

# TP 1 – Commandes de base & Processus

Les exercices sont à faire dans l'ordre.

## 1 Sous Unix/Linux : les manipulations de base à connaître

### a. Le manuel : *man*

Le manuel est la documentation sous Unix accessible en ligne de commande : *man* . Il contient entre autres le détail des commandes et fonction sous Unix/Linux. Il est décomposé en 8 volumes :

Commande	type de documentation
<code>man 1 &lt;nom&gt;</code>	documentation des commandes (par défaut)
<code>man 2 &lt;nom&gt;</code>	documentation des appels système
<code>man 3 &lt;nom&gt;</code>	documentation des fonctions
...	...

**Question :** Comparez les 3 *man* pour chacune de ces 3 fonctions/commandes :

- `exit`
- `kill`
- `sleep`

À partir de maintenant (et pour le reste de votre vie ;-) ) si vous cherchez un renseignement sur le comportement, la valeur de retour, ou le code erreur d'une fonction/commande, ayez le réflexe *man* !

### b. Commandes Shell

Le Shell fera l'objet d'un chapitre entier. Voici un bref aperçu nécessaire pour les premiers TPs.

Une commande est toujours de la forme : `nom_de_commande arg1 arg2...`

**Questions (à l'aide des commandes indiquées dans le tableau ci-dessous) :**

- ▶ Dans une fenêtre Xterm, créez un répertoire **Systeme** en ligne de commande (`mkdir`). Placez-vous à l'intérieur (`cd` et vous pouvez aussi tester la touche *tabulation*).
- ▶ Créez un fichier vide : `touch essai.txt`. Éditez-le avec `emacs`. Modifiez-le puis enregistrez-le. Listez le contenu du répertoire (avec `ls`, `ls -l` et `ls -la`). Il y a maintenant 2 fichiers. Supprimez celui dont le nom termine par un tilde `~` (`rm`), il s'agit de la version du fichier qui précède son dernier enregistrement.
- ▶ Dans ce répertoire, créez un nouveau répertoire, puis déplacez le fichier `essai.txt` dans ce répertoire (`mv`). Supprimez le répertoire (testez avec `rmdir` pour voir ce qu'il se passe, puis `rm -r`).

## Quelques commandes de base à connaître :

Commande	Description	Exemple
<code>mkdir</code>	Créer un répertoire (vide)	<code>mkdir &lt;nom du répertoire&gt;</code>
<code>rm</code>	Détruire un fichier	<code>rm &lt;nom du fichier&gt;</code> (option <code>-i</code> : interactif)
<code>rmdir</code>	Détruire un répertoire	<code>rmdir &lt;rep&gt;</code> s'il est vide ; <code>rm -r &lt;rep&gt;</code> sinon
<code>pwd</code>	Chemin absolu du répertoire courant	(print working directory)
<code>ls</code>	Lister le contenu du répertoire courant	<code>ls</code> , <code>ls -l</code> , ou <code>ls -la</code> (plus complet),
<code>cd</code>	Changer de répertoire	<code>cd /rep1/rep2</code> (chemin absolu) <code>cd rep1/rep2</code> (chemin relatif), <code>cd</code> (seul = revient au home directory)
<code>cp</code>	Copier un fichier dans un répertoire	<code>cp fichier.txt rep1/rep2/</code>
<code>mv</code>	Déplacer un fichier dans un répertoire	<code>mv fichier.txt rep1/rep2/</code>
<i>touche tabulation</i>	Permet de compléter un mot en cours d'écriture par le nom d'un fichier existant. Très pratique pour éviter de tout taper ou de faire des erreurs de frappe	<code>cd nomde</code> puis <i>tabulation</i> donne <code>cd nomderepertoireexistant</code>



## Caractères spéciaux :

- le caractère `*` : désigne n'importe quelle chaîne de caractères.
- le caractère `?` : désigne n'importe quel caractère.

