Université Joseph Fourier, Grenoble
Licence d'Informatique L3

Introduction aux Systèmes et Réseaux

TD n°1 : Introduction aux processus Unix

L'objectif de ce TD [1] est d'approfondir les notions relatives aux processus et de les appliquer au cadre spécifique d'Unix. Ces notions seront appliquées dans le TP n°1. Ce texte ne donne pas tous les détails sur les primitives décrites. En cas de besoin, pour les TP, utiliser `man`.

1 Création de processus

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.

En cas d'échec (table des processus pleine), aucun processus n'est créé, et la valeur -1 est renvoyée au processus appelant.

Dans la suite, pour simplifier le traitement des erreurs, on pourra utiliser une fonction `Fork()` (avec une majuscule), dont le programme est donné ci-après :

```
pid_t Fork(void)
{
   pid_t pid;

   if ((pid = fork()) < 0)
      unix_error("Fork error");
   return pid;
}
```

La fonction `unix_error` affiche un message d'erreur et termine l'exécution du programme. De même pour les autres primitives agissant sur les processus et les fichiers (`Wait`, `Open`, etc). Pour les TP, on devra inclure la déclaration de ces fonctions (`csapp.h`), dont le programme sera fourni (`csapp.c`).

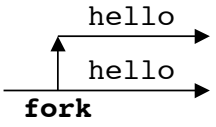
Exercice 1 Qu'affiche l'exécution du programme suivant (`fork_simple.c`) :

1. Plusieurs figures et exemples sont empruntés à R. E. Bryant, D. O'Hallaron. *Computer Systems : a Programmer's Perspective*, Prentice Hall, 2003.

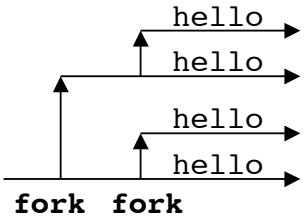
```c
#include "csapp.h"

int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

Exercice 2 On considère les deux programmes suivants et le schéma de leur exécution (l'axe du temps est orienté vers la droite). Illustrer l'exécution du programme obtenu en ajoutant un troisième `Fork()`.

```c
#include "csapp.h"
int main()
{
   Fork();
   printf("hello\n");
   exit(0);
}
```




```c
#include "csapp.h"
int main()
{
   Fork();
   Fork();
   printf("hello\n");
   exit(0);
}
```

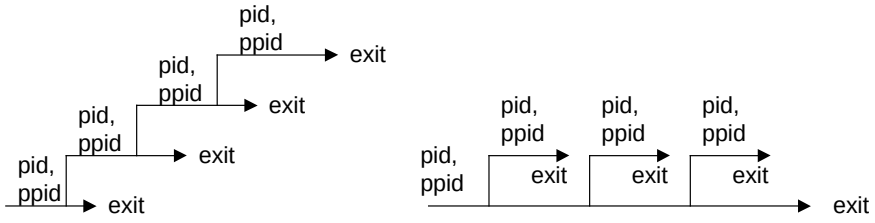


Exercice 3 Combien de lignes "hello" imprime chacun des deux programmes suivants ?

```
#include "csapp.h"
int main()
{
    int i;

    for (i = 0; i < 2; i++)
        Fork();
    printf("hello\n");
    exit(0);
}
```

