



UNIVERSITÉ DE MONTPELLIER
Faculté de Sciences
Département Informatique



INTERPRÉTEUR DE COMMANDES

☐ JONATHAN TWAMBA SAIBA

☐ IBRAHIMA TOUNKARA

☐ JOEL BEYA NTUMBA

— Encadrant : MICHEL LECLÈRE

À chaque fois que nous utilisons un ordinateur, une imprimante, un objet connecté ou tout automate, nous passons par une interface graphique qui nous affiche ce que nous attendons après avoir donné des instructions.

Mais derrière tout ça, il y a un programme qui lit tout ce que vous écrivez, traite vos saisies pour en faire des instructions si possible, et enfin les exécute. Ce sont toutes ces étapes qui constituent un **interpréteur de commandes**

Table des matières

1	Introduction	2
1.1	Introduction	2
1.2	Objet du projet	2
1.2.1	Problématique	2
1.2.2	Domaine de l'informatique	2
1.2.3	Utilisateur	2
1.2.3.1	Utilisateur de base	3
1.2.3.2	Programmeur	3
1.2.3.3	Administrateur système	3
1.3	Le Projet	3
2	Mise en Oeuvre	5
2.1	Analyse du besoin	5
2.2	Boucle de Fonctionnement du SHELL	6
2.3	Choix du langage	6
2.4	Nos commandes	6
2.5	Fonctions majeures du programme	7
2.5.1	Prompt()	7
2.5.2	LireCommande()	7
2.5.3	TraiterCommande()	7
2.5.4	ExecuterCommande()	7
2.6	Gestions des Signaux (SIGINT)	8
2.7	Redirections	8
2.7.1	Rediriger la sortie dans un fichier : >	8
2.7.2	Ajouter la sortie à un fichier : > >	8
2.7.3	Rediriger l'entrée : <	8
2.8	Communications entre différentes commandes à travers les tubes	9
2.9	Gestion des Variables	9
2.10	Autocomplétion	9
2.11	Gestion de l'historique	9
3	Conclusion	10
	Bibliographie	11
A	Annexe	I

1

Introduction

1.1 Introduction

Dans le cadre du projet informatique TERL2, on a réalisé un interpréteur de commandes s'inspirant du SHELL du système GNU/LINUX mais sans la partie script en mode interactif (selon la recommandation du professeur Michel LECLERE).

Un interpréteur de commandes est un logiciel système faisant partie des composants de base d'un système d'exploitation (UNIX).

Il lit et interprète les commandes qu'un utilisateur tape au clavier dans l'interface graphique en ligne de commande (TERMINAL/GNU). Écrit en langage C, ce système est ainsi portable sur la plupart des architectures systèmes.

1.2 Objet du projet

1.2.1 Problématique

L'interpréteur de commandes SHELL duquel nous nous sommes inspirés est un outil qui a été pensé dans les années 60/70 dans les laboratoires de recherche informatique d'AT&T aux USA pour pallier les besoins des ingénieurs développant le système d'exploitation UNIX dudit laboratoire.

1.2.2 Domaine de l'informatique

Le domaine de notre projet est la programmation système. Et l'interpréteur de commande est un système interactif qui sert d'interface entre la commande entrée par l'utilisateur sur l'entrée standard et le noyau (ici LINUX), il est la couche externe du système, un exécutable qui interprète et transmet au noyau les commandes que l'utilisateur donne à l'entrée standard, et en suite retourne le résultat du traitement de la commande par le noyau.

1.2.3 Utilisateur

L'interpréteur de commande associe à chaque utilisateur des renseignements suivants :

- Un nom de login : L'enregistrement de l'utilisateur

- Un mot de passe : Le password pour son authentification
- un UID (**U**ser**I**Dent**I**fier) : Son identification pour les droits d'accès à ces ressources ou à des domaines et donc pour la sécurité du système
- Un GID (**G**roup**I**Dent**I**fier) : Un UID ou plusieurs UID
- Un répertoire HOME : Emplacement du répertoire personnel de l'utilisateur actuellement connecté.
- Shell : Par défaut, c'est le bash sur le système UBUNTU

Un interpréteur de commande différencie les utilisateurs en utilisateur de base, en programmeur ou administrateur système.

