Ecole Supérieure National d'Informatique

2008 – 2009

TP système d'exploitation :

Réalisation d'un mini Shell Unix.

Réalisation par :

Afifi Sohaib

me@sohaibafifi.com

3 SIQ-2

Arbaoui Taha

tahaarbaoui@gmail.com

3 SIQ-2

Sommaire :

Linux

Linux est un système d'exploitation, tout comme Windows ou MacOS X. Il permet de travailler comme on le ferait sous Windows. Mais il fonctionne différemment.

Linux est un système d'exploitation de type UNIX, multi-tâches et multi-utilisateurs pour machines à processeurs 32 et 64 bits (en particulier les machines de type PC et PowerMac), ouvert sur les réseaux et les autres systèmes d'exploitation.

Certains voient encore Linux comme un horrible système plein de commandes compliquées. Ce n'est plus vrai. Linux possède désormais un système graphique agréable, confortable et simple à utiliser.

On a l'habitude de dire que Linux est un système d'exploitation. En réalité, Linux, c'est uniquement le "**noyau**" du système d'exploitation GNU/Linux. (Les sytèmes d'exploitation Microsoft Windows NT, 2000 et XP ont également un noyau, mais différent: le noyau NT.)

Le noyau s'occupe des basses besognes: la gestion de la mémoire, l'accès aux périphériques (disque dur, lecteur de CD-Rom, clavier, souris, carte graphique...), la gestion du réseau, le partage du temps microprocesseur entre les programmes (multi-tâche), etc.

Contrairement à Windows dont l'interface graphique vous est imposée, il existe différentes interfaces graphiques sous Linux, les principales étant Gnome, KDE et XFCE. Il est même possible de faire fonctionner Linux sans interface graphique, ou même de ne lancer l'interface graphique que quand vous le souhaitez.

GNU/Linux

GNU est un projet qui a apporté des tas d'utilitaires au noyau Linux, tel que le fameux compilateur *gcc*, et les milliers d'utilitaires (tar, tail, man, bash...).

Ces utilitaires GNU, associés au noyau Linux, constituent le système d'exploitation GNU/Linux.

Linux est donc un **noyau**.
GNU est un ensemble de **programmes utilitaires**.
GNU/Linux est le **système d'exploitation.**

Les processus UNIX

Un processus Unix est un programme (commande) en train de s'exécuter.

Un processus est caractérisé par par un numéro qui est unique sur le système (process ID = PID). On peut avoir cet identifiant en utilisant la commande **ps -al** et en regardant dans la colonne PID

```
F   UID   PID  PPID CP PRI NI   SZ  RSS    WCHAN S TT       TIME COMMAND
8     0   304     1  0  54 20 1496 1024 ar_g_hea S console   0:00 /usr/lib/saf
8   606   440   433  0  46 20 1288 1048 modlinka S pts/3     0:00 /bin/csh
8   606   624   440  0  48 202700014800 ar_g_hea S pts/3     0:18 /unige/ow3/b
8   606   641   624  0  57 2018512 3352 ar_g_hea S pts/3     0:00 (dns helper)
8   606  1117   440  0  49 202332812760 redirmin S pts/3     0:01 ugtool
8   606  1146     1  0  49 201560812792 redirmin S pts/3     0:02 /unige/frame
```

Chaque processus est crée par un autre procussus qui est sont père et retourne une valeur.

Un processus UNIX est représenté par

- trois segments mémoire:
  - le segment texte: code + données statiques.
  - le segment données: variables.
  - le segment stack: la pile du processus.
- un environnement:
  - information nécessaire au noyau pour gérer le processus (contenu des registres, priorités, fichiers ouverts,...).
  - peut être modifié par des appels système.

Gestion de Processus

Créér un nouveau processus: fork

La seule manière de créér un *nouveau* processus sous UNIX est d'utiliser l'appel système fork().(Seul le processus init (pid=1) est démarré de manière particulière par le kernel.)

Le processus courant peut créer un processus fils en utilisant l'appel système fork . Cette primitive provoque la duplication du processus courant . Son prototype est le suivant .

