Systèmes - Windows & Linux

OS - Shell 1

14 janvier 2022

Version 1

Fabrice BOISSIER <fabrice.boissier@epita.fr>

Copyright

Table des matières

1 Consignes Générales

Les informations suivantes sont très importantes :

Le non-respect d'une des consignes suivantes entraînera des sanctions pouvant aller jusqu'à la multiplication de la note finale par 0.

Ces consignes sont claires, non-ambiguës, et ont un objectif précis. En outre, elles ne sont pas négociables.

N'hésitez pas à demander si vous ne comprenez pas une des règles.

Consigne Générale 0 : Vous devez lire le sujet.
Consigne Générale 1 : Vous devez respecter les consignes.
Consigne Générale 2 : Vous devez rendre le travail dans les délais prévus.

Consigne Générale 3 : Le travail doit être rendu dans le format décrit à la section Format de Rendu.

Consigne Générale 4 : Le travail rendu ne doit pas contenir de fichiers binaires, temporaires, ou d'erreurs (`*~`, `*.o`, `*.a`, `*.so`, `*#*`, `*core`, `*.log`, `*.exe`, binaires, ...).

Consigne Générale 5 : Dans l'ensemble de ce document, la casse (caractères majuscules et minuscules) est très importante. Vous devez strictement respecter les majuscules et minuscules imposées dans les messages et noms de fichiers du sujet.

Consigne Générale 6 : Dans l'ensemble de ce document, **`nom1-nom2`** correspond à la combinaison des deux noms de votre binôme (par exemple pour Fabrice BOISSIER et Mark ANGOUSTURES, cela donnera **`boissier-angoustures`**).

Consigne Générale 7 : Dans l'ensemble de ce document, le caractère ␣ correspond à une espace (s'il vous est demandé d'afficher ␣␣␣, vous devez afficher trois espaces consécutives).

Consigne Générale 8 : Tout retard, même d'une seconde, entraîne la note non négociable de 0.

Consigne Générale 9 : La triche (échange de code, copie de code ou de texte, ...) entraîne **au mieux** la note non négociable de 0.

Consigne Générale 10 : En cas de problème avec le projet, vous devez contacter le plus tôt possible les responsables du sujet aux adresses mail indiquées.

Conseil : N'attendez pas la dernière minute pour commencer à travailler sur le sujet.

2 Format de Rendu

Responsable(s) du projet :	**Fabrice BOISSIER** `<fabrice.boissier@univ-paris1.fr>`
Balise(s) du projet :	**`[OS][TP1]`**
Nombre d'étudiant(s) par rendu :	1
Procédure de rendu :	Pas de rendu
Nom du répertoire :	`login.x-TP1`
Nom de l'archive :	`login.x-TP1.tar.bz2`
Date maximale de rendu :	Pas de rendu
Durée du projet :	Pas de rendu
Architecture/OS :	Linux - Ubuntu (x86_64)
Langage(s) :	sh
Compilateur/Interpréteur :	**`/bin/bash`**
Options du compilateur/interpréteur :	

L'arborescence attendue pour le projet est la suivante :

```
login.x-TP1/
login.x-TP1/AUTHORS
login.x-TP1/README
login.x-TP1/
login.x-TP1/src/
```

Les fichiers suivants sont requis :

`AUTHORS`	contient le(s) nom(s) et prénom(s) de(s) auteur(s).
`README`	contient la description du projet et des exercices, ainsi que la façon d'utiliser le projet.

3 Aide Mémoire

Le travail doit être rendu au format ***.tar.bz2***, c'est-à-dire une archive **bz2** compressée avec un outil adapté (voir **`man 1 tar`** et **`man 1 bz2`**).
Tout autre format d'archive (zip, rar, 7zip, gz, gzip, ...) ne sera pas pris en compte, et votre travail ne sera pas corrigé (entraînant la note de 0).

Dans ce sujet précis, vous ferez du code en script shell, qui affichera les résultats dans le terminal (et pourra donc être redirigé dans un fichier texte).

4 Cours

Commandes étudiées :	`sh`, `bash`, `man`, `ls`, `mkdir`, `touch`, `chmod`, `mv`, `rm`, `rmdir`, `cat`, `file`, `whereis`, `which`
Builtins étudiées :	`pwd`, `cd`, `exit`, `logout`, `echo`, `umask`, `type`, `>`, `>>`, `<`, `<<`, `|`
Notions étudiées :	Shell, Manuels, Fichiers, Répertoires, Droits, Redirections

4.1 Lancement et Manuels

Le shell est un programme permettant d'interpréter et d'exécuter des actions à partir de commandes écrites par l'utilisateur. Concrètement, dans le cas du shell d'un système d'exploitation, il s'agit d'un programme permettant de lancer des programmes selon ce que l'utilisateur demande. Les shells actuels permettent de chaîner des programmes et potentiellement faire d'autres actions (comme rediriger les messages d'erreur vers des fichiers, conditionner le lancement de certains programmes, etc).