1.2.3.1 Utilisateur de base

Utilise simplement les applications mises à sa disposition, il possède les droits d'accès traditionnels, et doit veiller à utiliser les ressources (disque, CPU,...) avec modération afin de ne pas surcharger le système.

1.2.3.2 Programmeur

Possède les mêmes droits que l'utilisateur de base, et a de plus, la possibilité de programmer ses propres applications. Il a accès aux outils de développement installés sur le système et à des interpréteurs pour pouvoir exécuter ses programmes et applications.

1.2.3.3 Administrateur système

En plus des droits du programmeur, il possède d'autres possibilités telles que :

- la gestion du système ;
- les droits d'accès plus étendus (il peut tout gérer dans l'arborescence de son système) ;
- La possibilité de création des comptes pour tous les utilisateurs de son serveur ;
- la veille du bon fonctionnement général du système en surveillant la régularisation de charge du système et des ressources de la machine ;
- l'installation des outils nécessaires à la communauté.

1.3 Le Projet

Pour notre projet, on ne s'intéressera pas au fonctionnement du système d'exploitation UNIX (dont Linux, le noyau, est un des composants de base) ainsi que l'implémentation du terminal (le GNOME du GNU/LINUX).

Nous avons donc hors mis la partie script/mode-interactif, réimplémenté ces fonctionnalités ci-dessous :

- affichage et modification du prompt ;
- lecture et traitement de caractères entrés au clavier ;

- interprétation et exécution de quelques commandes internes ;
- gestion des signaux (SIGINT) ;
- redirection des entrées et sorties ;
- communitation entre différentes commandes par les tubes ;
- complétion avec la tabulation et la double tabulation ;
- gestion des variables et des variables globales ;
- gestion de l'historique avec la bibliothèque readlines.

2

Mise en Oeuvre

Dans ce chapitre, nous allons parler des différentes étapes du projet de l'analyse du besoin à l'implémentation.

2.1 Analyse du besoin

Le premier point sur lequel nous nous sommes concentrés est l'analyse du sujet avant toute implémentation. Vous remarquerez que nous avons défini des fonctions externes comme étant internes pour des questions de personnalisation. Nous voulions faire un Shell à notre façon.

D'abord, nous avons étudié le fonctionnement du "Bash". Cela nous a permis d'avoir une idée précise des fonctions d'un interpréteur de commandes et au final de faire le choix sur les fonctions à implémenter.

Niveau implémentation, il nous fallait une structure pour nous permettre de gérer les commandes et leurs paramètres et une deuxième (liste chaînée) pour nous permettre de gérer les variables.

```
structure ParamCmd {  
  nom : chaine de caractères  
  argv : chaine de caractères  
  argc : entier  
};
```

```
structure Variables{  
  nom : chaine de caractères  
  valeur : chaine de caractères  
  suivant : structure variables // pointe vers l'adresse suivante  
} Variables;
```

2.2 Boucle de Fonctionnement du SHELL

```
TANT QUE l'utilisateur ne ferme pas la session
FAIRE
Emettre un signe d'invite (prompt)
Lire la ligne courante
Traiter la commande entree
Executer la commande indiquee sur cette ligne
FIN
```

https://fr.wikipedia.org/wiki/Interpr%C3%A9teur_de_commandes

2.3 Choix du langage

Nous avons choisi le C, du moins, il nous a été recommandé pour pouvoir programmer à bas niveau. Nous avons le choix entre utiliser les fonctions de la librairie standard et les appels systèmes. Notre choix s'est vite porté sur la librairie standard car elle a des fonctions déjà implémentées que nous pouvions réutiliser. La principale raison du choix était la portabilité sur différentes architectures.

2.4 Nos commandes

```

▼ Terminal - jtwambasaiba@l12: ~/TER/Interpreteur_de_commande_TER/Joel_beya - + x
Fichier Éditer Affichage Terminal Onglets Aide
Interpréteur de commandes 'MSH'
Ces commandes de msh sont définies de manière interne.
Tapez toujours « aide » pour voir cette liste.

  exit/finir          Quitter le msh
  chdos <dossier>     Changer le répertoire courant
  aide               Afficher l'aide
  nettoyer           Nettoyez l'ecran
  powerD             Afficher le répertoire courant
  powerDMSH          Afficher le répertoire courant de notre msh
  moi                Afficher le nom de l'utilisateur
  afficher           Afficher le contenu d'un fichier
  creerD             Créer un nouveau dossier
  detruireD          Effacer un dossier
  copier             Copier d'un fichier
  creerF             Créer un nouveau fichier
  renommer/deplacer  Renommer ou déplacer le fichier
  detruireF          Effacer un fichier
  set                Lister les variables
  unset              Supprimer une variable
  echo               Afficher une variable
  detruire_processus Tuer un processus
  revoire            Historique des commandes

jIj_>

