

# Processus

## Thèmes abordés

- Protocole d'usage de l'API système.
- Primitives de gestion des processus : création, terminaison, recouvrement.
- Modèle et protocole d'usage associé à la création de processus Unix.
- Coordination père/fils : attente de terminaison, attente d'un délai.
- Boucle de base d'un interpréteur de commandes.

**Ressources** : pour ce TP, comme pour les suivants, vous pourrez vous appuyer sur

- Le polycopié intitulé « Systèmes d'exploitation : Unix », qui fournit une référence généralement suffisante sur la sémantique et la syntaxe d'appel des différentes primitives de l'API Unix. Chaque section du sujet de TP indique la (ou les) section(s) du polycopié correspondant au contenu présenté.
- Les pages du manuel en ligne (commande `man`), et plus particulièrement les sections 2 et 3.

## Cheminement et objectifs pour la séance :

- Les sections 1 à 3 doivent être traitées par tous. La durée de la séance de TP devrait (normalement) (à peu près) permettre de traiter ces sections.
- La section 4 présente un point utile pour la suite, mais non essentiel.
- La section 5 est un exercice qui permet d'amorcer la réalisation du minishell ; cet exercice devrait être simple à réaliser, une fois les sections 1 à 3 traitées.

## 1 Interface shell/applications Unix (rappel)

Lors de la compilation, `gcc` réalise l'édition de liens avec un binaire (`crt1.o`) contenant la fonction `start()`, qui est la première appelée à l'exécution du programme. Cette fonction est une enveloppe, dont le cœur est un appel à la fonction `main()`.

Cette fonction est définie de manière à ce que le programme puisse être lancé depuis le shell, sous la forme d'une commande avec des paramètres. En effet le prototype de la fonction `main` est :

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

- `argc` est le nombre de paramètres de la ligne de commande saisie (le nom de la commande/du programme compte pour 1 paramètre)
- `argv` est un tableau de pointeurs vers des chaînes de caractères. Chaque élément du tableau correspond à un argument de la (ligne de) commande. `argv[0]` est un pointeur sur le nom de la commande.  
`argv[argc]` est un pointeur nul, qui marque la fin de la liste d'arguments.

### Exemple

On suppose que le programme suivant a été compilé dans un fichier exécutable de nom `arguments`<sup>1 2</sup>

```
#include <stdio.h>    /* printf */
#include <stdlib.h>   /* EXIT_SUCCESS */

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

    printf("argc=%d\n", argc);
    for (i=0; i<argc; i++) {
        printf("argv[%d]=\"%s\"\n", i, argv[i]);
    }

    return EXIT_SUCCESS;
}
```

1. Le code source de ce programme est disponible avec le sujet, sous le nom `arguments.c`; il est assorti de variantes fournies en commentaire.

2. L'instruction `return` de la procédure `main()` fournit le paramètre (constante `EXIT_SUCCESS`) de l'appel à `exit()` effectué par le module de lancement qui enveloppe la procédure `main()`.

La saisie de la ligne de commande `arguments -option1 26 toto` produira l'affichage :

```
argc = 4
argv[0]="arguments"
argv[1]="-option1"
argv[2]="26"
argv[3]="toto"
```

## 2 Gestion des processus (polycopié API Unix, sections 2.2.1 à 2.2.5)

### 2.1 Création et identité des processus (`fork`, `getpid`, `getppid`), attente d'un délai (`sleep`)

On considère le programme suivant :

```
#include <stdio.h> /*printf */
#include <unistd.h> /*fork */
#include <stdlib.h> /*EXIT_SUCCESS */

int main () {
    fork(); printf ("fork_1\n");
    fork(); printf ("fork_2\n");
    fork(); printf ("fork_3\n");

    return EXIT_SUCCESS;
}
```

Pour chacune des questions suivantes, prédire l'effet attendu **avant** de lancer le programme.

- Quel est le nombre total de processus engendrés par le lancement de ce programme? Dessinez (sur papier) la hiérarchie de processus créés.
- Combien d'occurrences de chacun des messages `fork i` seront affichées par ce programme?
- Dans quel ordre s'afficheront ces messages?

Saisir, compiler et exécuter le programme ci-dessus, puis vérifier les réponses aux questions précédentes. Dans le cas où l'effet obtenu n'est pas l'effet attendu, proposer une explication.

On souhaite vérifier/expliciter les résultats précédents en reconstituant l'arbre des processus créés.

