commandes soient passées à l'interpréteur au moyen de signaux et non plus de caractères tapés au clavier.
2. Écrire un programme qui mime un système producteur-consommateur: le processus père –le producteur– crée un processus fils –le consommateur– puis se met à "produire". À chaque fois qu'il produit, il envoie un signal au consommateur, lui indiquant qu'il y a un nouveau produit à consommer. Le consommateur consommera tant qu'il y aura des produits et sinon attendra de nouvelles productions. Les temps de production et de consommation pourront être régis par des durées choisies aléatoirement.

2 Threads

2.1 Introduction

Jusqu'à présent, nous considérions qu'un processus était un espace d'adressage dans lequel on exécutait un programme séquentiel. Le multithreading permet d'avoir plusieurs flots de contrôle concurrents, appelés ***threads***, dans un même processus, qui partagent le même espace d'adressage.

Le multithreading est une des caractéristiques majeures des systèmes d'exploitation modernes. Il est apparu pour tirer partie des machines multiprocesseurs (par exemple en assignant un ***thread*** par processeur) et pour gérer plus simplement les interfaces graphiques (par exemple un ***thread*** étant créé pour traiter un événement).

2.2 Gestion des *thread*

Le code d'un ***thread*** est défini par une fonction C. C'est la routine `thr_create` qui permet de créer un ***thread*** à partir d'une fonction C. Le prototype de la fonction est le suivant :

```
thr_create(void *stack,size_t stack_size,
           void *(*start_routine)(void*),void *arg,
           long flags, thread_t *new_thread);
```

Dans un premier temps, considérer seulement `start_routine` qui est l'adresse de la fonction, `arg` qui est le paramètre réel à lui passer, et `new_thread` qui est l'identificateur du ***thread*** créé retourné par la routine. Vous pouvez mettre tous les autres paramètres à `NULL`.

La fonction `thr_self` permet d'obtenir l'identificateur du ***thread*** courant.

```
thread_t thr_self(void);
```

Pour sortir d'un ***thread***, utiliser la fonction `thr_exit` :

```
void thr_exit(void *status);
```

Pour attendre la fin des ***threads***, utiliser la fonction `thr_join` :

```
int thr_join(thread_t wait_for, thread_t *departed, void *status);
```


Voici un exemple simple qui utilise les ***threads***. Ce programme demande de manière asynchrone des paramètres à l'utilisateur :

```
#include <thread.h>
#include <stdio.h>
#include <stdlib.h>

void * ask(int *result)
  {int i;
    printf("Entrer un entier : "); scanf("%d",result);
    thr_exit(0);
  }

void main()
  {int result;
   thread_t thr;

   thr_create(NULL,NULL,ask,&result,NULL,&thr);

   /*  Faire autre chose d'intéressant */

   thr_join(thr,NULL,&status);

   /* On peut se servir du nombre result rentré par l'utilisateur */
  }
```

Vous pouvez récupérerez ce code à l'URL: http://www.ens-lyon.fr/˜fvivien/Enseignement/SE/thread.c.

2.3 Exercice

La ligne de commande pour compiler un programme utilisant les ***threads***, nommé par exemple `essai_thr.c`, est :

```
lapin%cc -D _REENTRANT essai_thr.c -lthread
```

Programmer le problème classique du producteur et du consommateur, cette fois-ci avec des threads.