```

Figure 2.1 – Listes des commandes internes

2.5 Fonctions majeures du programme

Nous avons découpé le programme en plusieurs parties selon le fonctionnement d'un interpréteur de commandes.

2.5.1 Prompt()

L'interpréteur indique qu'il est prêt à recevoir une commande. Nous avons implémenté un prompt dynamique pour permettre à l'utilisateur d'avoir plusieurs choix pour l'affichage de son prompt. Par exemple, l'utilisateur a le choix d'afficher les variables d'environnement ou encore de renommer le prompt comme il le souhaite.

```
void prompteur()
```

2.5.2 LireCommande()

Elle nous permet de récupérer tout ce que l'utilisateur entre comme commande.

```
int lire_commande(char* syf (resultat de la saisie
de l'utilisateur))
```

2.5.3 TraiterCommande()

Dans cette phase, nous traitons la commande comme le nom de la fonction l'indique. Nous récupérons le nom de la commande et ses arguments à travers les paramCmd. Nous vérifions si la commande existe dans notre fonction void vérificateurCommande (saisie utilisateur) dans laquelle on compare le nom de la commande et les fonctions que nous avons implémentées en utilisant "strcmp" de "string.h" de la librairie standard.

```
int traiter_commande(struct param_cmd* commande)
```

2.5.4 ExecuterCommande()

C'est la dernière phase avant le réaffichage du prompt ou la fin du programme selon la commande entrée par l'utilisateur. A cette étape, nous gérons les processus et exécutons les commandes à travers "execvp" de la bibliothèque de la librairie standard "unistd.h" en recherchant par la même occasion le chemin. A travers cette même fonction, nous créons des processus fils. L'interruption par "CTRL+C" est désactivée si on se trouve dans le processus père et réactivée quand on est dans le fils. Aussi, notre fonction est capable d'exécuter une commande en arrière-plan à travers le ";". C'est-à-dire, le père n'attend pas que le processus fils ait fini pour passer à autre chose.

```
int executer_commande()
```

2.6 Gestions des Signaux (SIGINT)

SIGINT est envoyé lorsqu'un utilisateur presse la touche d'interruption du processus, typiquement Ctrl+C, dans notre cas. Alors nous avons absorbé le signal pour éviter que notre programme soit interrompu. Par contre on a fait en sorte d'interrompre les sous-programmes (ceux qui sont lancés dans le processus fils) quand l'utilisateur fait CTRL+C. Pour ce faire nous avons utilisé la structure "sigaction".

```
sig.sa_flags = 0;
    sig.sa_handler = SIG_IGN;
    sigaction(SIGINT, &sig, NULL);
```

2.7 Redirections

2.7.1 Rediriger la sortie dans un fichier : >

On peut rediriger la sortie standard d'une commande vers un fichier (caractère ">"). Le résultat de la commande sera placé dans le fichier au lieu de s'afficher sur l'écran. Exemple : `ls -l > toto.txt` Le résultat de `ls -l` ne s'affiche pas à l'écran, mais il est placé dans le fichier "toto.txt". Pour cela, nous créons un fichier qui portera le nom du fichier destination entré par l'utilisateur et écrivons ensuite le résultat de la commande. Si le fichier existe déjà on l'écrase.

```
int redirection_Out(char* syf(resultat de la saisie de
l'utilisateur), char* argv[], char* filename)
```

2.7.2 Ajouter la sortie à un fichier : > >

On veut parfois ajouter la sortie d'un programme à un fichier, sans effacer ce qui précède. Or, par défaut, si l'on tape plusieurs fois " `ls -l > toto` " à chaque fois, le contenu antérieur du fichier toto est écrasé par le contenu ultérieur. Pour éviter cela, il existe l'outil de redirection > >.

