Introduction

/*

* ----------

* « LICENCE BEERWARE » (Révision 42):

* Remi Dubourgnoux a créé ce fichier. Tant que vous conservez cet avertissement,

* vous pouvez faire ce que vous voulez de ce truc. Si on se rencontre un jour et

* que vous pensez que ce truc vaut le coup, vous pouvez me payer une bière en

* retour.

* ----------

*/

Ce cours est réalisé à l'intention des élèves de GI de l'IUT d'Informatique de Clermont-Ferrand dans le cadre du cours de Systèmes et réseaux du semestre 3.

Date de modification	Auteur	Version
18/09/2012	Remi Dubourgnoux	0.3
12/10/2012	Remi Dubourgnoux	0.4

Chapitre 1

Création d'un processus avec la fonction «fork();»

Rappels sur les processus, PID, UID, et la commande ps

Chaque Utilisateur possède un identifiant d'utilisateur ou UID (User IDentificator). C'est en fonction de l'uid que les droits d'utilisateur sont définis.

Chaque programme en exécution possède un identifiant de processus ou PID (Process IDentificator) et l'UID de l'utilisateur l'exécutant lui est associé.

Sous UNIX on peut consulter la liste des processus avec la commande ps.

1. ps -x Permet d'obtenir l'ensemble des processus en cours d'exécution.
2. ps -aux Permet de voir l'ensemble des processus du système.
3. l'option -f Permet de consulter les attributs des processus
4. Pour plus d'informations voir : man ps

La fonction de prototype :

pid_t getpid(void);

Renvoie comme son nom l'indique le pid du processus dans lequel est appelé cette fonction.

La fonction de prototype :

uid_t getuid(void);

Renvoie l'uid de l'utilisateur du processus dans lequel est appelé cette fonction.

Note : Set-UID (SUID)

set-uid est une permission propre aux exécutables binaires qui permet l'exécution de ce programme avec l'uid du propriétaire du fichier exécutable et donc les permission de lecture, d'écriture ou d'exécution de cet utilisateur..

Par exemple si root créé un programme avec la propriété set-uid et que vous exécutez ce programme, vous l'exécuterez avec les privilèges de root .

Pour définir cette propriété il faut faire un :

chmod +s

sur l'exécutable binaire pour lequel vous souhaitez définir cette propriété.

Fork

La fonction fork(); permet de créer un nouveau processus (donc un nouveau PID) depuis le programme en cours d'exécution. Le processus ainsi créé possède le même code source que le programme dans lequel il est appelé il continuera donc l'exécution d'une copie du code du programme l'ayant créé.

On parle souvent de processus père et de processus fils. Car le père engendre/créé ses fils.

Prototype de la fonction fork :

pid_t fork (void);

La fonction renvoie :

- -1 en cas d'erreur.
- 0 pour le processus fils.
- Le pid du fils après le lancement de ce dernier.

On peut donc filtrer la sortie avec un switch ou des if pour définir les comportements du fils et du père et gérer les erreurs.

IMPORTANT

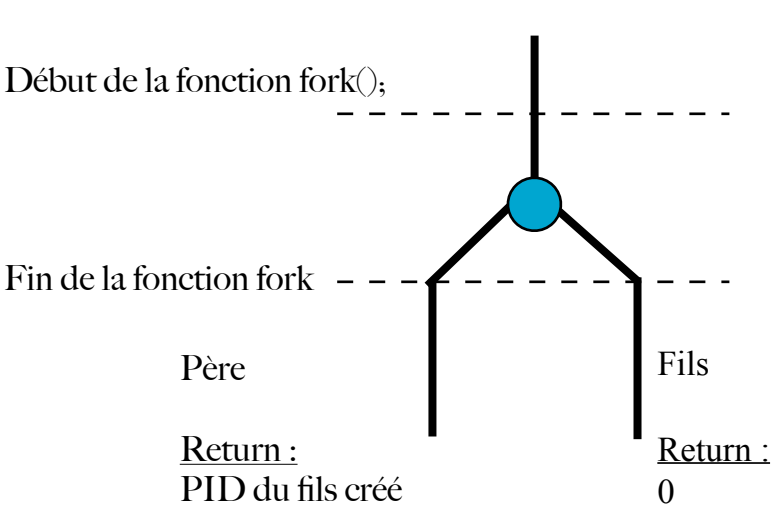
Lorsque le père se termine avant le fils ce processus ne disparait pas complètement il devient «zombie». Pour permettre a ce processus de disparaitre le père doit exécuter la fonction :

wait() ;

il attendra donc la fin du/des processus fils avant de se terminer lui même.

