TD1 : Processus UNIX

RS : Réseaux et Systèmes

Deuxième année

L'objectif de ce TD est d'approfondir les notions relatives aux processus et de les appliquer au cadre spécifique d'Unix.

★ **Exercice 1:** Création de processus avec l'appel système `fork`

La commande `fork()` crée un processus «fils» du processus appelant (le «père»), avec le même programme que ce dernier. La valeur renvoyée par `fork()` est :

— Au père : le numéro (PID) du processus fils.

— Au fils : 0 (il peut retrouver le PID de son père avec `getppid()`)

En cas d'échec (table des processus pleine), aucun processus n'est créé, et la valeur -1 est renvoyée. Dans la suite, nous supposerons que tous les appels réussissent sans problème, mais une application réelle devrait bien entendu traiter les cas d'erreurs.

▷ **Question 1:** Qu'affiche l'exécution du programme suivant :

```
1  int main() {
2      pid_t pid;
3      int x = 1;
4
5      pid = fork();
6      if (pid == 0) {
7          printf("Dans fils : x=%d\n", ++x);
8          exit(0);
9      }
10
11     printf("Dans père : x=%d\n", --x);
12     exit(0);
13 }
```

Réponse

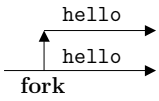
Dans fils : x=2 ; Dans père : x=0

Fin réponse

▷ **Question 2:** On considère les deux programmes suivants et leur schéma d'exécution.

- *Exemple :* un clonage

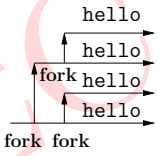
```
1 int main() {
2     fork();
3     printf("hello!\n");
4     exit(0);
5 }
```



- *Question 1 :* Illustrer l'exécution du programme suivant.

Réponse

```
1 int main() {
2     fork();
3     fork();
4     printf("hello!\n");
5     exit(0);
6 }
```

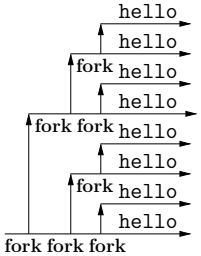


Fin réponse

- *Question 2 :* Même question pour le programme suivant.

Réponse

```
1 int main() {
2     fork();
3     fork();
4     fork();
5     printf("hello!\n");
6     exit(0);
7 }
```



Fin réponse

▷ **Question 3:** Combien de lignes «hello !» imprime chacun des programmes suivants ?

Programme 1 :

```
1 int main() {
2   int i;
3
4   for (i=0; i<2; i++)
5     fork();
6   printf("hello!\n");
7   exit(0);
8 }
```

Programme 2 :

```
1  void doit() {
2    fork();
3    fork();
4    printf("hello!\n");
5  }
6  int main() {
7    doit();
8    printf("hello!\n");
9    exit(0);
10 }
```

Programme 3 :

```
1 int main() {
2   if (fork())
3     fork();
4   printf("hello!\n");
5   exit(0);
6 }
```

Programme 4 :

```
1 int main() {
2   if (fork()==0) {
3     if (fork()) {
4       printf("hello!\n");
5     }
6   }
7 }
```

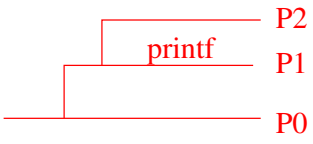
Réponse

Réponse 1 : 4 fois (c'est le second exemple de la question 2)

Réponse 2 : 8 fois. On crée 4 processus comme dans le premier cas, et chacun écrit deux fois «hello» (une fois dans doit(), une fois dans main())

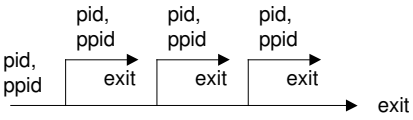
Réponse 3 : 3 fois. fork() renvoi non-nul (ie, `vrai`) dans le père seulement. Le père se clone donc à nouveau, tandis que les fils non.