Exemple :

`rm *.o` : détruit tous les fichiers dont le nom finit par `.o`

`ls *.c` : donne la liste de tous les fichiers dont le nom finit par `.c`

### c. Compilation C

Rappel : un fichier C se compile avec : `gcc -Wall fichier.c -o fichier`

[`fichier.c` : la source, `fichier` : l'exécutable]

L'option `-Wall` permet d'afficher tous les "**warnings**". Lors des TPs de système nous éviterons (contrairement aux consignes en TP de C;-) ) d'utiliser l'option `-ANSI`. En effet, certaines fonctions systèmes ne sont pas ANSI et peuvent entraîner des erreurs de compilation. Mais même sans cette option, vous avez le droit de programmer correctement !

**Question :** Réalisez un programme qui affiche "Hello world! Bonjour xxx! Je suis le programme yyy.". *xxx* sera remplacé par votre prénom qui sera passé en argument de la ligne de commande et *yyy* est le nom du programme.

Exemple :

```
machine$ prog john
Hello world : Bonjour John ! Je suis le programme prog.
```

Partez du code du Hello World de base :

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

int main(int argc, char ** argv){
    printf("Hello World !\n");
    exit(0);
}
```

Notez que la fonction *main* est correctement écrite :

- paramètres : `int argc`, `char ** argv`
- type de retour : `int`
- `return(0)` ou `exit(0)` ou `exit(EXIT_SUCCESS)` lorsque le programme se termine normalement.
- `exit(-1)` ou `exit(EXIT_FAILURE)` pour indiquer que le programme se termine après une erreur identifiée (par exemple si le nombre d'argument n'est pas valable).

*Tout manquement à ces points, pour un.e informaticien.ne, est passible de sanctions qui peuvent dépasser l'imagination (paraît-il 😊)*

Vous pouvez maintenant réaliser la question demandée.



#### Aide en C

*argc* est le nombre d'arguments donnés en ligne de commande + le nom de l'exécutable et *argv* est un tableau de chaînes de caractères (*argv[0]* est le nom du programme, *argv[1]* est le premier argument...).

## d. Processus

Il sera parfois nécessaire de vérifier le fonctionnement (ou l'arrêt) d'un processus.

## Questions (à l'aide des commandes dans le tableau ci-dessous) :

- ▶ Tout d'abord réalisez un programme C qui affiche le `pid` du processus et celui de son père (avec les fonctions `getpid()` et `getppid()`) de la manière suivante :
  - affiche : "`xxx` (père : `yyy`) : Début de programme"
  - dort pendant 10 secondes (fonction `sleep(10)`)
  - affiche "`xxx` (père : `yyy`) : Fin de programme" avec `xxx` le `pid` du processus et `yyy` le `pid` du processus père (qui n'est autre que le shell ayant lancé le programme).
- ▶ Lancez ce programme en arrière-plan (avec `&`) et avec la commande `ps -l` trouvez la colonne qui donne l'information de l'état du processus.
- ▶ Dans ce programme à la place de `sleep`, réalisez un tableau d'entier d'1 million de cases. Initialisez-le à 0. Puis 10000 fois ajoutez +1 à chaque case du tableau (=on parcourt 10000 fois le tableau). Tous les 1000 parcours faites afficher le nombre de parcours déjà réalisés (utilisez l'opérateur *modulo* (%) qui donne le reste de la division : `i%1000` donnera donc 0 si `i` est un multiple de 1000). Observez l'état du système avec la commande `top` (dans une autre fenêtre). Combien de place mémoire le processus utilise, l'occupation CPU, ...
- ▶ Réalisez la manipulation classique suivante qui consiste à faire passer une application de l'avant-plan à l'arrière-plan (exemple : vous avez un programme qui est relativement long mais en cours de toute vous voulez lancer une autre commande en attendant). Lancez le programme (précédent) en avant-plan (c'est-à-dire donc sans `&` à la fin) et durant son exécution, suspendez-le (`CTRL+z`) puis tapez la commande "`bg`". Vérifiez avec (avec `ps` et `top`) que le programme est bien en train de fonctionner en trouvant la colonne qui indique si le programme est actif ou s'il est "endormi".
 