Le père doit donc exécuter autant de wait que de fils.

Note : Pensez au #include <unistd.h>



Clonnage du code

La fonction fork(); commence,
le code est copié a partir de ce point et la fonction en cours d'exécution (fork) l'est aussi.
Chaque processus rends donc une valeur de retour de fork.
Le fils renvoi 0.
Le père renvoi le PID du fils créé par fork.

Chapitre 2

Recouvrement du code source et lancement d'un programme : fonctions exec et system.

Notion : Recouvrement

Les fonctions de la famille exec (execl, execlp, execv, ...) «recouvrent» le code source du programme dans lequel elles sont appelées. La fonction system appelle une fonction à l'intérieur d'un programme C.

C'est à dire que le programme en cours d'exécution **devient** le programme ou script shell **appelé par la fonction, a la place** d'exécuter son code source.

Rappels sur les arguments en ligne de commande

La fonction main d'un programme peut prendre des arguments, ces arguments sont indiqués à la suite du nom de l'exécutable. Les arguments sont séparés par un espace.

Ex :
$ ./monProgramme arg1 arg2 .... argN

Arguments de la fonction main en C :
• char* argv[] : Chaque case du tableau contient un argument sous forme d'une chaine de caractères.

int argc : nombre d'arguments passés à la fonction main.

Attention le premier argument passé à la fonction main est le nom de l'exécutable.

Exemple :

$ ./afficheArg a b c

argc vaut 4

argv[0] vaut ./afficheArg //Nom de l'exécutable

argv[1] vaut a //Premier Argument
argv[2] vaut b //Deuxième argument
argv[3] vaut c //Troisième argument

Appel système exec

Les fonctions de la famille exec (execl, execlp, execv, ...) «recouvrent» le code source du programme dans lequel elles sont appelées, c'est à dire que le programme en cours d'exécution devient le programme ou script shell appelé par la fonction, a la place d'exécuter le code source qui suit l'appel de cette fonction (si la fonction ne renvoie pas d'erreur).

Voir : man exec

Précisions :

Pour execl les argument sont donnés un a un séparés par une virgule.

Pour execv l'argument est un tableau.

IMPORTANT :

L'ordre des argument est le même que pour ceux du main : Nom, arg1, arg2, ..., argN , NULL.

Le **NULL** en dernier élément est **OBLIGATOIRE.**

Ces arguments sont passés à la méthode main appelée (ou au script shell en question).

Rappel : La variable $PATH

Les exécutables sont placés dans un des répertoires du PATH.
Le PATH est une variable système/globale qui indique où sont les programmes exécutables sur votre ordinateur. Si vous tapez echo $PATH vous aurez la liste de ces répertoires.

PATH : une liste des répertoires qui contiennent des exécutables que vous souhaitez pouvoir lancer sans indiquer leur répertoire. Si un programme se trouve dans un de ces dossiers, vous pourrez l'appeler quel que soit le dossier dans lequel vous vous trouvez.

Il vous suffit donc de déplacer ou copier votre script dans un de ces répertoires, comme /bin, /usr/bin ou /usr/local/bin (ou encore un autre répertoire du PATH). Notez qu'il faut être root pour pouvoir faire cela.

Le PATH peut aussi être modifié en rajoutant le chemin absolu d'un de vos dossier contenant des exécutables. (Chaque chemin est séparé par "**:**").

La fonction system

La fonction system permet de lancer un programme dans un programme C sans utiliser fork et exec.

on l'appelle avec la syntaxe :
system ("*maCommandeShell*");

Problèmes de sécurité liés a "exevp" et "system"

Les fonctions execvp et system peuvent présenter une menace de sécurité dans le cas d'un application SUID (Set User IDentificator).
En effet ces fonctions cherchent les exécutables par le PATH.
Le problème est que si l'utilisateur rajoute au début de son PATH un dossier contenant des exécutables portant le même nom que ceux qui seront appelés via l'application en SUID, ces dernières seront appelées à la place des fonctions que vous avez appelées.