Prototype :

```
# include <unistd.h>
pid_t  fork(void )
```

Fonctions :

- crée un nouveau processus (processus fils) par duplication du processus courant (processus père).
- copie les segments de données et de stack du processus père; le segment texte est partagé.
- les ``pid'' du père et du fils diffèrent.
- le fils hérite en partie de l'environnement du père.
- retourne un entier:
  - en cas de succès:
    - 0 dans le fils.
    - pid du fils dans le père.
  - en cas d'échec
    - -1 dans le père.
    - le fils n'est pas créé.

Suite à un *fork*, les processus père et fils poursuivent l'exécution du *même* programme. La valeur retournée par *fork* permet de distinguer entre le père et le fils.

Example:

```
int pid;
pid = fork();
switch(pid){
  case -1 :
    perror("Création procesus");
    break;
```

```
  case 0 :
    printf("Je suis le fils de pid %d et de père
%d\n",getpid(),getppid());
    break;
  default :
    printf("Je suis le père de pid %d et de fils %d\n",getpid(),pid);
}
```

- Ce programme crée un nouveau processus. Le père affiche son pid et celui de son fils et le fils l'inverse.
-
- Les causes d'échec de *fork* peuvent être les suivantes:
- Le nombre maximum de processus en exécution par l'utilisateur est atteint (variable suivant les systèmes: 40 sous AIX, 50 sous ULTRIX, 997 sous Solaris 2.4).
- Il ne reste pas suffisamment de mémoire système disponible pour dupliquer le processus.
- Il n'y a pas assez de *swap space*.

Erreurs :

La primitive fork retourne la valeur 0 au processus fils , et elle retourne l'identificateur du processus créé au processus père . Il est donc impératif de tester sa valeur de retour pour distinguer le code qui doit être exécuté par le processus père de celui qui doit être exécuté par le fils.

En cas d'échec, fork retourne la valeur -1 , et la variable ***errno*** peut prendre les valeurs suivants :

EAGAIN : le nombre maximal de processus pour l'utilisateur courant , ou dans le système , a été atteint

ENOMEM : le noyau n'a pas pu allouer suffisamment de mémoire pour créer un nouveau processus.

Terminaison d'un processus:

Le processus courant se termine automatiquement lorsqu'il cesse d'exécuter la fonction **`main`** , en utilisant l'instruction **`return`** . Il dispose également d'appels système lui permettant d'arrêter explicitement son exécution :

Exit : La fonction exit(int status) de la bibliothèque C standard appelle les (éventuelles) fonctions spécifiées grâce à **`atexit`** et **`onexit`**, puis effectue l'appel système _exit(int status):

Prototype :

```
#include <unistd.h>
void exit(int status)
void _exit(int status)
```

Fonctions :

- termine le processus appelant.
- retourne la valeur du byte de poids faible de `status` au processus père.
- ferme les descripteurs de fichiers ouverts.
- un signal `SIGCHLD` est envoyé au processus père.
- le PPID des processus fils du processus sortant devient 1 (`init`).

Attente de la terminaison d'un processus fils :

Wait : Un processus peut attendre la terminaison d'un de ses processus fils grâce à l'appel système wait:

Prototype :