- Remplacer l'instruction `printf("fork i\n");` suivant chaque `fork()` par :  
`printf("fork i : processus %d, de père %d\n", getpid(), getppid());`  
et ajouter `sleep(180);` avant l'instruction `return EXIT_SUCCESS;`
- Lancer le programme et afficher dans un autre terminal la liste des processus actifs.  
*Remarque* : l'option `fj`<sup>3</sup> de la commande `ps` ("`ps fj`") permet d'afficher la hiérarchie des processus listés.  
Vérifier que l'arbre donné par `ps` est conforme à celui que vous avez obtenu en répondant à la première question.
- Quel est l'intérêt de l'instruction `sleep(180);`?
- Est-il possible d'éviter d'ouvrir un autre terminal pour lancer la commande `ps`? Si oui comment, sinon pourquoi?

---

3. l'option `-Hej` donne la hiérarchie, mais pas le `ppid`; en outre cela fait un peu meuble Ikea.

## 2.2 Terminaison (exit), état d'un processus (actif, zombie...)

On considère le programme suivant (disponible avec le sujet sous le nom `ez.c`) :

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

int main(int argc, char *argv[]) {
    int tempsPere, tempsFils;
    int v=5; /* utile pour la section 2.3 */
    pid_t pidFils;

    tempsPere=120;
    tempsFils=60;
    pidFils=fork();
    /* bonne pratique : tester systématiquement le retour des appels système */
    if (pidFils == -1) {
        printf("Erreur fork\n");
        exit(1);
        /* par convention, renvoyer une valeur > 0 en cas d'erreur,
        * différente pour chaque cause d'erreur
        */
    }
    if (pidFils == 0) { /* fils */
        printf("processus %d (fils), de père %d\n", getpid(), getppid());
        sleep(tempsFils);
        printf("fin du fils\n");
        exit(EXIT_SUCCESS); /* bonne pratique :
                                terminer les processus par un exit ex
        */
    }
    else { /* père */
        printf("processus %d (père), de père %d\n", getpid(), getppid());
        sleep(tempsPere);
        printf("fin du père\n");
    }
    return EXIT_SUCCESS; /* -> exit(EXIT_SUCCESS); pour le père */
}
```

Exécuter ce programme, et vérifier avec la commande `ps fj` l'état des processus correspondants (l'option `j` de la commande `ps` fournit l'état des processus : `S` (Sleeping, en attente d'événement), `R` (Running, actif), `T` (sToppé), `Z` (Zombie)...) :

- juste après le lancement,
- après le message de fin du fils : quel est l'état du fils ?
- après le message de fin du père.

Modifier ce programme en échangeant les valeurs initiales des variables `tempsPere` et `tempsFils` :

- juste après le lancement,
- après le message de fin du père : quel est l'état du fils ? À quel processus est-il rattaché ?
- après le message de fin du fils.
- pourquoi l'invite (prompt) est-elle affichée avant le message de fin du fils ?

## 2.3 Héritage des données (polycopié API Unix, section 2.2.10)

Modifier le code précédent,

- en complétant l'affichage initial de chacun des processus (`printf("processus %d ... , de père..", getpid()...);`) par l'affichage la valeur de la variable `v`
- puis en affectant une valeur différente à `v` dans chacun des processus (par exemple : 10 dans le père, 100 dans le fils)
- et enfin, en complétant l'affichage final de chacun des processus (`printf("fin du ...");`) par l'affichage la valeur de la variable `v`.

Exécuter et interpréter le résultat.

### 3 Coordination père/fils (polycopié API Unix, sections 2.2.6 à 2.2.9)

#### 3.1 wait (wait, macros WIFEXITED, WEXITSTATUS, WTERMSIG)

Le code suivant (disponible avec le sujet sous le nom `we.c`) suit le même schéma que celui de la section précédente, en remplaçant, au niveau du processus père, l'attente d'un délai (`sleep(.)`) par l'attente de son processus fils (`wait(.)`).

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h> /* wait */

int main(int argc, char *argv[]) {
    int tempsFils, codeTerm;
    pid_t pidFils, idFils;

    tempsFils=60;
    pidFils=fork();
    /* bonne pratique : tester systématiquement le retour des appels système */
    if (pidFils == -1) {
        printf("Erreur_fork\n");
        exit(1);
        /* par convention, renvoyer une valeur > 0 en cas d'erreur,
         * différente pour chaque cause d'erreur
         */
    }
    if (pidFils == 0) { /* fils */
        printf("processus_d_(fils),_de_père_d\n", getpid(), getppid());
        sleep(tempsFils);
        printf("fin_du_fils\n");
        exit(EXIT_SUCCESS);
    }
    else { /* père */
        printf("processus_d_(père),_de_père_d\n", getpid(), getppid());
        idFils=wait(&codeTerm);
        if (idFils == -1) {
            perror("wait_");
            exit(2);
        }
        if (WIFEXITED(codeTerm)) {
            printf("[%d]_fin_fils_d_>_par_>_exit_>_d\n", codeTerm, idFils, WEXITSTATUS(codeTerm));
        } else {
            printf("[%d]_fin_fils_d_>_par_>_signal_>_d\n", codeTerm, idFils, WTERMSIG(codeTerm));
        }
        printf("fin_du_père\n");
    }
    return EXIT_SUCCESS; /* -> exit(EXIT_SUCCESS); pour le père */
}
```

