SY5 – Systèmes d'exploitation
TD n° 6 : 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 :

```
for (i=0; i<3; i++) {
  fork();
}
```

```
if (fork() == 0) {
  fork();
}
```

```
if (fork()) {
  if (fork()) fork();
}
```

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 hauteur de hiérarchie de k.

Exercice 4 : clonage et héritage

Qu'affiche le programme suivant ?

```
int 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);
```

Rappel *(ou pas, selon la date de votre TD...)* : 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`.

Exercice 5 : `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);
}
```

À votre avis, quel est le rôle des appels à `sleep()` ?

Exercice 6 : `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. 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.