```
#include "csapp.h"
void doit()
{
    Fork();
    Fork();
    printf("hello\n");
    return;
}
int main()
{
    doit();
    printf("hello\n");
    exit(0);
}
```

Exercice 4 On considère les deux structures de filiation (arbre et chaîne) représentées ci-après. Écrire un programme qui réalise un arbre de `n` processus (cf. figure de gauche), où `n` est passé en paramètre de l'exécution de la commande (par exemple, n = 4 sur la figure ci-dessus). Faire imprimer le numéro de chaque processus et celui de son père. Même question avec la structure en chaîne (cf. figure de droite).



2 Relations entre processus père et fils

Un processus termine son exécution en appelant la primitive `exit(int statut)`. La valeur de `statut` est utilisée pour renvoyer un code de retour qui donne des informations sur la terminaison du processus. Habituellement, on utilise la valeur 0 pour une terminaison normale, et une valeur différente de 0 pour signaler une condition anormale (cette valeur indiquant la nature de l'anomalie selon une convention fixée).

Lorsqu'un processus se termine, il ne restitue pas toutes ses ressources (il continue à occuper une entrée dans la table des processus du système d'exploitation) et reste dans un état appelé "zombi", tant que son père n'a pas pris connaissance de son statut, par une primitive `wait` ou `waitpid`. Ces primitives sont utilisées par un processus père pour attendre la fin d'un ou plusieurs de ses fils [2].

Nous utilisons la primitive `waitpid`, définie comme suit :

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

2. Si un processus se termine sans avoir attendu la fin de ses fils, ceux-ci sont rattachés au processus de numéro `pid` = 1, qui finira par les éliminer.

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *ptrStatut, int options);
```

Le paramètre `pid` permet de sélectionner le processus dont on attend la fin [3].

— si `pid` > 0, attendre la fin du processus fils de numéro `pid`

— si `pid` $= -1$, attendre la fin d'un processus fils quelconque

La valeur renvoyée est -1 en cas d'erreur, et autrement le numéro du processus fils (zombi) effectivement pris en compte. Si l'erreur est causée par un paramètre `pid` incorrect, (c'est-à-dire quand il n'existe pas de fils correspondant à la demande), le code d'erreur `ECHILD` est assigné à la variable globale `errno`.

L'entier `ptrStatut` donne des informations sur l'état du processus terminé. Pour l'interpréter, il est préférable d'utiliser des fonctions prédéfinies, par exemple :

— `WIFEXITED(ptrStatut)` retourne vrai si le fils s'est terminé normalement (c'est-à-dire s'il ne s'est pas terminé en réaction à l'arrivée d'un signal [4]), faux sinon.

— `WEXITSTATUS(ptrStatut)` donne le statut de sortie (code de retour) du fils, *uniquement si la primitive précédente a renvoyé vrai.*

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

```
#include "csapp.h"
#define N 2

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

    for (i = 0; i < N; i++) {
        if ((pid = Fork()) == 0) {  /* child */
            exit(100+i);
        }
    }

    /* parent waits for all of its children to terminate */
    while ((pid = waitpid(-1, &status, 0)) > 0) {
        if (WIFEXITED(status)) {
            printf("child %ld terminated normally with exit status=%d\n",
                   (long)pid, WEXITSTATUS(status));
        } else {
            printf("child %ld terminated abnormally\n", (long)pid);
        }
    }
```

3. Il y a d'autres possibilités liées aux groupes de processus, non examinées ici.

4. Les *signaux* sont des mécanismes de communication entre processus. Ils seront détaillés ultérieurement.

```
    if (errno != ECHILD) {
        unix_error("waitpid error");
    }

    exit(0);
}
```

Exercice 5 Modifier ce programme pour qu'il imprime le code de retour des processus en respectant dans l'ordre dans lequel ils ont été créés.