Ainsi, si vous tapez " `ls -l > > toto` " le fichier toto contiendra à la suite tout ce que vous a renvoyé la commande. Dans ce cas, nous ouvrons le fichier s'il existe déjà, sinon on le crée et ajoute le résultat contrairement à la redirection simple qui écrase le fichier.

2.7.3 Rediriger l'entrée : <

On peut aussi rediriger l'entrée standard d'une commande (caractère "<"). La commande lira alors le fichier au lieu du clavier. De cette façon, nous récupérerons le ré-

sultat de la commande de droite puis nous la traitons avec la commande de gauche. Exemple : "afficher < message" va afficher le contenu de message.

```
int redirection_In(char* syf(resultat de la saisie de
l'utilisateur), char* argv[], char* filename) {
```

2.8 Communications entre différentes commandes à travers les tubes

Le "|" permet de connecter deux processus. La sortie de la première instruction est utilisée en entrée de la deuxième et ainsi de suite. Le déclenchement de la fonction exécuterCommande dans la deuxième instruction dépendra donc de la première.

2.9 Gestion des Variables

Dans cette partie nous avons notre structure variables qui va nous permettre de manipuler des listes chaînées et ainsi gérer nos variables. Par exemple, lorsqu'on fait val=16, on affecte la valeur 16 à la variable val.

```
void ajout_Variable(char* nom_var, char val_var)
```

2.10 Autocomplétion

L'autocomplétion permet par exemple d'éviter de saisir complètement un nom de fichier. Il suffit donc de taper sur la touche "tab" dans l'interpréteur. Exemple avec toto.txt : L'appui sur la touche "tab" va afficher celui-ci sous réserve qu'il n'y ait qu'un fichier commençant par t. En cas de présence de plusieurs fichiers, il faut affiner en tapant la seconde lettre, puis éventuellement la troisième ...

Si nous trouvons une seule concordance, nous recopions le nom de fichier dans la chaîne readline, sinon rien ne se passe.

2.11 Gestion de l'historique

Nous créons un tableau pour nous permettre de stocker ce qui sera entré par l'utilisateur. Nous ouvrons ensuite le fichier .myshell_history qui enregistre les commandes saisies par l'utilisateur et nous lisons toutes les lignes et les stockons. Nous avons été aidé par la bibliothèque "readline/history.h", la bibliothèque "readline/readline.h" et "glob.h" que nous avons importé et nous avons utilisé quelques fonctions comme :

```
void add_history (const char *string)
HIST_ENTRY * remove_history (int which)
void clear_history (void)
```

3

Conclusion

Tout au long du projet, nous avons rencontré certaines difficultés liées à la synchronisation de nos heures de travail et à l'implémentation de certaines fonctions qui étaient très complexes compte tenu du niveau que nous avions.

Très vite, nous nous sommes remis à niveau grâce aux efforts fournis et au travail personnel que nous avons effectué. Le projet nous a permis de rehausser les connaissances en C que nous avons acquises dans le cours de système d'exploitation et de nous familiariser avec le langage.

Cette approche n'est qu'un début pour plusieurs autres projets beaucoup plus étendus.

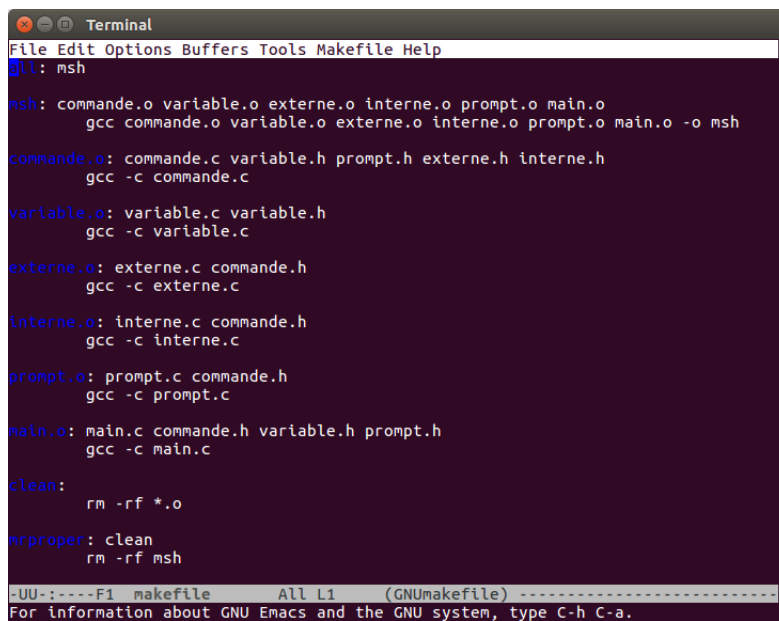
Nous espérons que nous étions à la hauteur et que notre effort sera apprécié. A cet effet, nous remercions Michel LECLERE pour son suivi, la compréhension et la patience dont il a fait preuve à notre égard sur toute la durée du projet.