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *statusp)
```

Fonctions :

- attend la terminaison ou l'arrêt d'un processus fils. Donc, s'il reste au moins un processus fils non terminé (et si un wait a été fait pour tous les fils déjà terminés), le processus appelant est suspendu jusqu'à la terminaison (ou l'arrêt) d'un processus fils.
- retourne le pid d'un processus terminé, -1 si tous sont déjà terminés.
- stocke le *status code* du fils dans l'entier pointé par statusp.

Un processus terminé dont le *status code* n'a pas encore été récupéré est un *zombie* (il occupe une entrée dans la table des processus).

Il y a trois façons pour un processus de se terminer: par un appel à exit, suite à la réception d'un signal fatal ou lors d'un crash de la machine. Le *status code* récupéré par statusp permet de distinguer entre les deux premiers cas

Si le processus fils s'est terminé par exit(exp), le byte de poids faible du *status code* est zéro et le byte de poids immédiatement supérieur est exp .

- Si le processus fils s'est terminé sur réception d'un signal, le byte de poids faible est le numéro de ce signal.

Waitpid : Un processus peut aussi attendre la terminaison d'un de ses processus fils grâce à l'appel système waitpid:

Prototype :

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

Fonctions :

La primitive waitpid suspend l'exécution du processus courant jusqu'à ce qu'un processus fils spécifié par le paramètre pid se termine . Si un processus fils correspond à pid s'est déjà terminé, waitpid retourne le résultat immédiatement.

Le résultat de waitpid dépend de la valeur du paramètre pid :

- Si pid est positif, il spécifie le numéro du processus fils à attendre ;
- Si pid est nul, il spécifie tout processus fils dont le numéro de groupe de processus est égal à celui du processus appelant ;
- Si pid est égal à -1 , il spécifie d'attendre la terminaison du premier processus fils ; dans ce cas , waitpid offre la même sémantique que wait ;
- Si pid est inférieur à -1 , il spécifie tout processus fils dont le numéro de groupe de processus est égal à la valeur absolue de pid .

Deux constantes, déclarées dans <sys/wait.h> , peuvent être utilisées pour initialiser le paramètre options , afin de modifier le comportement de waitpid :

- **WNOHANG** : provoque un retour immediat si aucun processus fils ne s'est encore terminé .
- **WUNTRACED** : Provoque la prise en compte de fils dont l'état change , c'est-à-dire des processus fils dont l'état est passé de prêt à suspendu .

Dans les deux primitives l'état du processus est retourné dans la variable dont l'adresse est passée dans le paramètre **`statusp`**

L'interprétation de cet état est effectuée grâce à des macro-instructions définies dans le fichier d'en-tête `<sys/wait.h>` :

- **`WIFEXITED`** : Non nul si le processus fils s'est terminé par un appel à **`_exit`** ou **`exit.`**
- **`WEXITSTATUS`** : Code de retour transmis par le processus fils lors de sa terminaison.
- **`WIFSIGNALED`** : Non nul si le processus fils a été interrompu par la réception d'un signal.
- **`WTERMSIG`** : Numéro du signal ayant provoqué la terminaison du processus fils.
- **`WIFSTOPPED`** : Non nul si le processus fils est passé de l'état prêt à suspendu.
- **`WSTOPSIG`** : Numéro du signal ayant causé la suspension du processus fils.

Exécuter un programme: exec

La seule manière d'exécuter un nouveau programme sous UNIX consiste à effectuer l'appel système *exec*. Exec remplace le programme exécuté par le processus appelant par un nouveau programme. Attention! il **n'y a pas** création d'un nouveau *processus*: un nouveau *programme* est exécuté dans le contexte du processus appelant. D'ailleurs, le PID du processus appelant est inchangé.

Exec se présente sous plusieurs formes:

```
#include <unistd.h>
int execve(const char *path, const char *argv[])
int execve(const char *path, const char *argv[], const char *envp[])
int execvp(const char *name, const char *argv[])
int execl(const char *path, const char *arg0, ..., const char *argn,
NULL)
int execle(const char *path, const char *arg0, ..., const char *argn,
NULL , char *const envp[])
int execlp(const char *name, const char *arg0, ..., const char *argn,
NULL)
```

L'appel système **exec** retourne une valeur (-1) uniquement en cas d'erreur.

- **path** doit être le nom d'un fichier exécutable par le *user-ID* effectif.
- Par convention, au moins arg0 doit être présent. Il deviendra le nouveau nom du processus (apparaissant lors d'un ps par exemple).
- Les segments texte et données du processus appelant sont remplacés par ceux du programme spécifié.
- Le processus démarre l'exécution du nouveau programme.
- L'environnement du processus est (presque totalement) conservé.

Les fonctions **exec** remplacent le programme en cours d'exécution dans un processus par un autre programme. Lorsqu'un programme appelle la fonction exec, le processus cesse immédiatement d'exécuter ce programme et commence l'exécution d'un autre depuis le début, en supposant que l'appel à exec se déroule correctement.

Au sein de la famille de exec existent plusieurs fonctions qui varient légèrement quant aux possibilités qu'elles proposent et à la façon de les appeler.

Les fonctions qui contiennent la lettre **p** dans leur nom (**execvp** et **execlp**) reçoivent un nom de programme qu'elles recherchent dans le path courant; il est nécessaire de passer le chemin d'accès complet du programme aux fonctions qui ne contiennent pas de p.

Les fonctions contenant la lettre v dans leur nom (execv, execvp et execve) reçoivent une liste d'arguments à passer au nouveau programme sous forme d'un tableau de pointeurs vers des chaînes terminé par **NULL**. Les fonctions contenant la lettre **l** (**execl**, **execlp** et **execle**) reçoivent la liste d'arguments via le mécanisme du nombre d'arguments variables du langage C.

Les fonctions qui contiennent la lettre e dans leur nom (**execve** et **execle**) prennent un argument supplémentaire, un tableau de variables d'environnement. L'argument doit être un tableau de pointeurs vers des chaînes terminé par **NULL**. Chaque chaîne doit être de la forme "VARIABLE=valeur".

Dans la mesure où **exec** remplace le programme appelant par un autre, on n'en sort jamais à moins que quelque chose ne se déroule mal.

Les arguments passés à un programme sont analogues aux arguments de ligne de commande que vous transmettez à un programme lorsque vous le lancez depuis un shell. Ils sont accessibles par le biais des paramètres **argc** et **argv** de main. Souvenez-vous que lorsqu'un programme est invoqué depuis le shell, celui-ci place dans le premier élément de la liste d'arguments (**argv[0])** le nom du programme, dans le second (**argv[1])** le premier paramètre en ligne de commande, etc. Lorsque vous utilisez une fonction **exec** dans votre programme, vous devriez, vous aussi, passer le nom du programme comme premier élément de la liste d'arguments.

Erreur s:

L'appel système **execXXX** provoque le recouvrement des segments de code , de données et de pile par ceux du programme spécifié . En cas de succès , il n'ya donc pas de retour , puisque le processus appelant exécute alors un nouveau programme .

En cas d'échec, **`execXXX`** retourne la valeur -1, et la variable **errno** peut prendre les variables suivantes :

- **`E2BIG`** : la liste des arguments ou des variables d'environnement est de taille trop importante.
- **`EACCESS`** : le processus n'a pas accès en exécution au fichier spécifié par **pathname**
- **`EFAULT : pathname`** contient une adresse invalide.
- **`EANAMETOOLANG : pathname`** spécifie un nom de fichier trop long.
- **`ENOENT : pathname`** se réfère à un nom de fichier inexistant.
- **`ELOOP`** : Un cycle de lien symbolique a été rencontré
- **`ENOEXE`** : Un fichier spécifie par **`pathname`** n'est pas un programme exécutable.
- **`ENOMEM`** : la mémoire disponible est trop réduite pour exécuter le programme.
- **`ENOTDIR`** : Un des composants de **`pathname`**, utilisé comme nom de répertoire, n'est pas un répertoire.
- **`EPERM`** : Le système de fichiers contenant le fichier spécifié par **`pathname`** a été monté avec des options interdisant l'exécution de programme.

L’interpréteur de commandes (Shell) :

L'interpréteur de commandes est l'interface entre l'utilisateur et le système d'exploitation, d'où son nom anglais «**shell**», qui signifie «coquille».

Le shell est ainsi chargé de faire l'intermédiaire entre le système d'exploitation et l'utilisateur grâce aux lignes de commandes saisies par ce dernier. Son rôle consiste ainsi à lire la ligne de commande, interpréter sa signification, exécuter la commande, puis retourner le résultat sur les sorties.

Il existe plusieurs shells, les plus courants étant **sh** (appelé «Bourne shell»), **bash** («Bourne again shell»), **csh**(«C Shell»), Tcsh («Tenex C shell»), **ksh** («Korn shell») et **zsh** («Zero shell»).

Sous UNIX, la ligne de commande a toujours été le moyen privilégié de communication avec l'ordinateur. Le Bourne shell (**sh**) est l'interpréteur originel de l'environnement UNIX. À son époque, sa grande originalité était l'utilisation de tubes (caractère “|”), qui permettent de connecter la sortie d'une commande à l'entrée d'une autre. On peut ainsi écrire des commandes complexes à partir de commandes simples.

GNU/Linux, la famille BSD et autres dérivés d'UNIX ont hérité de cette particularité, même s'ils disposent également d'interfaces graphiques complètes (notamment X Window et Xorg).

Parmi ces dérivés, Mac OS X se présente comme un environnement essentiellement graphique, mais dispose d'un interpréteur de commandes (**tcsh** ou **bash**) qui s'active à partir du programme Terminal.

D'autres langages de scripts tels que **Perl, Python** ou **Ruby**, remplacent progressivement les interpréteurs qui sont encore prédominants dans les environnements de démarrage de systèmes UNIX.

La dépendance des interpréteurs vis-à-vis de commandes externes rend les scripts peu portables, même si les interpréteurs eux-mêmes ont été portés sur des environnements autres que leur environnement originel UNIX.

Le travail de tout interprète de commande peut se résumer à l'algorithme très simple suivant:

```
TANT QUE l'utilisateur ne ferme pas la session
FAIRE
# Émettre un signe d'invite (<em>prompt</em>)
# Lire la ligne courante
# Exécuter la commande indiquée sur cette ligne
FIN
```

Chaque utilisateur possède un shell par défaut, qui sera lancé à l'ouverture d'une invite de commande. Le shell par défaut est précisé dans le fichier de configuration **/etc/passwd** dans le dernier champ de la ligne correspondant à

l'utilisateur. Il est possible de changer de shell dans une session en exécutant tout simplement le fichier exécutable correspondant, par exemple :

/bin/bash

Invite de commande (prompt)

Le shell s'initialise en lisant sa configuration globale (dans un fichier du répertoire **/etc/**), puis en lisant la configuration propre à l'utilisateur (dans un fichier caché, dont le nom commence par un point, situé dans le répertoire de base de l'utilisateur, c'est-à-dire **/home/nom_de_l_utilisateur/.fichier_de_configuration**), puis il affiche une invite de commande (en anglais prompt) comme suit :

```
machine:/repertoire/courant$
```

Par défaut dans la plupart des shells le prompt est composé du nom de la machine, suivi de deux points (:), du répertoire courant, puis d'un caractère indiquant le type d'utilisateur connecté :

- «$» indique qu'il s'agit d'un utilisateur normal
- «#» indique qu'il s'agit de l'administrateur, appelé «root»

Notion de ligne de commande

Une ligne de commande est une chaîne de caractères constituée d'une commande, correspondant à un fichier exécutable du système ou bien d'une commande du shell ainsi que des arguments (paramètres) optionnels :

```
ls -al /home/jf/
```

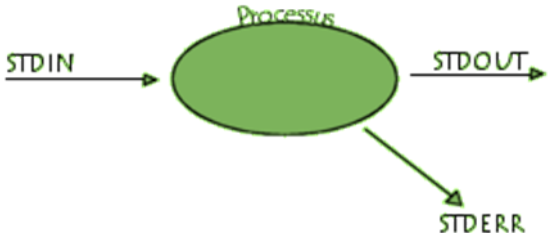
Dans la commande ci-dessus, ls est le nom de la commande, -al et /home/jf/ sont des arguments. Les arguments commençant par - sont appelés options. Pour chaque commande il existe généralement un certain nombre d'options pouvant être détaillées en tapant une des commandes suivantes :

```
commande --help
commande -?
man commande
```

Entrées-sorties standard

Lors de l'exécution d'une commande, un processus est créé. Celui-ci va alors ouvrir trois flux :

- **Stdin** : appelé entrée standard, dans lequel le processus va lire les données d'entrée. Par défaut stdin correspond au clavier ; STDIN est identifié par le numéro 0 .
- **Stdout :** appelé sortie standard, dans lequel le processus va écrire les données de sortie. Par défaut stdout correspond à l'écran ; STDOUT est identifié par le numéro 1 .
- **Stderr** : appelé erreur standard, dans lequel le processus va écrire les messages d'erreur. Par défaut stderr correspond à l'écran. STDERR est identifié par le numéro 2 .



Par défaut lorsque l'on exécute un programme, les données sont donc lues à partir du clavier et le programme envoie sa sortie et ses erreurs sur l'écran, mais il est possible de lire les données à partir de n'importe quel périphérique d'entrée, voire à partir d'un fichier et d'envoyer la sortie sur un périphérique d'affichage, un fichier, etc.

Redirections

Linux, comme tout système de type Unix, possède des mécanismes permettant de rediriger les entrées-sorties standards vers des fichiers.

Ainsi, l'utilisation du caractère «>» permet de rediriger la sortie standard d'une commande située à gauche vers le fichier situé à droite :