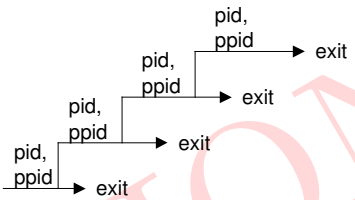
Réponse 4 : Une seule fois. Le père (P0) lance un fils (P1) et c'est tout (car fork ligne 2 renvoie != dans le père). Son fils P0 fait un fork supplémentaire ligne 3 qui crée P2. Ce fork renvoie == 0 dans P3 et != 0 dans P2. Donc, seul P2 entre dans le corps du if et exécute le printf.
En résumé, ca crée trois processus, mais ca n'affiche qu'une seule fois la ligne.



Fin réponse

▷ **Question 4:** On considère les deux structures de filiation (chaîne et arbre) représentées ci-après.





Écrire un programme qui réalise une chaîne de `n` processus, où `n` est passé en paramètre de l'exécution de la commande (par exemple, n = 3 sur la figure ci-dessus). Faire imprimer le numéro de chaque processus et celui de son père. Même question avec la structure en arbre.

Réponse

Il est plus simple de demander la chaine avant l'arbre.

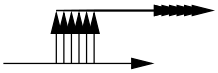
```
1  /*-*- coding: utf-8 -*-*/
2  #include <stdlib.h> /* exit, atoi */
3  #include <stdio.h>  /* printf */
4  #include <unistd.h> /* getpid, fork */
5  #include <wait.h>   /* wait */
6
7  void chaine(int n) {
8    int i;
9    printf("Lance %d processus en chaine\n\n",n);
10   printf("proc %d fils de %d (shell)\n",getpid(),getppid());
11
12   for (i=0; i<n; i++) {
13     if (fork() == 0) { /* fils */
14       printf("proc %d fils de %d\n",getpid(),getppid());
15       exit(0);
16     } else {
17       wait(NULL); /* RQ1: Si on le met pas, on a pas le bon schéma */
18     }
19   }
20 }
21
22 void arbre(int n) {
23   int i;
24   printf("Lance %d processus en arbre\n\n",n);
25   printf("proc %d fils de %d (shell)\n",getpid(),getppid());
26   for (i=0; i<n; i++) {
27     if (fork() == 0) { /* fils */
28       printf("proc %d fils de %d\n",getpid(),getppid());
```

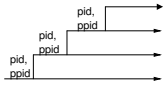
```
    } else {
      /* wait(NULL); RQ2: Si on le met, on a pas le bon schéma */
      exit(0); /* RQ3: Si on l'enleve, on a pas le bon schéma */
    }
  }
}

/* la fonction précédente est assez simplement récursivable; le résultat est très élégant, non ? */
/* Il n'est pas indispensable de présenter cette solution, mais ça fait pas de mal non plus */
void arbreRec (int n) {
   printf ("proc %d fils de %d \n", getpid(), getppid());
   if (n >0)
     if (fork() == 0 )
       arbre (n-1);
   exit(0);
}


int main(int argc, char *argv[ ]) {
  if (argc<2) {
    printf("Usage:\n%s 1 <n> pour lancer <n> processus en arbre\n%s 2 <n> pour lancer <n> processus en chaine\n",
           argv[0], argv[0]);
    exit(1);
  }
  switch(atoi(argv[1])) {
  case 1: arbre(atoi(argv[2]));  break;
  case 2: chaine(atoi(argv[2])); break;
  }
  return 0;
}
```

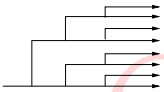
Y'a trois erreurs classiques, aux trois lignes marquées `RQ1`, `RQ2` et `RQ3`.

— `RQ1` : Si on ne met pas ce wait, on change le schéma d'exécution : 
wait est donc une primitive de syncronisation (en plus de nettoyer les zombies).