Sur UNIX, il existe historiquement deux familles de shells :
- les dérivés du *bourne shell* (sh) : bash (Bourne again shell), ksh (Korn shell), zsh, ...
- les *C-shells* (en voie de disparition) : csh, tcsh

D'autres shells plus récents sont également apparus, mais nous ne les étudierons pas. Il faut principalement retenir que le langage reconnu par les *bourne shells* n'est absolument pas reconnu par les *C-shells*. Mais n'importe quel script écrit pour *sh* sera reconnu et interprété correctement par *bash*, *ksh*, *zsh*, ...
Il est important de comprendre la différence entre le **langage** que vous écrivez, et le **programme** qui le lit : une implémentation différente engendrera parfois des comportements différents pour une même instruction. Dit autrement, deux programmes différents peuvent reconnaître le même langage, mais ne pas effectuer strictement les mêmes opérations (par exemple, il existe deux façons d'implémenter la multiplication, mais leurs résultats restent les mêmes : pour $n \times m$, on peut faire n additions de m [$m+m+...+m$], ou m additions de n [$n+n+...+n$]).

On pourra noter que les entreprises utilisent le *Korn shell* étant donné que des implémentations payantes de celui-ci sont vendues et maintenues. Il existe néanmoins quelques implémentations gratuites et libres : *mksh*, *pdksh* (Public Domain ksh), ... On pourra noter que le *ksh* étant certifié POSIX, il est possible de le compiler puis l'exécuter sur n'importe quel système conforme à POSIX (donc Windows peut potentiellement lire et exécuter des scripts écrits pour *sh*).

Intéressons-nous maintenant aux commandes classiques sur les UNIX/Linux. Le tableau 1 montre les commandes habituellement utilisées dans le shell, et qui correspondent à ce que vous feriez avec une interface graphique.
Connectez-vous sur une machine, et lancer un terminal. Lorsque celui-ci est lancé, quelques lignes devraient s'afficher dedans, il s'agit du **prompt**.

Commande	Description
ls	Lister le contenu d'un répertoire
cp	Copier un fichier
mv	Déplacer un fichier
rm	Supprimer un fichier
ln	Créer un lien (dur ou symbolique) vers un fichier ou un répertoire
mkdir	Créer un répertoire
rmdir	Supprimer un répertoire
cd	Changer de répertoire courant
pwd	Afficher le chemin complet du répertoire courant

TABLE 1 – Commandes classiques sur UNIX/Linux

Il y a plusieurs prompts par défaut, voici une liste non exhaustive :

```
$
%
>
bash$>
```

L'utilité du prompt est de savoir si le shell vous offre la posibilité de taper des commandes, ou s'il est en train d'effectuer un traitement. Vous pouvez le personnaliser pour qu'il affiche des informations utiles (l'heure, la date, le dossier où vous vous trouvez, ...).

Pour démarrer, apprenez à vous servir de la commande `man` (pour manuel).

```
man man
```

Une documentation apparaitra, vous pouvez vous déplacer dedans avec les touches fléchées, et quitter en appuyant sur la touche 'q'. Pour bien utiliser les manuels, indiquez en argument la commande que vous souhaitez étudier. Par exemple, pour se documenter sur `ls`, tapez :

```
man ls
```

Pour se documenter sur la commande shell/builtin `printf`, il faut demander explicitement la section 1 du manuel.

```
man 1 printf
```

Pour se documenter sur la fonction C `printf`, il faut demander explicitement la section 3 du manuel.

```
man 3 printf
```

Dans le même cas, pour se documenter sur l'appel système (syscall) `read`, utilisez la

section 2 du manuel.

```
man 2 read
```

Par défaut, `man` essaye d'utiliser la section la plus petite pour l'argument qu'on lui donne (`read` dispose des sections 1 et 2, mais `lseek` ne dispose pas de section 1) :

```
man read
man lseek
```

Une autre commande est également utilisé dans le monde du GNU : `info`. Celle-ci donne des informations sur la version GNU des outils, et est beaucoup plus complète dans certains cas.

```
info ls
```

Pour quitter le shell et fermer sa session, utilisez les commandes `exit` ou `logout`.

4.2 Déplacement dans le Shell

Pour connaitre votre position dans l'arborescence, utilisez la commande `pwd`.

