Université de Strasbourg

S31 Sujet N°1

Comprendre les Systèmes

gancarski@unistra.fr

IUT Robert Schuman Université de Strasbourg

Processus

Objectif : Comprendre la création et synchronisation de processus père/fils

Notions : Appels système de création de processus

Tout est processus ...

Dans les systèmes modernes, tout est processus : une commande shell, une application utilisateur (Navigateur, ...), la gestion du réseau, la compilation d'un programme, un serveur WEB...
Ainsi une commande (par ex. `"ls -il"`) est toujours exécutée par un processus créé pour cela :

1. Lecture de l'entrée (clavier par exemple) par le shell associé à la console (`bash` par exemple)
2. Vérification par celui-ci de l'existence d'un fichier exécutable correspondant (ici : `/bin/ls`)
3. Duplication du processus shell : `fork`.
4. Le processus fils remplace son code par celui contenu dans le fichier de la commande : `exec`
5. Le processus ainsi modifié exécute alors la commande
6. A la fin, il disparait et le processus shell reprend la main

1 Fork

L'appel système `fork()` permet la création d'un processus fils copie conforme de son père en dupliquant intégralement l'espace d'adressage :

→ Copie "physique" de l'espace d'adressage du père ⇒ même code, mêmes données (exceptée la valeur de retour de la fonction fork), mêmes registres, fichiers ouverts ....

→ À la suite de cet appel, il existe deux processus dont les deux seules différences sont leurs pids et la valeur de retour du `fork()` (processus fils : 0, processus père : pid du processus fils créé)

Le fichier *unistd.h* fourni d'autres fonctions telles que :

— `getpid()` : renvoie le pid du processus (type pid_t, défini dans unistd.h).
— `getppid()` : renvoie le pid du processus père.

Le fichier (`td1_ex1.1.c`) sur Moodle contient un exemple d'utilisation de `fork` :

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

int main(int argc, char *argv[]){
int i;
int E = 100;
 if (( i = fork()) == 0) {
   printf(" Je suis le fils , mon PID : %d \n",getpid());
   printf(" Je suis le fils , le PID de mon pere %d \n",getppid());
   E = E + 4;
   printf(" Je suis le fils : terminé \n");
 } else {
```

```
    printf(" Je suis le pere , mon PID  : %d d\n",getpid());
    printf(" Je suis le père , le PID de mon fils vaut : %d \n",i);
    E = E + 4;
    printf("  Je suis le pere : terminé  \n");
  }
}
```

1.1 - Lors de l'exécution, la sortie à l'écran a ressemblé à ceci :

```
Je suis le pere,mon PID  : 66566
Je suis le père,le PID de mon fils vaut : 66568
Je suis le pere : terminé
Je suis le fils,mon PID : 66568
Je suis le fils,le PID de mon pere 66566
Je suis le fils  : terminé
```

Est-ce toujours le cas ? Si oui pourquoi ? Si non pourquoi ?

1.2 - Il arrive que le pid du père affiché par le fils soit égal à 1. Pourquoi d'après vous ?

1.3 - Comment résoudre ces "problèmes" ?

1.4 - Que vaut E dans chacun des processus juste après l'instruction `E=E+4`.

1.5 - On désire que le père crée 6 fils. Voila le programme proposé :

