

SY5 – Systèmes d'exploitation

TD n° 7 : création de processus

(par souci de concision, les erreurs ne sont pas gérées ; on suppose que les appels système réussissent)

Exercice 1 :

Déterminer la généalogie de processus créée par chacun des bouts de code suivants :

| | | |
|---|---|--|
| <pre>for (i=0; i<3; i++) { fork(); }</pre> | <pre>if (fork() == 0) { fork(); }</pre> | <pre>if (fork()) { if (fork()) fork(); }</pre> |
|---|---|--|

Exercice 2 : fork et wait

On considère le (morceau de) programme suivant :

```

1 switch (fork()) {
2     case 0 :
3         write(STDOUT_FILENO, "a", 1);
4         exit(0);
5     default :
6         write(STDOUT_FILENO, "b", 1);
7         switch (r = fork()) {
8             case 0 :
9                 write(STDOUT_FILENO, "c", 1);
10                exit(0);
11            default :
12                // waitpid(r, NULL, 0);
13                // wait(NULL);
14                write(STDOUT_FILENO, "d", 1);
15                exit(0);
16        }
17 }
```

1. Faire un schéma représentant le déroulement des clonages et affichages.
2. Quelles sont toutes les sorties possibles ?
3. On décommente la ligne 12 (qui a pour effet de bloquer le processus appelant jusqu'à la terminaison de son fils de pid r) ; quelles sont alors les sorties possibles ?
4. On recommente la ligne 12, et on décommente la ligne 13 (qui a pour effet de bloquer le processus appelant jusqu'à ce qu'un quelconque de ses enfants termine) ; quelles sont alors les sorties possibles ?

(si le processus appelant n'a pas de fils, `wait` et `waitpid` retournent -1 immédiatement)

Exercice 3 : contrôler la généalogie

1. Écrire un programme où le processus père crée k fils, où k est donné en argument.
2. Que faut-il ajouter si on veut que le père attende la fin des k fils avant de se terminer ?
3. Écrire un programme qui crée un unique fils, qui lui-même crée un unique fils et cela jusqu'à une hiérarchie de hauteur k .

Faites en sorte qu'un processus parent ne termine jamais avant son processus fils.

Exercice 4 : clonage et héritage

Qu'affiche le programme suivant ?

```

int i, res=0;
for (i=0; i<10; i++) {
    if (fork() == 0) { printf("%d : %d\n", i, res); res += 1; exit(0); }
}
printf("%d : %d\n", i, res);
```

Exercice 5 : fork et stdout

On considère le (morceau de) programme suivant :

```

1  printf("a");
2  switch (fork()) {
3      case 0 : printf("b"); exit(0);
4      default : printf("c"); exit(0);
5  }
```

1. Quelles sont toutes les sorties possibles ?
2. Et si l'appel à `exit(0)` à la ligne 4 est remplacé par `_exit(0)` ?
3. Et si on ajoute à la fin de la ligne 1 `printf("\n");` ?

Rappel : si *cmd* est le nom d'une commande (i.e. une référence valide d'un fichier exécutable, ou le nom d'un fichier exécutable appartenant à l'un des répertoires du *PATH*) et *argv* un tableau de chaînes de caractères terminant par *NULL*, un appel à `execvp(cmd, argv)` recouvre le code du processus appelant par celui de *cmd*, réinitialise sa pile et son tas, réinitialise les registres du processeur, et appelle la fonction *main* de *cmd* avec les arguments définis par *argv*; un appel à `execlp(cmd, arg0, ... , argn, NULL)`, est équivalent à l'appel `execvp(cmd, argv)` où *argv* = {*arg0*, ... , *argn*, *NULL*}.

Exercice 6 : fork-bombe

Que pensez-vous des deux programmes suivants ?

```

int main() {
    while (1) {
        fork();
        sleep(2);
        write(STDOUT_FILENO, "*", 1);
    }
}
```

```

int main(int argc, char* argv[]) {
    fork();
    sleep(2);
    write(STDOUT_FILENO, "*", 1);
    execvp(argv[0], argv);
    exit(1);
}
```

Exercice 7 : fork, exec, wait et dup

1. Écrire un programme `execute` qui prend en argument une commande et ses éventuels arguments, l'exécute, puis affiche un message indiquant que l'exécution est terminée.

Par exemple :

```

$ ./execute gzip -v gros-fichier
gros-fichier: 30.3% -- replaced with gros_fichier.gz
... exécution terminée
```

2. Simuler la commande « `sleep 5 ; ps aux` », c'est-à-dire écrire un programme qui exécute d'abord la commande « `sleep 5` », puis, une fois celle-ci terminée, la commande « `ps aux` ».
3. Simuler la commande « `ls fic && echo "fichier trouvé"` », i.e. écrire un programme qui prend en paramètre un nom de fichier *fic*, et exécute d'abord la commande « `ls fic` » puis, si elle s'est bien passée, exécute la commande « `echo "fichier trouvé"` ».
4. On souhaite utiliser les erreurs de la commande « `ls -R rep` » pour déterminer quels sont les sous-répertoires inaccessibles dans l'arborescence de racine *rep*. Pour cela, on peut rediriger sa sortie standard vers le pseudo-fichier `/dev/null` – autrement dit, simuler la ligne de commande « `ls -R rep > /dev/null` ». Écrire le programme correspondant.