Master Ingénierie du Logiciel Libre – 2ème Année (M2I2L)
Module système, réseaux et sécurité
Année 2009-2010
TP N°3 : gestion mémoire, processus, interruptions/exceptions, entrées/sorties

D. DUVIVIER (duvivier@lil.univ-littoral.fr) – LIL/ULCO

Ce document est constitué en partie d'une liste de commandes, extraites des transparents « de cours ». Toutes les recommandations/vérifications reprises sur les transparents de cours ne figurent pas dans ce document.

1. Manipulation des modules (Rappel)

- lsmod : liste l'ensemble des modules présents en mémoire
- insmod : insère individuellement un module en mémoire

  Il est préférable d'utiliser la commande modprobe indiquée ci-dessous
- rmmod : supprime un module de la mémoire
- depmod : crée une liste des modules utilisables avec le noyau
- modprobe : insère un module en mémoire, ainsi que toutes ses dépendances

2. Mémoire et processeur

Pour le processeur, une commande « cat /proc/cpuinfo » ou la commande « procinfo » vous donneront quelques informations. Pour la mémoire, une commande « cat /proc/meminfo » ou la commande « memstat » vous donneront quelques informations. Vous pouvez bien évidemment utiliser des commandes comme « top » ou « htop ». Certaines commandes additionnelles (comme « cpufreq-info ») donnent des informations sur la fréquence du processeur.

Pour obtenir la carte mémoire d'un processus, il suffit de taper « cat /proc/<N°_PID>/maps. Pour le processus en cours, il faut utiliser « cat /proc/self/maps ».

3. Compilation (rappel)

Pour compiler un code source « mon_programme.c » en un exécutable nommé « mon_programme », utilisez :

gcc mon_programme mon_programme.c

Vous pouvez aussi utiliser la commande suivante :

make mon_programme

4. Utilitaire pour parser/analyser les options sur la ligne de commande

man 3 getopt

5. Quelques documentations à consulter

Documentations utiles :

- paquet manpages - Pages de manuel pour le système GNU/Linux
- paquet manpages-dev - Pages de manuel sur l'utilisation de GNU/Linux pour le développement
- paquet manpages-fr - Version française des pages de manuel sur l'utilisation de GNU/Linux
- paquet manpages-fr-dev - Version française des pages de manuel pour le développement
- paquet manpages-fr-extra - Version française des pages de manuel
- paquet linux-manuel-2.6.30 - Explique l'API du noyau linux

Consultez les pages de manuel ; la section des pages de man est parfois précisée entre parenthèses :

- Il suffit d'invoquer « man » ou « man <n°_de_section> » sur : exec (voir execl, execlp, execle, execv, execvp, execve, fexecve) ; fork(2) ; vfork ; pthread_create(3) ; credentials(7) ; wait ; waitpid ; waittid ; wait4(2); getpid ; getppid ; pthreads ; gettid ; getrlimit ; clone(2) ; kill(2) ; ptrace(2) ; sigaction(2) ; sigemptyset(3) ; sigpending(2) ; sigprocmask(2) ; sigsuspend(2) ; signal(2) ; signal(7) ; exit(3) ; on_exit(3) ; atexit(3) ; tmpfile(3) ; sleep(3) ; alarm(2) ; nanosleep(2) ; getitimer(2) ; setitimer(2) ; timer_create(3) ; timer_delete(3) ; timer_getoverrun(3) ; timer_gettime(3) ; timer_settime(3) ; ualarm(3) ; time(7) ; perror ; err(3) ; errno(3) ; error(3) ; strerror(3) ; umask(2) ; chmod(2) ; mkdir(2) ; open(2) ; close(2) ; stat(2) ; mknod(2) ; pipe(2) ; utime(2) ; read(2) ; write(2) ; chown(2) ; fcntl(2) ; fsync(2) ; ioctl(2) ; lseek(2) ; mkfifo(3)
- Une lecture seine pour programmer correctement : emacs /usr/include/stdlib.h

6. Prêt pour le TP3 ?

Pour ce TP, j'ai mis cinq fichiers à votre disposition :

- makefile - Fichier utilisé pour la compilation des 4 programmes ci-dessous
- tpi2ldd1execl.c - Test de execl()
- tpi2ldd1fork.c - Test de fork()
- tpi2ldd1signal.c - Test de signal()
- tpi2ldd1wait.c - Test de wait()

Le programme « tpi2ldd1fork.c » est juste destiné à illustrer l'utilisation de la fonction « fork() ». Remarquez qu'à l'aide de la variable « pid » et des fonctions « getpid() » et « getppid() », le processus père peut connaître son pid et celui de son fils, le processus fils peut également connaître son pid et celuis de son père (ppid). Grâce à ces informations ces processus peuvent par exemple s'échanger des signaux via la fonction « kill() ».