```
(...)
for(int n = 0; n<6, n++) {
 if (( i = fork()) == 0) {
   printf(" Je suis le fils, mon PID : %d \n",getpid());
   printf(" Je suis le fils,le PID de mon pere %d \n",getppid());
   printf(" Je suis le fils, j'ai fini \n");
 } else {
   printf(" Je suis le pere, mon PID  : %d d\n",getpid());
   printf(" Je suis le père, le PID du fils créé vaut : %d \n",i);
   printf(" Je suis le pere, j'ai fini \n");
 }
}
(...)
```

Ce code est-il correct ? Si oui pourquoi ? Si non pourquoi et corrigez-le ?

1.6 - Modifier ce programme pour que le père crée un fils qui créra lui-même un fils après 2 secondes ... Et ce jusqu'à N processus descendants en tout. N est donné en paramètre au lancement du père : `./a.out 6` par exemple.

2 Wait

L'appel système `wait(int *status)` permet à un père d'attendre de façon *passive* la terminaison d'un processus fils. Cette attente bloque le processus jusqu'à ce qu'un fils se termine (voir `man 2 wait`). La variable `status` permet d'obtenir des informations sur la terminaison du processus fils, et notamment sa valeur de retour.

— `WIFEXITED(status)` : renvoie vrai si le fils c'est terminé normalement.

— `WEXITSTATUS(status)` : si WIFEXITED est vrai, donne la valeur de retour du fils (sur 8 bits).

L'appel système `waitpid(pid_t pid, int* status, int option)` permet à un père d'attendre la terminaison du processus fils spécifié par `pid`. L'`option = WNOHANG` permet de ne pas bloquer le processus père : *attente passive* .

2.1 - Reprendre le programme 1.5. Le père devra attendre que le fils soit terminé avant de créer le suivant. Chaque fils attendra 3 secondes avant de commencer. Le père doit afficher les différentes

informations sur la fin de son fils.

2.2 - Modifier ce programme :
- Chaque fils attend entre 0 et 9 secondes avant de commencer (Utiliser `rand()%10`).
- Le père attend la fin d'un fils paire avant de créer le suivant (un fils impaire). Il n'attend pas la fin des fils impairs pour continuer

2.3 - Modifier le programme 2.2 pour que le père ne meurt que quand il est sûr que tous les processus fils sont morts

3 Serveur de calcul

Utilisez tout le savoir accumulé pour produire la version V1a du serveur de calcul.

4 Exec

L'appel système `execv(const char *path, char *const argv[]);)` permet de demander au système de remplacer le code du processus appelant par le code exécutable donnée par `path` et de relancer ce code avec les paramètres donnés par `command`. Par exemple, après que le shell se soit dupliqué (étape 3 de l'exécution d'une commande), le shell fils doit exécuter "/bin/ls -il". Ce processus fils remplace son code par celui contenu dans le fichier de la commande grace à l'appel système `execv` Le fichier `shell.c` sur Moodle contient le code correspondant :

```
const char *path = "/bin/ls"
char * myCommand[3]
myCommand[0] = "ls";
myCommand[1] = "-il";
myCommand[2] = NULL;
execv(path, myCommand);
```

remplace le code du processus par le code contenu dans le fichier `/bin/ls` puis relance le processus avec les paramètres contenus dans `myCommand`.

Dans la suite, on désire créer un "shell" ...

4.1 - Modifier ce code afin que l'utilisateur puisse passer en paramètre la commande à exécuter. Par exemple `./a.out /bin/ls -l` doit réaliser cette commande "ls -l"

4.2 - Idem mais l'utilisateur peut ajouter '&' à la fin de la commande (pas d'attente de la fin de l'exécution). Par exemple `./a.out /bin/ls -l \&` doit être équivalent à `ls -l &`.
Pourquoi faut-il mettre `\&` et non simplement `&` ?

4.3 - Modifier le programme pour que le shell effectue une boucle infinie consistant à lire le clavier et à réaliser les commandes entrées par l'utilisateur (Pour simplifier on supposera que les fichiers exécutables des commandes existent toujours)

5 Serveur de calcul

Utilisez tout le savoir supplémentaire accumulé pour produire la version V2a du serveur de calcul.

6 System

L'appel système `int system(const char *command)` permet de demander au système de lancer un nouveau processus réalisant la commande donnée par la chaîne de caractères `command`. Exemple :

```
char * myCommand = "/bin/ls -l /etc/passwd";
system(myCommand);
```

6.1 Comment cet appel système fonctionne-t-il concrètement ?

7.2 - *(Facultatif)* Modifier le programme shell précédent pour remplacer les `execv` par des `system`

Rappel : `char *strcat(char *s1, const char *s2);` s2 est recopié à la fin de s1
Il faut avoir déclarer chaîne de caractères initiale s1 par exemple par : `char s1[100];`