#### Questions

1. Quel sera l'affichage si on laisse le programme s'exécuter jusqu'à son terme? Vérifier votre réponse en exécutant le programme effectivement.
2. Quel sera l'affichage si on tue le fils depuis un autre terminal (`kill -9 pid_fils`)? Vérifier votre réponse en exécutant ce scénario. Essayer de terminer le fils par l'envoi de différents signaux.  
*Rappel* : la commande `kill -l` permet d'obtenir la liste des signaux disponibles.

#### 3.2 Recouvrement : les primitives `exec` (`execl`, `execlp`, `execvp`, `execve...`)

1. Ecrire un programme qui exécute la commande `ls -l <nom_fichier>`, et affiche un message indiquant si la commande a été correctement exécutée ou non.
  - Est-il utile d'afficher un message dans le cas où la commande a été correctement exécutée?
  - Testez ce programme avec un nom de fichier correct, puis avec un nom de fichier n'existant pas dans le dossier.
2. Essayez différentes variantes de commandes de la famille `exec`.

## 4 Environnement des processus Unix

Le shell gère un ensemble de variables d'environnement, qui permettent à l'utilisateur de configurer son environnement de travail : `PATH`, `TERM`, `SHELL`... L'accès aux variables d'environnement est permis via une variable globale externe (c'est-à-dire définie dans un autre module objet) `environ`, qui doit être déclarée dans le programme principal, avant la procédure `main` :

```
extern char **environ;
```

Cette variable correspond à un tableau de chaînes (terminé par un pointeur nul). Chaque chaîne du tableau est une ligne de la forme :

```
identifiant=valeur
```

où `identifiant` est l'identifiant d'une variable d'environnement et `valeur` sa valeur.

Les opérations suivantes, définies dans `<stdlib.h>` permettent d'accéder aux variables d'environnement :

- `char * getenv(const char *name)` : fournit la valeur de la variable de nom `name`
- `int putenv(char *string)` : ajoute un élément (ligne) au tableau des variables d'environnement
- `int setenv(const char *name, const char *value, int overwrite)` : construit et ajoute/remplace une ligne « `name=`value » du tableau des variables d'environnement
- `int unsetenv(const char *name)` supprime la ligne correspondant au nom `name`

### Exercices

1. Implémenter une commande `environnement`, listant le contenu du tableau des variables d'environnement. Son résultat est donc le même que celui obtenu par la commande `env`.
2. L'environnement peut aussi être transmis au programme principal sous la forme d'un troisième paramètre d'appel. Le prototype du programme principal est alors

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

Compléter le programme principal pour lister le contenu de la variable `environ` et celui du tableau `arge` transmis en troisième paramètre. Vérifier que les valeurs affichées sont identiques.

3. Ecrire un programme qui positionne une variable d'environnement (par exemple la variable `PATH`), puis crée un processus fils, lequel lance un exécutable externe qui affiche son environnement.

## 5 Exercice de synthèse : miniminishell

Ecrire un programme `miniminishel.c` qui répète les actions suivantes dans une **boucle** :

- demande à l'utilisateur d'entrer le nom d'une commande (sans paramètre) de moins de 30 caractères, et la lit :

```
char buf[30] ; /* contient la commande saisie au clavier */
int ret ; /* valeur de retour de scanf */
...
ret=scanf("%s", buf) ; /* lit et range dans buf la chaine entrée au clavier */
```

- puis lance la commande et indique si elle a été correctement exécutée ou non en affichant un message commençant (exactement) par `SUCCES` ou `ECHEC`.

L'exécution du programme ne sortira de cette boucle que lorsque l'entrée standard sera fermée.

- En mode interactif, la fermeture de l'entrée standard est provoquée par la frappe de la combinaison de touches `ctrl-D`
- Dans ce cas, la valeur de retour de `scanf("%s", buf)` est `EOF`, et non 1

Lorsque le programme sort de la boucle, il affiche un message commençant (exactement) par `Salut`, puis se termine.

**Question complémentaire** : compléter le programme précédent pour permettre de sortir de la boucle lorsque (exactement) la chaîne `exit` est saisie au clavier.

### Notes

- Le comportement des commandes lancées ne doit pas impacter l'exécution du programme `miniminishell`.
- Le message invitant l'utilisateur à saisir une nouvelle commande devra (exactement) commencer par les 3 caractères `>>>`