```
ls -al /home/jf/ > toto.txt
echo "Toto" > /etc/monfichierdeconfiguration
```

La commande suivante est équivalente à une copie de fichiers :

```
cat toto > toto2
```

La redirection «>» a pour but de créer un nouveau fichier. Ainsi, si un fichier du même nom existait, celui-ci sera écrasé. La commande suivante crée tout simplement un fichier vide :

```
> fichier
```

L'emploi d'un double caractère «>>» permet de concaténer la sortie standard vers le fichier, c'est-à-dire ajouter la sortie à la suite du fichier, sans l'écraser.

De manière analogue, le caractère «<» indique une redirection de l'entrée standard. La commande suivante envoie le contenu du fichier toto.txt en entrée de la commande cat, dont le seul but est d'afficher le contenu sur la sortie standard (exemple inutile mais formateur) :

```
cat < toto.txt
```

Enfin l'emploi de la redirection «<<» permet de lire sur l'entrée standard jusqu'à ce que la chaîne située à droite soit rencontrée. Ainsi, l'exemple suivant va

lire l'entrée standard jusqu'à ce que le mot STOP soit rencontré, puis va afficher le résultat :

```
cat << STOP
```

Pipes :

Un **tube Unix**, ou **pipeline**, ou **pipe** est un ensemble de processus chaînés par leurs flux standard, de sorte que la sortie d'un processus (**stdout**) alimente directement l'entrée (**stdin**) du suivant. Chaque connexion est implantée par un tube anonyme. Les programmes filtres sont souvent utilisés dans cette configuration. **Douglas McIlroy** a inventé ce concept pour les shells Unix et le nom anglais découle de l'analogie avec un pipeline physique.

Le pipe est caractérisé généralement par le symbole **|**.

Par exemple :

```
programme1 | programme2
```

Le programme *programme1* est exécuté par le système qui envoie les résultats au *programme2* qui à son tour renvoie les résultats sur la sortie standard du système.

Le pipe est très utilisé sur Unix et plus particulièrement pour associer les commandes **cat** et **grep** pour sélectionner des éléments particuliers dans un fichier, par exemple sous Unix:

```
cat fichier | grep "mot cle" | sort
```

Plusieurs pipes peuvent être réunis sur une même ligne, ouvrant la voie à des connexions entre les différentes opérations, et ainsi la réalisation de script shell plus compacts et mieux construits.

Tubes anonymes et tube nommés :

La gestion des tubes est intégrée dans le système de fichiers. L'accès aux tubes est réalisé par des descripteurs d'entrées / sorties : un descripteur pour la lecture dans le tube et un descripteur pour l'écriture dans le tube.

Historiquement, le premier type de tubes est le tube anonyme .Il est créé par un processus et la transmission des descripteurs associés ne se fait que par héritage vers ses descendants. Ce mécanisme est contraignant puisqu'il ne permet la communication qu'entre processus dont un ancêtre commun est le créateur du tube.

Les tubes nommés (FIFO ou named pipe dans la littérature anglo-saxonne) permettant lever cette contrainte car ils sont manipulés exactement comme des fichiers en ce qui concerne les opérations d'ouverture , de fermeture , de lecture et d'écriture. Ceci est possible car ils existent physiquement sur le système de fichiers :

```
Sohaib# ls –al tube_exemple
Prw-r—r--  1    dumas    users   0  May  1  16:17
tube_exemple
Sohaib# file tube_exemple
tube_exemple :  fifo (named pipe)
```

Il est possible d'identifier un fichier de type tube nommé par l'attribut p affiché par la commande ls. Il est possible de créer un tube nommé sous l'interpréteur de commandes grâce à la commande.

Appels système :

Tube anonymes :

Le création de pipes anonymes se fait à l'aide du system call pipe