Indice : Dans cette colonne R signifie *Running* (actif), S signifie *Sleeping* (endormi), ...
- ▶ Maintenant relancez le programme en avant-plan et durant son exécution, suspendez-le (`CTRL+z`) puis observez le processus (avec `ps` et `top`) pour voir s'il est actif ou "endormi". [ne le passez pas en arrière-plan - et faire la question suivante]
- ▶ Tapez `jobs` pour connaître le numéro de la tâche et faites une reprise du programme en arrière-plan. Observez l'état du *prompt*, vous pouvez en effet taper une commande (`ls` par exemple) même si le programme en arrière-plan affiche des choses à l'écran.
- ▶ Lancez le programme et à l'aide d'une autre fenêtre, récupérez le numéro du processus (avec `ps` et des options) et tuez-le avec la commande `kill`.
- ▶ Dans votre programme, faites une erreur de syntaxe (mais si vous pouvez le faire :-), lancez l'enchaînement des commandes de compilation et du programme des deux manières suivantes :
  - `<compilation...> ; <prog et arguments...>`
  - `<compilation...> && <prog et arguments...>`

Quelle différence remarquez-vous ? Dans le premier cas, le programme s'est lancé alors que la compilation n'a pas fonctionné, c'est ça ? Effacez l'exécutable et recommencez.

Avec `&`, on attend le succès de la commande précédente... Or la compilation n'a pas réussi donc n'a pas renvoyé 0 donc la deuxième commande n'a pas été lancée (même si le fichier existait). Donc faites attention avec l'enchaînement des commandes et l'importance de faire terminer un programme par un succès.

### Commandes permettant de contrôler l'exécution d'un processus :

Commande	Description	Commentaire
<code>Ctrl-Z</code>	Mise en sommeil de la tâche courante en avant-plan	=suspension
<code>&lt;cmd&gt; &amp;</code>	Lancement en arrière-plan (dans un sous-shell)	Le prompt réapparaît avant la fin de la commande
<code>jobs</code>	Liste des tâches en cours dans le shell	
<code>fg [%&lt;n&gt;]</code>	Reprise ou mise en avant-plan (<n>=numéro de tâche bash)	Pour connaître le numéro de la tâche ( $\neq$ pid) tapez <code>jobs</code> . Ex : <code>fg %2</code> (par défaut : %1)
<code>bg [%&lt;n&gt;]</code>	Reprise en arrière-plan (<n>=numéro de tâche bash)	
<code>&lt;cmd1&gt; ; &lt;cmd2&gt;</code>	Lancement séquentiel des deux commandes	<code>&lt;cmd2&gt;</code> attend la fin de la <code>&lt;cmd1&gt;</code> avant de débiter
<code>&lt;cmd1&gt; &amp;&amp; &lt;cmd2&gt;</code>	Lancement séquentiel des deux commandes si la <code>&lt;cmd1&gt;</code> finit avec succès	
<code>Ctrl+C</code>	Interruption de l'exécution	(le processus est tué - si le programme le permet)
<code>kill -SIGKILL &lt;n&gt;</code>	Terminaison "brutale" du processus de numéro <n>	=le signal SIGKILL est envoyé au processus. <code>kill -9 &lt;n&gt;</code> est possible aussi.
<code>killall nom</code>	Terminaison de tous les processus de nom <code>nom</code>	
<code>ps</code>	Liste les informations sur les processus référencés dans le système	<code>ps -l -u votre_login</code> (processus vous appartenant dans le shell) <code>-e</code> : pour les processus sur le serveur, <code>-f</code> : pour plus de détails
<code>top</code>	Liste les informations d'exécution des processus en cours	(commande d'avant-plan)


## 2 Utilisation de *fork*

Prenez l'habitude de récupérer les exemples du cours pour ne pas perdre de temps et éviter de réinventer la roue ou de recopier des choses fausses provenant du web...

### a. Création simple

Écrire un programme C créant un processus père et un processus fils. On prendra soin de réaliser des affichages à l'écran pour savoir où l'on se situe lors de l'exécution. Vous afficherez les pid de chaque processus (*getpid()* et *getppid()*).