Bibliographie

- [1] Jean-Marie Rifflet et Jean-Baptiste Yunes (2003) Unix Programmation et communication.
- [2] Jean-Paul GOURRET (2012) Programmation Système - Maîtrisez les interfaces système avec le langage C sous Linux.
- [3] Christophe Blaess (2005) Programmation système en C sous Linux : Signaux, processus, threads, IPC et sockets.
- [4] Mathieu Nebra (2000) <https://openclassrooms.com/courses/reprenez-le-contrôle-a-l'aide-de-linux>.
- [5] Joëlle Delacroix (2009) Linux - Programmation système et réseau.
- [6] Arnold Robbins et Nelson H. F. (2005) Beebe Introduction aux Scripts Shell Automatiser les tâches Unix.
- [7] Nicolas Pons et Renaud Medici (2017) Linux Administrez le système : exercices et corrigés.
- [8] Kiki Novak (2017) Débuter avec LINUX Maîtrisez votre système aux petits oignons.
- [9] Richard Blum (2017) Linux pour les nuls.
- [10] Nicolas Pons et Renaud Medici (2017) Linux - Utilisation et administration avancée du système.
- [11] Joëlle Delacroix (2016) Linux Programmation système et réseau - Cours et exercices corrigés.
- [12] Philippe Pinchon (2016) Linux - Administration avancée Maintenance et exploitation de vos serveurs.
- [13] Philippe Pierre (2015) Debian GNU/Linux Vers une administration de haute sécurité.
- [14] Jean-Francois Bouchaudy (2014) Linux Administration - Tome 1 Les bases de l'administration système.
- [15] Christophe Blaess (2016) Développement système sous Linux Ordonnancement multitâche, gestion mémoire, communications, programmation réseau.
- [16] Cameron Newham et Bill Rosenblatt (2006) Le shell bash.
- [17] Christine Deffaix Rémy (2008) Programmation Shell sous Unix/Linux.
- [18] Christophe Blaess (2008) Shells Linux et Unix par la pratique
- [19] Carl Albing, JP Vossen, Cameron Newham (2007) Bash : Le livre de recettes
- [20] Christophe Blaess (2004) Scripts sous Linux Shell Bash, Sed, Awk, Perl, Tcl, Tk, Python, Ruby...

A

Annexe



```
Terminal
File Edit Options Buffers Tools Makefile Help
ll: msh

msh: commande.o variable.o externe.o interne.o prompt.o main.o
      gcc commande.o variable.o externe.o interne.o prompt.o main.o -o msh

commande.o: commande.c variable.h prompt.h externe.h interne.h
      gcc -c commande.c

variable.o: variable.c variable.h
      gcc -c variable.c

externe.o: externe.c commande.h
      gcc -c externe.c

interne.o: interne.c commande.h
      gcc -c interne.c

prompt.o: prompt.c commande.h
      gcc -c prompt.c

main.o: main.c commande.h variable.h prompt.h
      gcc -c main.c

clean:
      rm -rf *.o

mrproper: clean
      rm -rf msh

-UU-:---F1 makefile      All L1      (GNUmakefile) -----
For information about GNU Emacs and the GNU system, type C-h C-a.
```