```
#include <unistd.h>
int pipe(int fd[2]);
```

l'appel à pipe crée une paire de *file descriptors* stockée dans fd. fd[0] sert à la lecture du pipe et fd[1] sert à l'écriture dans le pipe.

Pour établir une communication (unidirectionnelle!) à l'aide d'un pipe, il faut que le processus père crée un pipe avant de faire un (ou plusieurs) appel(s) à fork. Dès lors, après le **`fork`**, le processus père et le(s) processus fils peuvent accéder à la pipe en utilisant **`fd`**.

Pour une communication bidirectionnelle, il faut créer deux *pipes*.

Tube nommés :

La création de pipe nommée se fait à l'aide du *system call* **`mkfifo`**

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

Une fois créé, la pipe nommée apparaîtra comme un fichier spécial dans le système le fichier, à l'endroit indiqué par **`pathname`**. Le paramètre mode quant à lui, permet de donner les permissions (lecture, écriture, ...) à ce fichier spécial (pour plus de détails sur les constantes utilisables pour mode.

Implementation d'un MiniShell :

Le but de ce TP est de programmer un *shell* minimal en C.

Analyse du travail:

Notre *shell* sera une simple boucle d'interaction qui affiche une invite de commande coloré `(prompt())`, lit une ligne entrée au clavier (`lire()`), analyser cette ligne (decoupe()), l'exécute, et recommence.

Le *shell* quitte quand l'utilisateur tape **Contrôle+D** , **quit** ou **exit** à la place d'une commande.

On a rendre l'édition de la ligne de commande un peu plus conviviale: support des flèches pour la navigation et la correction, accès à l'historique des commandes, complétion automatique, etc. C'est difficile à faire à la main! Heureusement, il existe la bibliothèque **readline** de **GNU** pour faire cela.

Si une commande est en cours d'exécution, un appui sur Contrôle+C interrompe la commande et redonne la main à l'utilisateur sans toutefois quitter le *shell*. On utilisera pour cela `sigaction(2)`.

```
struct sigaction sig;
sig.sa_flags = 0;
sig.sa_handler = child_signal;
/* désactivation l'interruption par Contrôle+C */
sig.sa_handler = SIG_IGN;
sigaction(SIGINT, &sig, NULL);
```

Pour la gestion des plans d'exécutions on a aussi traité deux types de commandes: une commande terminée par `&` sera lancée en tâche de fond tandis qu'une commande non terminée par `&` sera lancée au premier plan. À un moment donné, il y a au plus une commande au premier plan et un nombre arbitraire de commandes en tâche de fond. (il faut aussi faire attention aux interactions entre le *handler* du signal `SIGCHLD` et la commande `waitpid(2)`...)

Dans le cas d'un programme lancée en arriere plan le *shell* n'attend pas qu'il termine mais redonne tout de suite la main à l'utilisateur. Toutefois, le *shell* devra tout de même détecter la terminaison (asynchrone) des programmes en tâche de fond afin de:

- éviter les processus *zombies*,
- prévenir l'utilisateur de la bonne ou mauvaise terminaison de la commande.

Il faudra pour cela gérer le signal SIGCHLD grâce à un *handler* (voir sigaction(2)). Notant qu'on ne peut recevoir qu'un seul signal SIGCHLD pour indiquer la terminaison de plusieurs fils.

