Manipulation de processus en langage C

1 La primitive fork()

```
#include <unistd.h>
int fork();
```

L'appel à `fork()` duplique le processus. L'exécution continue *dans les deux processus* après l'appel à `fork()`. Tout se passe comme si les deux processus avaient appelé `fork()`. La seule différence (outre le PID et le PPID) est la valeur retournée par `fork()` :

- dans le processus *père* (celui qui l'avait appelé), `fork()` retourne le PID du processus fils créé;
- dans le processus fils, `fork()` retourne 0.

Notons que le fork peut échouer par manque de mémoire ou si l'utilisateur a déja créé trop de processus; dans ce cas, aucun fils n'est créé et `fork()` retourne -1.

Un exemple de programme appelant `fork()` est donné page pageref.

```
/*  Exemple utilisation primitive fork() sous UNIX
 *  Emmanuel Viennet, Juin 1995
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void main(void) {
    int pid; /* PID du processus fils */
    int i;

    pid = fork();
    switch (pid) {
    case -1:
        printf("Erreur: echec du fork()\n");
        exit(1);
        break;
    case 0:
        /* PROCESSUS FILS */
        printf("Processus fils : pid = %d\n", getpid() );
        exit(0); /* fin du processus fils */
        break;
    default:
        /* PROCESSUS PERE */
        printf("Ici le pere: le fils a un pid=%d\n", pid );
        wait(0); /* attente de la fin du fils */
        printf("Fin du pere.\n");
    }
}
```

2 Les primitives getpid() et getppid()

L'appel système `getpid()` retourne le PID du processus appelant. `getppid()` retourne le PID du père du processus.

3 La primitive exec()

La primitive `execlp()` permet le recouvrement d'un processus par un autre exécutable (voir 6.2).

```
int execlp( char *comm, char *arg, ..., NULL );
```

`comm` est une chaîne de caractères qui indique la commande à exécuter. `arg` spécifie les arguments de cette commande (argv).

Exemple : exécution de `ls -l /usr`

```
execlp( "ls",  "ls", "-l", "/usr", NULL );
```

Notons que la fonction execlp() retourne -1 en cas d'erreur. Si l'opération se passe normalement, `execlp()` *ne retourne jamais* puisque qu'elle détruit (remplace) le code du programme appelant.

4 La primitive exit()

`exit()` est une autre fonction qui ne retourne jamais, puisqu'elle termine le processus qui l'appelle.

```
#include <stdlib.h>
void exit(int status);
```

L'argument `status` est un entier qui permet d'indiquer au shell (ou au père de façon générale) qu'une erreur s'est produite. On le laisse à zéro pour indiquer une fin normale.

5 La primitive wait()

```
#include <sys/types.h>
#include <sys/wait.h>

// int wait( int *st );
```

L'appel `wait()` permet à un processus d'attendre la fin de l'un de ses fils. Si le processus n'a pas de fils ou si une erreur se produit, `wait()` retourne -1. Sinon, `wait()` bloque jusqu'à la fin de l'un des fils, et elle retourne son PID.

L'argument `st` doit être nul.

6 La primitive sleep()

```
#include <unistd.h>
int sleep( int seconds );
```

L'appel `sleep()` est similaire à la commande shell `sleep`. Le processus qui appelle `sleep` est bloqué durant le nombre de secondes spécifié, sauf s'il reçoit entre temps un signal.

Notons que l'effet de `sleep()` est très différent de celui de `wait()` : wait bloque jusqu'à ce qu'une condition précise soit vérifiée (la mort d'un fils), alors que sleep attend pendant une durée fixée. `sleep` ne doit jamais être utilisé pour tenter de synchroniser deux processus.