Figure A.1 – *Makefile* utilisé pour compiler nos fichiers

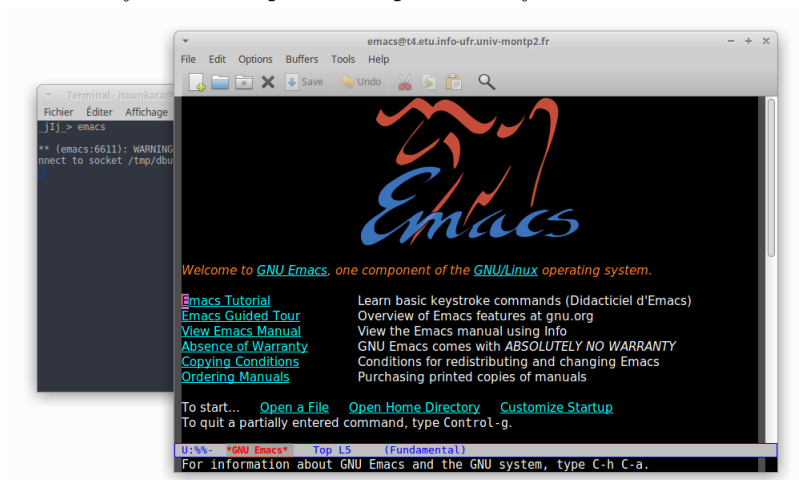
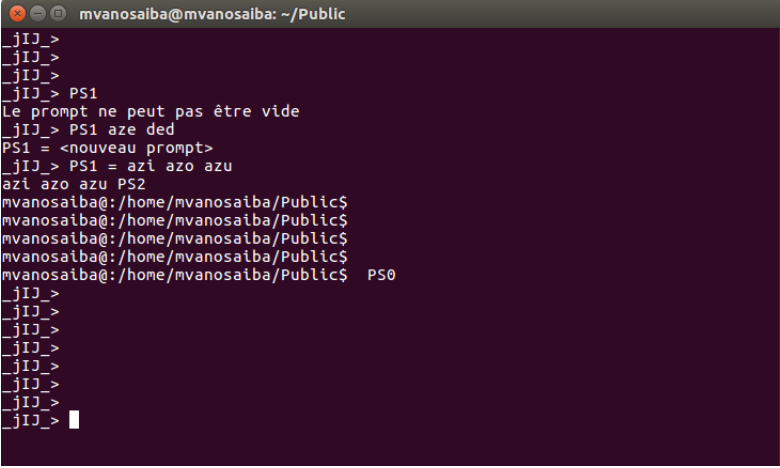
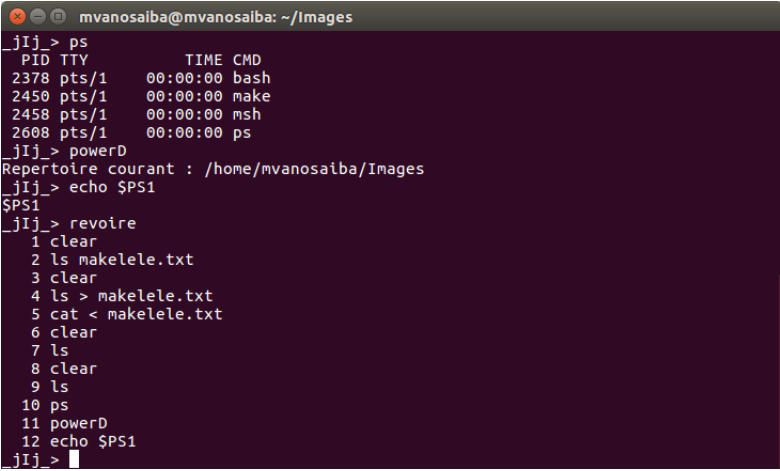