Exercice 6 On considère le programme suivant. Combien de lignes ce programme imprime-t-il ? Discuter les ordres possibles pour l'impression de ces lignes.

```
#include "csapp.h"
int main()
{
    int status;
    pid_t pid;

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

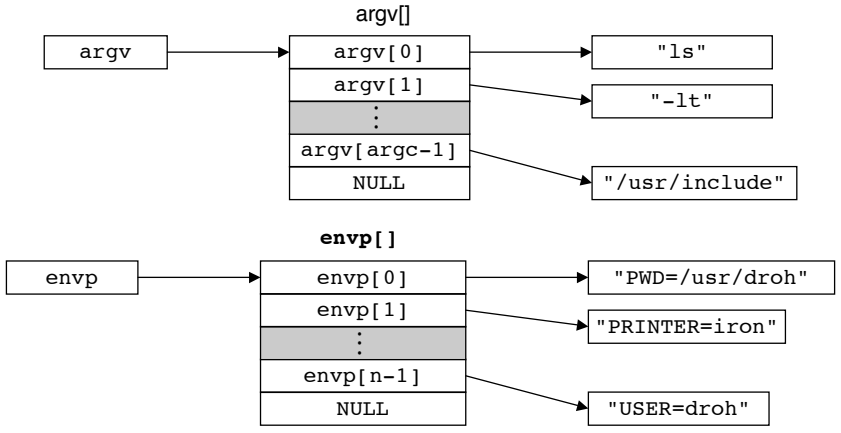
3 Exécution d'un programme

Rappelons d'abord les conditions d'exécution d'un fichier binaire (produit par exemple par compilation d'un programme C). La fonction exécutée a pour en-tête :

```
int main(int argc, char *argv[]);
```

où `argc` est le nombre d'arguments, et `argv` un tableau contenant ces arguments, la fin étant marquée par un élément de valeur `NULL`.

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 `execv` pour exécuter un programme en lui passant un tableau d'arguments, et `execve` en lui passant en outre un tableau de variables d'environnement (variables prédéfinies utilisées pour donner des informations telles que le nom de l'utilisateur, le *shell* préféré, les chemins de recherche des fichiers, etc.).

```
#include <unistd.h>
int execve(char *filename, char *argv[], char *envp[]);
```

La primitive `execv` ne comporte pas l'argument `envp[]`. Le paramètre `filename` pointe vers le nom (absolu ou relatif) du fichier exécutable, `argv` vers le tableau contenant les arguments (terminé par `NULL`), `envp` vers le tableau contenant les variables d'environnement (terminé par `NULL`). Par convention, le paramètre `argv[0]` contient le nom du fichier exécutable, les arguments suivants étant les paramètres successifs de la commande (par exemple la commande `ls` dans l'exemple ci-après).

Par ailleurs, il faut savoir qu'une variable prédéfinie appelée `environ` pointe, par convention, vers le premier emplacement des variables d'environnement dans la mémoire d'un processus (c'est-à-dire vers l'emplacement noté `envp[0]` sur la figure).

```
#include<stdio.h>
#include<unistd.h>
#define MAX 5
int main() {
   char *argv[MAX];
   argv[0] = "ls"; argv[1] = "-lt"; argv[2] = "/"; argv[3] = NULL;
   execv("/bin/ls", argv);}
```

Exercice 7 Que fait le programme ci-dessus ? Noter que les primitives `exec` provoquent le "recouvrement" de la mémoire virtuelle du processus appelant par le nouveau fichier exécutable. Il n'y a donc pas normalement de retour (sauf en cas d'erreur, par exemple fichier inconnu, auquel cas la primitive renvoie -1).

Exercice 8 Écrire un programme `execcmd` qui exécute une commande Unix qu'on lui passe en paramètre. Exemple d'invocation :

```
execcmd /bin/ls -Ft /
```

Exercice 9 Écrire un programme `manbis` qui exécute la commande `man` avec deux arguments qu'on lui passe en paramètres : le numéro de la section concerné et le sujet recherché. Exemple d'invocation : `manbis 3 printf`

On souhaite que la commande `man` soit exécutée avec certains réglages particuliers :

— La documentation doit être affichée en anglais. Pour cela, la variable d'environnement `LANG` doit être positionnée à la valeur `en_US` (la valeur pour une configuration française est `fr_FR`).

— L'affichage du manuel dans le terminal doit substituer après que l'on ait quitté `man`. Pour cela, la variable d'environnement `MANPAGER` doit être définie et être positionnée à la valeur `less -X`.

Question 10

Modifier le programme de la question 8 pour qu'il ne soit plus nécessaire de préciser le chemin de la commande à exécuter (en supposant que ce dernier soit inclus dans la variable d'environnement `PATH`).