— `RQ2` : Si on ajoute un wait ici, on change le schéma : 

Les pères attendent leur fils respectif.
La question suivante est de savoir ce qui se passe si on ne met pas ce wait. Que deviennent les zombies dont le père est mort ? ? La réponse tient en deux éléments :
- Tout processus dont le père est mort est adopté par le processus 1 (init). Donc getppid=1 pour les orphelins.
- Init est surtout utilisé au lancement de la machine (il est chargé de lancer tous les processus nécessaires dans l'espace utilisateur, comme par exemple `login`) et à l'arrêt de la machine (c'est lui qui fait du propre parmis les processus utilisateurs). Entre temps, il passe son temps dans une boucle `while(1) { wait(NULL) }`. Il attend donc la fin des processus orphelins.

— `RQ3` : Si on enlève cet exit, on fait beaucoup trop de fils (car chaque père recree des fils à chaque tours de boucle). On fait donc 2^n processus au lieu de n.



Fin réponse

★ **Exercice 2:** Relations entre processus père et fils

▷ **Question 1:** Le programme ci-après lance un processus qui attend la fin de ses fils et imprime leur code de retour.

```
#define N 2

int main() {
    int status, i;
    pid_t pid;

    for (i = 0; i < N; i++)
        if ((pid = fork()) == 0)  /* fils */
            exit(100+i);

    /* le père attend la fin de ses fils */
    while ((pid = waitpid(-1, &status, 0)) > 0) {
        if (WIFEXITED(status))
            if (WEXITSTATUS(status) == 0) {
               printf("Le fils %d s'est terminé normalement avec le code 0.\n",pid);
               printf("Il n'a donc pas rencontré de problème.\n");
            } else {
               printf("Le fils %d s'est terminé avec le code %d.\n",pid,WEXITSTATUS(status));
               printf("Il s'agit en général d'un code d'erreur (cf. la page man du processus déclenché)\n");
            }
        else
            printf("Le fils %d s'est terminé à cause d'un signal\n", pid);
```

```
    }
    if (errno != ECHILD)
        perror("erreur dans waitpid");

    exit(0);
}
```

Modifier ce programme pour qu'il traite les fils dans l'ordre dans lequel ils ont été créés (en affichant leur code de retour).

Réponse

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <wait.h>

#define N 2

int main() {
    int status, i;
    pid_t pid[N+1],retpid;                                  /* MODIF */

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)  /* fils */             /* MODIF */
            exit(100+i);

    /* le père attend la fin de ses fils */
    i = 0;                                                  /* MODIF */
    while ((retpid = waitpid(pid[i++], &status, 0)) > 0) {  /* MODIF */
        if (WIFEXITED(status))
            printf("Le fils %d s'est terminé normalement avec le code %d\n", retpid, WEXITSTATUS(status));
        else
            printf("Le fils %d s'est terminé anormalement\n", retpid);
    }
    if (errno != ECHILD)
        perror("erreur dans waitpid");

    exit(0);
}

```

C'est le bon moment pour parler du traitement d'erreur autour des appels systèmes (on en a pas assez parlé en cours). La plupart des appels systèmes renvoient un résultat positif si tout s'est bien passé, et -1 en cas de problème. Ensuite, pour trouver le type d'erreur, il faut consulter la valeur de la variable `errno` (définie dans `#include <errno.h>`). Cette variable peut prendre diverses valeurs, et il faut lire la page de man de l'appel système pour voir les différentes causes potentielles d'erreur.

Par exemple, waitpid peut échouer pour trois raisons potentielles :

- ECHILD : on attend un process qui n'est pas l'un de ses fils.
- EINVAL : (erreur "invalide") l'un des arguments est invalide (&status =?= NULL)
- EINTR : (erreur "interrompue") On a pas pu attendre jusqu'à la mort d'un fils parce qu'on a reçu un signal entre temps.

Ensuite, on peut faire un bel affichage avec la fonction `perror`, qui va afficher qqch comme `erreur dans waitpid: pas de tel processus`.

Fin réponse

▷ **Question 2:** On considère le programme suivant :