**Prenez l'exemple donné en cours et adaptez-le.**

 **aide C**

- ✎ Pour écrire les messages d'erreurs (par exemple lorsque *fork* renvoie -1), vous utiliserez la fonction *perror("...")* suivie d'un *exit(n)* où n est différent de 0.
- ✎ Pour les autres messages, vous utiliserez la fonction *printf("...")* ; suivie de *fflush(stdout)* ; pour s'assurer de l'écriture à l'écran.

[flush en anglais ? ... la chasse (des WC)... la but de cette fonction devient plus claire non 😊]

### b. Recouvrement avec *execl*

On veut appeler un programme avec 2 arguments. Ces 2 arguments seront transmis à un autre programme qui fera une opération dessus.

Ainsi en tapant **Pere 9 trucs** on doit obtenir "J'ai 18 trucs". L'affichage et la multiplication sera faite en réalité par un autre programme (dit recouvrant).

Écrire deux programmes C :

- ▶ Le premier est le processus père : Il crée le processus fils (avec *fork* comme le précédent). Le programme père prendra 2 arguments (1 entier et une chaîne de caractères). Ces arguments (contenus généralement dans *argv* dans la fonction *main*, voir l'encart ci-dessous) sont des chaînes de caractères (même les entiers). Mais comme le processus père ne manipulera pas ces arguments, il ne fera que les transmettre à son fils, il n'aura donc pas besoin de les convertir. La transmission est d'ailleurs transparente car après le *fork*, le fils connaît aussi directement les arguments du père (*argv*) puisqu'il y a recopie des variables.
- ▶ Le deuxième est le programme qui recouvrira le programme du fils. Ce programme prendra 2 arguments (1 entier et une chaîne de caractères) et il affichera la chaîne de caractères et le double de l'entier. Les deux arguments seront transmis "par le père" par l'intermédiaire de la fonction *execl* dans la partie fils du code (après le *fork*).

Exemple : **recouvrant 9 trucs** affichera par exemple "J'ai 18 trucs". Ainsi **Pere 9 trucs** lancera un fils qui exécutera **recouvrant 9 trucs**.

N'oubliez pas de compiler les 2 programmes (séparément). Vous pouvez là encore afficher des messages pour bien comprendre ce qui se passe et **vous devez mettre un message d'erreur juste après la fonction `execl(...)` (message qui sera affiché seulement si le `execl` ne fonctionne pas) ainsi que `exit` pour éviter que le programme ne continue à s'exécuter.**

Prenez l'exemple donné en cours (pour voir où se situe `execl`) et adaptez-le.



### aide C

- ☞ La fonction *main* d'un programme prend comme premier paramètre le nombre d'arguments (`int argc`) de l'exécutable et stocke ces arguments dans le deuxième paramètre qui est un tableau de chaîne(s) de caractères (`char * argv[]`) : `argv[0]` est la chaîne de caractère contenant le nom de l'exécutable, `argv[1]` le premier argument, ...
- ☞ Pour les paramètres de la fonction `execl(...)`, il faut donc mettre après le premier paramètre (qui est le chemin de l'exécutable), tous les arguments (de 0 à N) puis finir par `NULL`. Exemple : `execl("./recouvrant", "recouvrant", argument1, argument2, NULL)`; avec `argument1` et `argument2` des chaînes de caractères (par exemple `char argument1[10]` ;).
- ☞ Pour transformer une **chaîne en entier** (vous en aurez besoin dans le programme `recouvrant`, passer "9" en 9), on peut utiliser `sscanf(chaine, "%d", &entier)` ou la fonction `atoi(chaine)` (`AsciiToInt`) qui renvoie l'entier.
- ☞ Pour transformer un **entier en chaîne de caractères** (dans cet exercice vous n'en aurez pas besoin), vous pourriez utiliser la fonction `sprintf(chaine, "%d", 8)` ; (8 étant l'entier)



### ATTENTION

Cet exercice devra être terminé pour la prochaine séance (TP2)