`pwd`

Le chemin absolu va s'afficher. Il s'agit du chemin depuis la racine de l'arborescence.

Pour connaitre les fichiers et dossiers présents là où vous vous trouvez, utilisez la commande `ls`.

`ls`

Pour connaitre les **droits** et d'autres propriétés de chaque élément, tout en listant un fichier par ligne, utilisez l'option `-l` (lettre L).

`ls -l`

Le paramètre **`-a`** affiche les fichiers et dossiers considérés comme cachés (c'est-à-dire commençant par un point **`.`**).
`le -l -a`

On peut combiner les paramètres **`-a`** et **`-l`** ainsi : **`ls -la`** pour pouvoir afficher l'ensemble des fichiers et dossiers contenus dans un répertoire, le tout un fichier par ligne, et avec le maximum d'informations possibles.

`ls -la`

Vous allez créer un dossier/répertoire nommé "MonDossier" en utilisant la commande `mkdir`.

`mkdir MonDossier`

Pour vous déplacer dans ce dossier, utilisez la commande/builtin `cd`.

`cd MonDossier`

Maintenant, observez le nouveau chemin absolu.

`pwd`

Deux dossiers sont toujours présents dans *tous* les dossiers, affichez-les en utilisant l'option `-a` de `ls`.

```
ls -a
```

Le dossier "." correspond au dossier courant, le dossier ".." correspond au dossier parent. Essayez donc ces deux commandes en testant le chemin absolu :

```
cd .
pwd
cd ..
pwd
```

Allez dans le dossier */bin* avec `cd`, et regardez le chemin absolu.

```
cd /bin
pwd
```

Essayez maintenant d'utiliser la commande/builtin `cd` sans argument, et regardez où vous vous trouvez.

```
cd
pwd
```

Il s'agit de votre **home directory**, celui qui vous est alloué pour vos fichiers et dossiers. Lorsque `cd` est utilisé sans argument, vous revenez automatiquement à votre dossier utilisateur/home directory.
Essayez maintenant d'utiliser la commande/builtin `cd` avec l'argument `-`.

```
cd -
pwd
```

Vous vous retrouvez dans le dossier *précédent*. Il ne s'agit pas du dossier parent, mais du dossier où vous étiez précédemment (si vous avez utilisez des chemins absolus, vous serez déplacés vers l'ancien répertoire).

Retournez dans le dossier "MonDossier" que vous avez créé dans votre home directory.

```
cd ~/MonDossier
```

Le ~ (tilde) correspond à votre dossier utilisateur/home directory.

Créez ici deux dossiers : "Dossier1" et "Dossier2".

```
mkdir Dossier1 Dossier2
cd Dossier1
pwd
```

Il est possible d'enchaîner les chemins :

```
cd ../Dossier2
pwd
cd /usr/bin/../../etc
pwd
```

4.3 Fichiers, Répertoires, et Droits

```
cd ~/MonDossier
```

Il est possible de créer des fichiers vides, s'ils étaient inexistants, grâce à la commande `touch`.

```
ls -la
touch MonFichier
ls -la
```

L'option `-l` de `ls` permet d'afficher les droits et d'autres informations. La commande `chmod` permet de modifier ces droits.

```
chmod 755 MonFichier
ls -la
chmod 640 MonFichier
ls -la
```

Pour rappel, les droits sous forme de nombres correspondent à : *Propriétaire-Groupe-Autres* sous forme de combinaison (1 droit d'*exécution*, 2 droit d'*écriture*, 4 droit de *lecture*). Le droit d'exécution permet aux dossiers d'être accessibles, et aux fichiers d'être lancés.

La commande `echo` permet d'afficher du texte.

```
echo "Coucou !"
```

Afin de remplir le fichier, nous allons utiliser un opérateur de redirection > pour transférer les données écrites par `echo` vers le fichier "MonFichier" (nous verrons plus tard les détails).

```
echo "Coucou !" > MonFichier
```

La commande `cat` permet d'afficher le contenu d'un fichier passé en paramètre (mais pas seulement...).

```
cat MonFichier
```

Testons les droits de nouveau.

```
chmod 000 MonFichier
cat MonFichier

chmod 400 MonFichier
echo "A" > MonFichier
```

```
cat MonFichier

chmod 200 MonFichier
echo "B" > MonFichier
cat MonFichier

chmod 640 MonFichier
echo "C" > MonFichier
cat MonFichier
```

L'option `-e` de `echo` permet d'interpréter certains caractères tels que `\n` ou encore `\t`.

```
echo -e "Ceci\nest\nun\ntest." > MonFichier
cat MonFichier
```