```
struct sigaction sig;
sig.sa_flags = 0;
sig.sa_handler = child_signal;
sigemptyset(&sig.sa_mask);
sigaction(child_signal,&sig,NULL);
```

On n'a pas travaillé avec la fonction (fgets) -dans la lecture du clavier- qui peut être interrompue par l'exécution asynchrone du *handler*.

On a géré la substitution des ~ en répertoire *home*. Il suffit de regarder la valeur de la variable d'environnement HOME. Et on ajouté la colorisation de la commande **ls** et **grep** en ajoutant un alias vers (**ls ou grep) --color=auto**.

On a ajouté la **gestion des caractères spéciaux** (les méta caractères) dans les noms de fichiers (*, ?, etc.).Il est bien sûr possible de parcourir à la main les répertoires pour trouver tous les fichiers correspondant à un motif donné. C'est long à programmer! On risque de plus d'interpréter les motifs d'une manière légèrement différente de celle du shell préféré de l'utilisateur, ce qui ne manquera pas de l'agacer. Heureusement, il existe une fonction standard glob(3) pour faire ce travail à notre place!

Chaque fois on décompose notre ligne en tableau des commandes qui marchent en séquentielle (séparées par des ;) (**cmds_seq**), grâce au procédure **decoupe_ligne_seq** .

Pour chaque case (commande) du tableau **cmds_seq** si elle contient un '&' on la redécompose en tableau des commandes qui marchent en parallèle (**cmds_par**), grâce au procédure **decoupe_ligne_par** .

Pour chaque case (commande) du tableau **cmds_par ou cmds_seq** (selon le cas) on détecte l'existence de la redirection ou pipes lors de la décomposition.

Puis on traite les trois cas: redirection, pipes ou simple(en arrière plan ou en premier plan).

Pour la redirection on suppose que le reste de la ligne après le '<' ou '>' ou '>>'est le nom du fichier et on le nettoie avec la fonction **clean_filename**. Et on traite les trois types de redirection

- **lire** : en cas de '<'.

- **écriture** : en cas de '>'
- **mise à jour**: en cas de'>>' c'est-à-dire ajouter la sortie à la suite du fichier, sans l'écraser.

Pour l'**<u>exécution des commandes</u>** on a utilisé **execvp (**pour chercher dans le PATH aussi**)**

Cette appelle est lancée dans le fils après le **fork** (pour qu'on ne sort pas après l'exécution de cette commande)

Si la commande est en premier plan on doit l'attendre dans le processus père.

On a choisi comme commandes prédéfinis (interne) les commandes suivantes :

- **quit , exit** : pour sortir du shell.
- **set** : pour changer ou ajouter une variable d'environnement
  syntaxe : **set var valeur**
- **history :** affiche l'historique des commandes.
- **cd :** change le répertoire courant. Cette commande accepte aussi la variable ~ du répertoire HOME.
- **Help :**affiche l'aide sur les commandes internes.

Les procedures et les fonctions du programme(en prototypes) :

Colorer un texte :

```
char* colorer(int attribute, int foreground, int background, char *text) ;
```

Initialiser les variables globales :

```
void init() ;
```

Affichage du prompe :

```
void prompt();
```

Netoyer le nom du fichier :

```
char *clean_filename(char *filename);
```

lire une ligne des commandes a partire du clavier :

```
void lire();
```

Découper la ligne et extraire les commandes séquentielles :

```
void decoupe_ligne_seq();
```

Découper la ligne et extraire les commandes parallèles :

```
void decoupe_ligne_par();
```

Découper la ligne et extraire les commandes en pipes :

```
void decoupe_ligne_pipe(char input[]);
```

Découper la commande en mots et les mettre dans elem :

```
void decoupe_cmd(char line[] , char *elem[]);
```

Attente du signal de fils :

permet à child_signal de notifier à execute que la commande au premier plan vient de se terminer la variable étant partagée entre le programme principal (lance_commande) et un handler de signal asynchrone (child_signal), elle dont être marquée volatile

```
void child_signal(int signal);
```

Exécuter une commande simple:

```
void execute(char* elem[]);
```

Exécuter les pipes récursivement :

```
void recursive_pipe(int i);
```

Exécuter les commandes en pipes :

```
void runPipe();
```

Deuxieme methode pour executer les commandes en pipes:

Cette méthode utilise des fichiers temporaires.

```
void ExecutePipes();
```

Exécuter une commande contient une redirection de type fileid (STDOUT ou STDIN) vers file :

```
int runRedirectedCommand(char *items[], char *file, int fileid) ;
```

Le programme principal :

```
int main();
```

Initialiser la liste de l'auto complétion (fichier rline.c):

```
void initialize_readline ();
```

Statistique sur le code source du programme :

- Nombre de lignes de code : 612.
- Nombre de lignes de commentaires : 147.
- Nombre de lignes commentaires et code : 37.
- Nombre total des lignes : 942.
- Nombre de fichiers : 4.

Jeu d'essais :

Notre mini Shell a pu interpréter ces testes :

```
Programme
Programme&
Programme 1; programme 2;….. ;programme10[ ;]
Programme 1;|programme2 |….. |programme10
ls –l *.c  l*.txt
cd
Programme 1; programme 2|programme3;….. ;programme10[ ;]
ls -l > file.txt
ls –a ../ >> file.txt
grep a < file.txt
exit
cd ~
set PATH /usr/bin
```

Remarque :

Pour compiler le code source vous devez avoir les bibliothèques suivantes :

- Libc6-dev.
- Libreadline-dev.

On a crée un script shell '**run.sh**' qui va tous faire pour la compilation et l'exécution.