4 - PROCESSUS UNIX

0. RAPPELS

L'appel système `fork()` crée un processus fils qui diffère de son père uniquement par ses numéros de `pid` et de `ppid`. `fork()` renvoie 0 pour le fils et le `pid` du fils créé pour le père. Juste après le `fork()`, les deux processus disposent des mêmes valeurs de données et de pile. Mais il n'y a pas de partage : chaque processus a sa propre copie.

L'appel système `wait()` bloque un processus en attente de la fin de l'exécution d'un fils qu'il a créé. `wait()` renvoie le numéro du fils qui vient de se terminer. On peut passer à `wait()` un paramètre de type `int*` qui permet de récupérer des informations sur la terminaison du fils.

Les appels système de la famille `exec` lancent un programme exécutable. Si l'appel système est réussi, le processus faisant le `exec` se termine lors de la fin du processus qu'il a lancé : on ne revient pas d'un `exec` réussi. Si l'appel échoue (et dans ce cas seulement), les instructions qui suivent le `exec` sont exécutées.

1. EXECUTION D'UN FORK SIMPLE

Soit le programme C suivant :

```
main(){
   int pid;
   printf("debut\n");
   pid = fork();
   if (pid == 0) {
       printf("execution 1\n"); }
   else {
       printf("execution 2\n"); }
   printf("Fin\n");
   return EXIT_SUCCESS;
}
```

1.1.

Donnez les affichages effectués par le processus père et par le processus fils.

Processus père :	Processus fils :
`debut`	`execution 1`
`execution 2`	`fin`
`fin`	

2. EXECUTION D'OPERATIONS FORK IMBRIQUEES

Soit le programme suivant :

```
1:   int main(int argc, char *argv[]) {
2:
3:        int a, e;
4:
5:        a = 10;
6:        if (fork() == 0) {
7:             a = a *2 ;
8:             if (fork() == 0) {
9:                  a = a +1;
10:                 exit(2);
11:            }
12:            printf(" %d \n", a);
13:            exit(1);
14:       }
15:       wait(&e);
16:       printf("a : %d ; e : %d \n", a, WEXITSTATUS(e));
17:       return(0);
18:  }
```

2.1.

Donnez le nombre de processus créés, ainsi que les affichages effectués par chaque processus.

Le programme crée 1 processus fils (ligne 6) qui crée à son tour un processus petit-fils(ligne 8). Il y a donc en tout 3 processus (en comptant le père).

Le père affiche les valeurs de a et e.

a a été initialisée à 10 et le père ne l'a pas modifié par la suite. **Affichage de a : 20**.

e contient des informations sur la terminaison du fils. En particulier, la valeur de l'exit effectué par le fils peut-être obtenue à l'aide de la macro WEXITSTATUS appliquée à e (d'autres macros permettent l'accès à d'autres informations sur la terminaison, cf. `man`). **Affichage de WEXITSTATUS(e) : 1**.

Lors de la création du fils, a vaut 10. Le processus créé fait passer la valeur à 20 avant d'effectuer l'affichage. **Affichage : 20**

Le petit-fils ne fait pas d'affichage.

2.2.

On supprime la ligne 10, reprenez la question 2.1. en conséquence.

Les exécutions du père et du fils ne sont pas modifiées, par conséquent leurs affichages non plus. Dans ce nouveau programme, le petit-fils exécute maintenant la ligne 12. Lorsqu'il est créé, la variable a vaut 20, elle est ensuite incrémentée de 1. **Affichage : 21**

2.3.

Modifiez le programme initial pour créer un processus zombie pendant 30 secondes.

Un zombie est un processus qui s'est terminé, mais dont le père n'a pas encore été averti de la terminaison. La terminaison est prise en compte par l'exécution du wait.