Le programme « tpi2ldd1wait.c » illustre l'utilisation de la fonction « wait() » pour permettre (dans cet exemple) au processus père d'attendre la fin d'exécution de son processus fils.

Le programme « tpi2ldd1execl.c » illustre l'utilisation de « execl() » pour remplacer dynamiquement le code du processus fils par un autre code exécutable. Remarquez bien qui (du processus père et/ou du processus fils) exécute la fin du code du programme « tpi2ldd1execl.c » en vous basant sur les affichages placés dans le code.

Le programme « tpi2ldd1signal.c » illustre l'interception des signaux par un gestionnaire de signaux. Pour le tester, lancez le programme en arrière plan : ./testi2ldd1signal &

Puis utilisez les commandes suivantes :

- kill -SIGKILL <N°_pid_du_père>
- kill -SIGTERM <N°_pid_du_fils>

Étudiez le comportement des processus père et fils selon l'ordre dans lequel ces processus sont tués. Parvenez-vous à faire apparaître un processus « zombie » ? Si oui, comment faites vous pour détecter ce processus zombie ? Quand cela se produit-il ?

A partir des programmes sus-nommés, il s'agit pour vous de réaliser une synchronisation en alternance ou alternée (assimilée à une partie de « ping-pong ») : deux processus (un processus père et un processus fils dans un premier temps) échangent des signaux à l'image d'un match de ping-pong :

1. le processus père envoie un signal SIGUSR1, simulant le « ping » à son processus fils
2. le processus fils – après un certain délai – retourne un signal SIGUSR2, simulant le « pong » à son processus père
3. le processus père – après un certain délai – retourne à l'étape 1. ci-dessus.

Dans un premier temps nous pouvons supposer que chaque processus boucle sur un appel à la fonction sleep() et que cette boucle va être périodiquement interrompue par le gestionnaire de signaux propre à chaque processus (père et fils) lors de la réception d'un signal.

Problème : comment réagit la fonction sleep() (qui est un appel système) lors de la réception d'un signal ? Est-elle insensible au signal ? Est-elle interrompue ? Comment le savoir ? Y-a-t-il un code de retour associé à cette situation ?

Une fois la situation clarifiée, ajoutez les tests et/ou boucles nécessaire pour s'assurer que les échanges de signaux ne deviennent pas trop fréquents (dans le cas où la fonction sleep() serait effectivement sans arrêt interrompue par des signaux).

Dans une première version des programmes, vous pouvez supposer que la gestion de la synchronisation en ping pong est partiellement gérée dans le programme principal et partiellement gérée dans le gestionnaire de signaux.

Dans une seconde version, tout doit être géré par les gestionnaires de signaux et les deux processus doivent uniquement comporter une boucle sur un appel à sleep() dans le code principal (pour simuler un calcul ou une requête en cours de traitement). Pour éviter les attentes (sur un appel à sleep()) au sein même du gestionnaire de signaux, vous pouvez utiliser la fonction alarm() pour générer une temporisation qui va « réveiller » votre gestionnaire de signaux (en utilisant éventuellement un autre signal) afin de renvoyer le signal attendu, au second processus, après un certain délai conformément à ce qui est demandé dans l'énoncé.

Dans une troisième version, les programmes « ping » et « pong » sont deux programmes indépendants (l'un ne lance pas l'autre via un appel à fork()). Le premier processus lancé est le processus pong qui se met en attente d'un signal SIGUSR1. Le second processus lancé est le processus ping avec en paramètre le pid du processus pong (pour permettre d'envoyer le signal SIGUSR1 au bon processus (pong). A partir de là le ping pong commence !

7. Correction partielle

Pour cette correction, j'ai mis quatre fichiers à votre disposition :

- makefile - Fichier utilisé pour la compilation des 3 programmes ci-dessous
- consommateursig.c - Consommateur de signaux (surtout ne lancez pas ce programme directement)
- producteursig.c - Producteur de signaux
- tpi2ldd1pingpongsig.c - Ping-pong de base non correctement programmé (sleep dans les handlers)

En fait ces programmes ne donnent pas une correction complète du TP, mais comportent tous les éléments pour y parvenir. En effet, producteursig et consommateursig montrent comment envoyer des signaux entre deux programmes « presque indépendants » car le consommateur est lancé par le producteur. Le programme tpi2ldd1pingpong réalise bien une synchronisation en ping pong mais les deux processus sont liés par un appel à fork. De plus, les gestionnaires de signaux comportent des appels à sleep() ce qui est intolérable car le programme principal est bloqué. Or, nous voulons que la synchronisation en alternance (ping pong) soit aussi transparente que possible pour les processus impliqués dans la synchronisation et principalement utilisé pour effectuer une tâche de calcul ou gérer des requêtes si ce sont des serveurs. A vous de compléter/corriger !

Voilà, c'est terminé !