```
int main() {
    int status;
    pid_t pid;

    fprintf(stderr,"Hello\n");
    pid = fork();
    fprintf(stderr,"%d\n", !pid);
    if (pid != 0) {
        if (waitpid(-1, &status, 0) > 0) {
            if (WIFEXITED(status) != 0)
                fprintf(stderr,"%d\n", WEXITSTATUS(status));
        }
    }
    fprintf(stderr,"Bye\n");
    exit(2);
}
```

Combien de lignes ce programme imprime-t-il ? Discuter les ordres possibles dans lesquels ces lignes sont imprimées.

Réponse

(on écrit sur stderr pour éviter les problèmes de bufferisation).

Le père affiche «hello» puis crée son fils. Ensuite, le père affiche tour à tour : « 0 », « 2 », « Bye ». Le fils affiche : «1», «Bye». En ce qui concerne l'ordre (la synchronisation), on sait que hello arrive premier, et que le «2» du père arrive après l'exit(2) du fils (et donc après le «Bye» du fils). Le reste peut être entrelacé. Ce qui nous donne :

— H 1 B 0 2 B
— H 1 0 B 2 B
— H 0 1 B 2 B

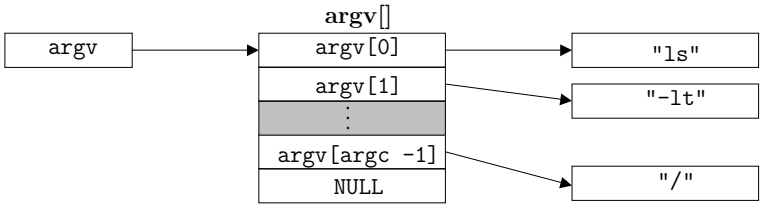
Fin réponse

★ **Exercice 3:** Exécution d'un programme

La famille de primitives `exec` permet de créer un processus pour exécuter un programme déterminé (qui a auparavant été placé dans un fichier, sous forme binaire exécutable). On utilise en particulier `execvp` pour exécuter un programme en lui passant un tableau d'arguments. Le paramètre `filename` pointe vers le nom (absolu ou relatif) du fichier exécutable, `argv` vers le tableau contenant les arguments (terminé par `NULL`) qui sera passé à la fonction `main` du programme lancé. Par convention, le paramètre `argv[0]` contient le nom du fichier exécutable, les arguments suivants étant les paramètres successifs de la commande.

```
1 #include <unistd.h>
2 int execvp(char *filename, char *argv[]);
```

Noter que les primitives `exec` provoquent le «recouvrement» de la mémoire du processus appelant par le nouveau fichier exécutable. Il n'y a donc pas normalement de retour (sauf en cas d'erreur – fichier inconnu ou permission – auquel cas la primitive renvoie -1).



▷ **Question 1:** Que fait le programme suivant ?

```
1 #include<stdio.h>
2 #include<unistd.h>
3 #define MAX 5
4 int main() {
5    char *argv[MAX];
6    argv[0] = "ls"; argv[1] = "-lR"; argv[2] = "/"; argv[3] = NULL;
7    execvp("ls", argv);
8 }
```

Réponse

il exécute la commande shell "ls -lR /" (ls récursif, avec affichage long)

Fin réponse

▷ **Question 2:** Écrire un programme `doit` qui exécute une commande Unix qu'on lui passe en paramètre. *Exemple :*

```
1    doit ls -lt /
```

Réponse

Le plus difficile dans l'histoire est de préparer un vecteur d'arguments prêt à passer à execvp. La première idée est de se lancer dans une copie complète : On malloc un tableau char** assez grand (ie, argc-1), et on recopie chaque élément de argv dans new_argv (on strdup les éléments pour copier chaque chaîne, même).

Mais c'est un peu inutile, car un exec() réalise un recouvrement de la mémoire : les zones mémoires que l'on dupplique (ie argv) avant de les utiliser vont être écrasées et perdues dans l'opération. Donc, pas besoin de recopier argv, y'a qu'à l'utiliser en place.

En utilisant argv en place

```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h> /* exit */

int main(int argc,char *argv[]) {
   if (argc < 2) {
      printf("Necessite au moins un argument\n");
      exit(1);
   }
   execvp(argv[1],argv+1);
   perror("execvp a échoué\n");
   return 1;
}

```

Fin réponse