Si après la création du fils, on empêche le père de s'exécuter pendant 30 s, le fils une fois terminé restera zombie jusqu'à la reprise du père. On modifie le programme de la manière suivante :

```
...
14:           }
14 bis :      sleep(30);
15:           wait(&e);
16:           printf("a : %d ; e : %d \n", a, WEXITSTATUS(e));
```

3. CREATION DE PROCESSUS EN CHAINE

On souhaite faire un programme qui crée une chaîne de processus telle que le processus initial (celui du main) crée un processus qui à son tour crée un second processus et ainsi de suite jusqu'à la création de N processus.

3.1.

Ecrivez ce programme.

```
void creerChaine (int nb) {
      int p, i;
      i = 0;
      p = 0;
      while (i < nb && (p = fork()) == 0) {
            i++;
      }
      if (p == -1) {
            perror("fork");
            exit(1);
      }
}

int main(int argc,  char *argv[]) {
      creerChaine (N);
      return EXIT_SUCCESS;
}
```

3.2.

Modifiez le programme pour que le processus initial attende uniquement la fin de son fils.

La fonction modifiée est bien sûr la fonction creerChaine.

```
void creerChaine (int nb) {
      int p, i;
      i = 0;
      p = 0;
      while (i < nb && (p = fork()) == 0) {
            i++;
      }
      if (p == -1) {
            perror("fork");
            exit(1);
      }
      if (i == 0) {
            wait(NULL);
      }
}
```

3.3.

Modifiez le programme pour que le processus initial attende la fin de *tous* les processus créés.

Un processus ne peut attendre la fin que de ses fils et pas des descendants plus éloignés. Il faut donc ici que chaque processus créé (sauf le dernier qui n'a pas de descendance) attende la fin de son fils pour garantir que le processus initial se termine après tous les processus créés.

```
void creerChaine (int nb) {
      int p, i;
      i = 0;
      p = 0;
      while (i < nb && (p = fork()) == 0) {
            i++;
      }

      if (p == -1) {
            perror("fork");
            exit(1);
      }

      if (i < nb) {
            wait(NULL);
      }
}
```

4. MODIFICATION DU CODE EXECUTE : LA PRIMITIVE EXEC

On considère le programme suivant :

```
int main() {
    int p;
    p = fork();
    if (p == 0) {
       execl("/bin/echo", "echo", "je", "suis", "le", "fils", NULL);
    }
    wait(NULL);
    printf("je suis le pere\n");
    return EXIT_SUCCESS;
}
```

4.1.

Donnez le résultat de l'exécution de ce programme (on suppose que l'appel `execl` est réussi).

Le processus fils fait un appel execl. Il lance la commande echo avec les paramètres (je suis le fils) : affichage de "je suis le fils" et terminaison.

Après la terminaison du fils, le père affiche "je suis le pere".

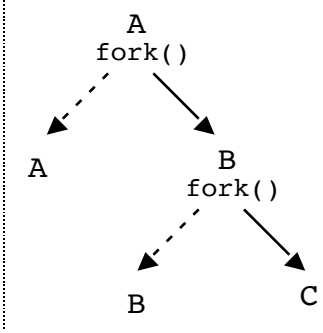
Soit le programme "nemaxe" suivant en C sous Unix, où "prog" est un programme qui ne crée pas de processus.

```
main(int argc, char*argv[]) {
   int retour;
   printf("%s\n", argv[0]);                                    // A
   switch (fork()) {
       case -1:
          perror("fork1()"); exit(1);
       case 0:// B
          switch (fork()) {
              case -1:
                 perror("fork2()");
                 exit(1);
              case 0:                                          // C
                 printf("ICI\n");
                 if (execlp("prog","prog",0) == -1) {
                     perror("execlp"); exit(1);
                 }
                 break;
              default:
                 exit(0);
          }
       default:
          wait(&retour);
   }
   printf("fin\n");
}
```

4.2.

Représentez tous les processus créés par ce programme sous forme d'un arbre, en vous servant des lettres en commentaires.

Le processus initial (celui qui exécute A) crée un fils. Ce fils exécute B et crée à son tour un fils qui exécute C. On a donc l'arbre suivant :



On change maintenant le nom de "nemaxe" par "prog" (celui de l'appel à execlp).

4.3.

Représentez le début de l'arbre de processus.