Figure A.2 – Lancement d'Emacs

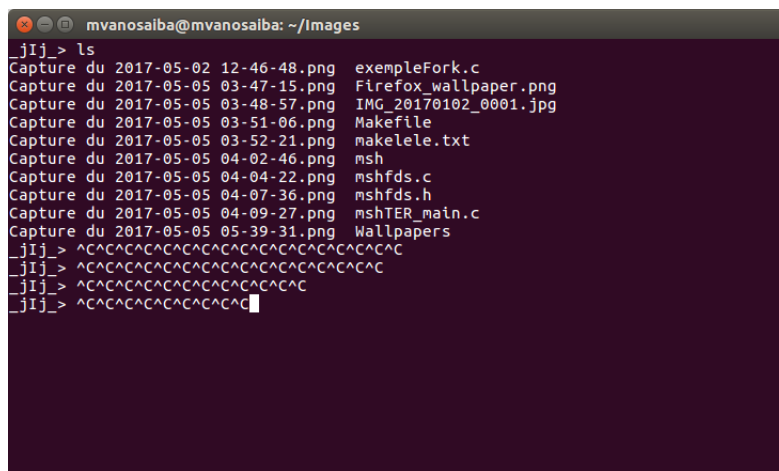
A terminal window titled 'mvanosaiba@mvanosaiba: ~/Public'. The user enters several commands to manipulate the shell prompt. The prompt starts as '_jIj_>'. After 'PS1', it changes to 'Le prompt ne peut pas être vide'. After 'PS1 aze ded', it changes to 'PS1 = <nouveau prompt>'. After 'PS1 = azi azo azu', it changes to 'azi azo azu PS2'. After 'PS0', it changes to 'PS0'. The user then enters several empty lines, each resulting in a new '_jIj_>' prompt.

```
mvanosaiba@mvanosaiba: ~/Public
_jIj_>
_jIj_>
_jIj_>
_jIj_> PS1
Le prompt ne peut pas être vide
_jIj_> PS1 aze ded
PS1 = <nouveau prompt>
_jIj_> PS1 = azi azo azu
azi azo azu PS2
mvanosaiba@:/home/mvanosaiba/Public$
mvanosaiba@:/home/mvanosaiba/Public$
mvanosaiba@:/home/mvanosaiba/Public$
mvanosaiba@:/home/mvanosaiba/Public$
mvanosaiba@:/home/mvanosaiba/Public$ PS0
_jIj_>
_jIj_>
_jIj_>
_jIj_>
_jIj_>
_jIj_>
_jIj_>
_jIj_>
```

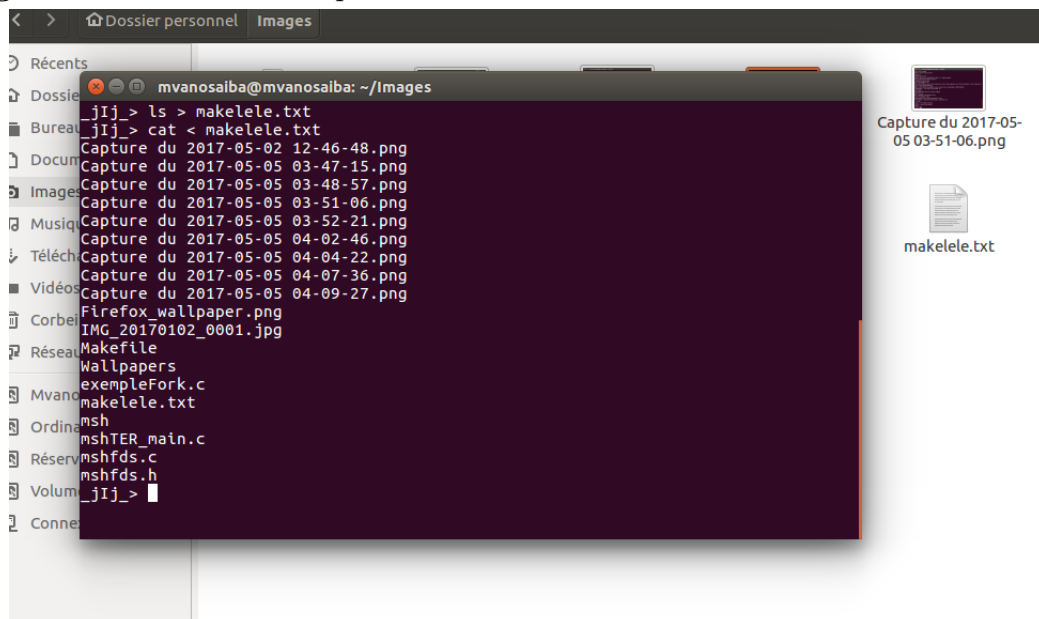
Figure A.3 – *Prompt*A terminal window titled 'mvanosaiba@mvanosaiba: ~/Images'. The user enters 'ps' to show running processes. Then 'powerD' to show the current directory. Then 'echo \$PS1' to show the prompt. Finally, 'revoir' to show a numbered list of commands entered in the session.