Exemple :
J'ai remarqué en faisant un ls -l que l'exécutable est en SUID
$ ls -l
-rwsr-sr-s 1 root root 4361 Oct 8 15:57 programmeExecutable

Je remarque egalement que lors de l'utilisation de cette application la console est effacée à un moment donné. En plaçant /users/remi/mesBin/ au début de mon path et en plaçant un script shell nommé "clear" dans ce dossier, ma fonction sera exécutée à la place de clear dans "programmeExecutable" si la fonction utilisée pour cette appel est une fonction qui utilise le PATH. En effet les exécutables sont cherchés dans les dossiers du PATH en commençant par le premier chemin présent puis en continuant dans les chemins suivants.

Chapitre 3

Threads Posix

Pointeurs de fonction

Un pointeur de fonctions en C est une variable permettant de designer une fonction C.
Comme n'importe quelle variable, elle peut être utilisée comme une variable ou comme argument.
On déclare un pointeur de fonction comme un prototype de fonction, mais on ajoute une étoile (*)devant le nom de la fonction.

Threads Posix sous Linux

1. Un thread c'est quoi ?

Un Thread ou *fil d'exécution* en français est une fonction qui se déroule en parallèle à l'exécution du programme.

• Premier intérêt :
Lancer un programme sans monopoliser la console.
L'utilisateur peut alors interrompre le calcul sans taper un ctrl-C

• Autre Intérêt :
Effectuer un traitement parallèle sur un processeur multi-coeurs.

Les fonctions de threads sont dans la bibliothèque pthread.h et on doit complier avec la lib pthread.a.

$ gcc monProg.c -o monProg -lpthread

2. Création de threads et attente de terminaison

Library :

```
#include <pthread.h>
```

Création d'un thread :

Pour créer un thread il faut créer une fonction à exécuter dans ce thread.

Cette fonction doit avoir pour prototype :

```
void* maFonction (void* monArgument);
```

La fonction pthread_create lance le traitement de la fonction passée en argument via un pointeur de fonction.

pthread_create à pour prototype :

```
int pthread_create(pthread_t* thread, pthread_attr_t* attributs, void*(*maFonction)(void* monArgument), void* arg);
```

1.Premier argument : Passage par adresse du thread.
2. 2° argument : attributs. Désigne les arguments du thread et on peut donner la valeur NULL pour utiliser les attributs par défauts.
3. 3° argument : pointeur vers la fonction à exécuter.
4. 4° arg est l'argument de la fonction passée par pointeur.

Retour de la fonction :
Si l'exécution se déroule sans problèmes la fonction pthread_create() retourna 0.
Sinon la fonction retourne un numéro d'erreur indiquant l'erreur qui a eu lieu.

Attente d'un thread :

Le processus exécutant le main (père) est un thread, c'est le thread principal. Il peut attendre la fin de l'exécution d'un autre thread avec la fonction pthread_join (correspondant a la fonction wait() dans le fork()).
On récupère ainsi le retour de la fonction exécutée par le 2eme thread.

Le prototype de pthread_join est le suivant :

```
int pthread_join(pthread_t thread, void **thread_return);
```

Données Partagées et exclusion mutuelle

Lors d'un fork les variables (y compris globales, etc) sont copiées (et donc utilisées par chaque processus) puis chaque processus (fils et père) travaille sur ses propres variables.

Dans le cas d'un thread la mémoire est partagée, les variables globales sont partagées entre les différents threads s'exécutant en parallèle. On doit donc protéger les données contre des accès simultanés.

Par exemple empêcher ces cas :
• Deux thread essayent en même temps de modifier une variable globale.
• Un tread modifie une structure de données pendant qu'un autre essaye de la lire.

Pour cela on utilise un mécanisme d'exclusion mutuelle : le «mutex».
De type : **pthread_mutex_t**
Un thread peut verrouiller un mutex avec pthread_mutex_lock(); pour accéder à une variable ou un flot (fichier, stdin, stdout, ...).
Une fois cet accès terminé on déverrouille le mutex avec la fonction pthread_mutex_unlock();.

Si un thread tente de déverrouiller un mutex verrouillé par un autre thread il attends simplement que ce thread libère le mutex.

Le prototype de cette fonction est :
```
int pthred_mutex_lock (pthread_mutex_t *mutex);
```

Attention a ne pas verrouiller deux fois le même mutex. Sinon le thread sera bloqué **définitivement.**

```
int pthred_mutex_unlock (pthread_mutex_t *mutex);
```

Sémaphores

Rappel : section critique

Une section critique est une portion de code dans laquelle il doit être garanti qu'il n'y aura jamais plus d'un thread/processus simultanément.

Il est nécessaire d'utiliser des sections critiques lorsqu'il y a accès à des ressources partagées.

Une section critique peut être protégée par un mutex , un sémaphore, ou d'autres mécanismes d'exclusion.

Un thread/processus qui utilise la section critique, il doit la libérer au plus vite, puis-ce que d'autre threads/processus peuvent attendre la libération de cette section critique. Libérer la section critique permet d'améliorer les performances en permettant aux threads/processus en attente d'y accéder.

Rappel : atomicité, opérations atomiques

Une opération atomique est un ensemble d'instructions qui s'exécutent «d'un bloc» sans qu'on puisse interrompre cette exécution. On doit attendre la fin de ces opérations pour pouvoir intervenir.

Les opérations atomiques sont donc utilisées pour vérifier l'accès à une ressource partagée, en effet notre but et d'empêcher que deux threads demandent la vérification de l'accès à la ressource simultanément.

Si deux threads veulent vérifier l'accès à une ressource en même temps ce n'est plus possible car l'opération est atomique, elle ne peut pas être exécutée deux fois simultanément.

Utilisation des semaphores :

#include <semaphore.h>

Un semaphore est une variable de type sem_t, c'est un **compteur** contenant un entier positif ou nul. Il existe 4 opération principales sur les semaphores.

On peut représenter un semaphore comme une **boite de jetons** :

Lorsqu'il y a des jetons, on peux en prendre dans la boite. sinon nous devons attendre que quelqu'un place a nouveau des jetons dans la boite pour pouvoir en prendre.

2 solutions :

- Attendre devant la boite (sem_wait) le prochain jeton sera a nous.
- Faire autre chose (continuer son traitement en dehors de la section critique) (sem_trywait).

Nous pouvons mettre des jetons dans la boite. (sem_post).

Nous pouvons regarder combien il y a de jetons dans la boite. (sem_getvalue).

Prototypes :

Creation / init : sem_init

```
/*      Initialisation d'un semaphore
 *      Arguments :
 *      sem : pointeur sur le semaphore à initialiser
 *      pshared : (process shared) indique si le semaphore doit etre partagé entre threads (0) ou entre processus (!=0)
 *      value : nombre de jetons disponibles apres l'initialisation/ valeur de départ du compteur.
 *      retour :
 *      sem_init() returns 0 on success; on error, -1 is returned, and errno is set to indicate the error.
 */
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Attente d'un jeton : sem_wait

```
/*      Attente d'un jeton puis décrementation du compteur
 *      Argument :
 *      sem : pointeur sur le semaphore à décrementer
 */
int sem_wait(sem_t *sem);
```

Essai d'Attente d'un jeton : sem_trywait

```
/*      Décrementation du compteur, si ce n'est pas possible errno est defini a EAGAIN.
 *      Cette opération n'est pas bloquante. (le traitement continue en dehors de la section critique.
 *      Argument :
 *      sem : pointeur sur le semaphore à décrementer
 */
int sem_trywait(sem_t *sem);
```

Incrémentation du compteur : sem_post

```
/*      Incrémentation du compteur
 *      Argument :
 *      sem : pointeur sur le semaphore à incrémenter
 */
int sem_post(sem_t * sem);
```

Obtenir la valeur du compteur : sem_getvalue

```
/*      sauvegarde à l'emplacement pointé  par  sval
        la valeur courante du compteur du sémaphore sem.

 *      Argument :
 *      sem : pointeur sur le semaphore dont on veux obtenir la valeur.
 *      val : pointeur sur un entier qui contiendra la valeur du semaphore.
 */
int sem_getvalue(sem_t * sem, int * sval);
```