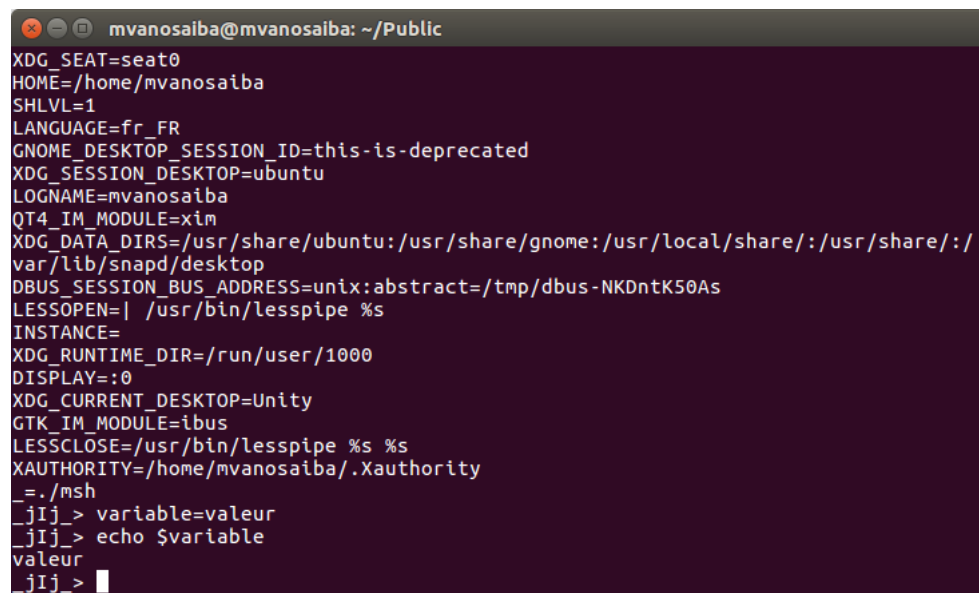
```
mvanosaiba@mvanosaiba: ~/Images
_jIj_> ps
PID TTY          TIME CMD
2378 pts/1        00:00:00 bash
2450 pts/1        00:00:00 make
2458 pts/1        00:00:00 msh
2608 pts/1        00:00:00 ps
_jIj_> powerD
Repertoire courant : /home/mvanosaiba/Images
_jIj_> echo $PS1
$PS1
_jIj_> revoir
1 clear
2 ls makelele.txt
3 clear
4 ls > makelele.txt
5 cat < makelele.txt
6 clear
7 ls
8 clear
9 ls
10 ps
11 powerD
12 echo $PS1
_jIj_>
```

Figure A.4 – *Historique*



```
mvanosaiba@mvanosaiba: ~/Images
_jIj_> ls
Capture du 2017-05-02 12-46-48.png  exempleFork.c
Capture du 2017-05-05 03-47-15.png  Firefox_wallpaper.png
Capture du 2017-05-05 03-48-57.png  IMG_20170102_0001.jpg
Capture du 2017-05-05 03-51-06.png  Makefile
Capture du 2017-05-05 03-52-21.png  makelele.txt
Capture du 2017-05-05 04-02-46.png  msh
Capture du 2017-05-05 04-04-22.png  mshfds.c
Capture du 2017-05-05 04-07-36.png  mshfds.h
Capture du 2017-05-05 04-09-27.png  mshTER_main.c
Capture du 2017-05-05 05-39-31.png  Wallpapers
_jIj_> ^C
_jIj_> ^C
_jIj_> ^C
_jIj_> ^C
```

Figure A.5 – *SIGINT bloqué*Figure A.6 – *Redirections*



```
mvanosaiba@mvanosaiba: ~/Public
XDG_SEAT=seat0
HOME=/home/mvanosaiba
SHLVL=1
LANGUAGE=fr_FR
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
XDG_SESSION_DESKTOP=ubuntu
LOGNAME=mvanosaiba
QT4_IM_MODULE=xim
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share:/usr/share:/var/lib/snapd/desktop
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-NKDntK50As
LESSOPEN=| /usr/bin/lesspipe %s
INSTANCE=
XDG_RUNTIME_DIR=/run/user/1000
DISPLAY=:0
XDG_CURRENT_DESKTOP=Unity
GTK_IM_MODULE=ibus
LESSCLOSE=/usr/bin/lesspipe %s %s
XAUTHORITY=/home/mvanosaiba/.Xauthority
_=./msh
_jIj_> variable=valeur
_jIj_> echo $variable
valeur
_jIj_>
```

Figure A.7 – *Gestions des variables*