Vérifiez le droit d'exécution avec les tests suivants (attention aux guillemets " et '). Pour exécuter un fichier, il suffit de le préfixer de `./` comme dans l'exemple `./script.sh`.

```
echo -e '#! /bin/sh\n\necho "Ca marche"\n' > MonFichier

chmod 750 MonFichier
cat MonFichier
./MonFichier

chmod 640 MonFichier
cat MonFichier
./MonFichier

chmod 100 MonFichier
cat MonFichier
./MonFichier

chmod 750 MonFichier
```

La commande `umask` permet de modifier les droits par défaut lors de la création d'un fichier ou répertoire. `umask` prend en paramètre un nombre similaire à celui de `chmod`, mais celui-ci sera l'inverse des droits donnés. Précisément, il s'agit d'un masque filtrant les droits demandés : donner `777` à `umask` signifie que tous les droits seront filtrés (et interdits), donc que l'équivalent d'un `chmod 000` est appliqué. À l'inverse, un `umask 000` signifie que l'on ne filtre aucun droit, donc que les nouveaux fichiers peuvent être créés avec n'importe quels droits. De façon plus pragmatique, un `umask 022` semble correct, car il acceptera au maximum un `chmod 755` sur les fichiers créés.

```
umask
umask 777
touch TestMask1
ls -la
umask 022
```

```
touch TestMask2
ls -la
umask 077
touch TestMask3
ls -la
umask 022
```

Il est possible de déplacer un fichier ou un dossier avec la commande `mv`.

```
mv MonFichier ..
mv ../MonFichier .
```

Un autre effet de la commande `mv` est de renommer les fichiers.

```
mv MonFichier script.sh
```

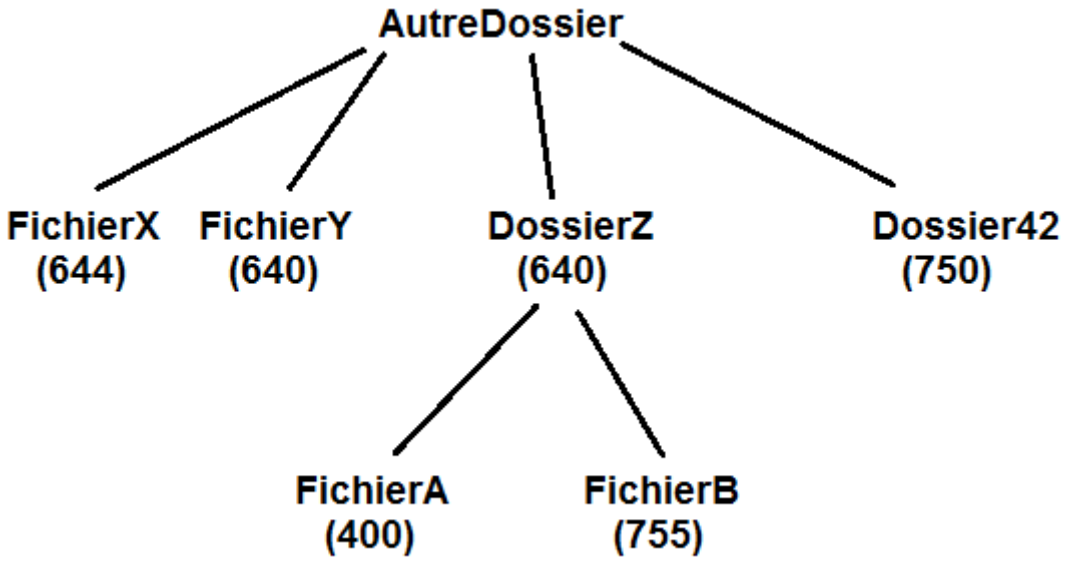
Pour supprimer un fichier, il faut utiliser `rm`. En ajoutant `-i`, on demandera toujours une confirmation avant la suppression. Inversement, pour forcer la suppression OU ignorer les erreurs, on peut utiliser l'option `-f`.

```
touch test1
rm -i test1
rm test1
rm -f test1
touch test1
rm -f test1
```

Pour supprimer un dossier, il faut utiliser `rmdir` si et seulement si le dossier est vide, ou alors utiliser `rm -f`.

```
mkdir dir1
rm dir1
rmdir dir1
mkdir dir1
touch dir1/file
rmdir dir1
rm -rf dir1
```

Entraînez-vous à créer cette arborescence avec ces droits :



Un équivalent de cette arborescence :

```
AutreDossier/
AutreDossier/FichierX
AutreDossier/FichierY
AutreDossier/DossierZ/
AutreDossier/DossierZ/FichierA
AutreDossier/DossierZ/FichierB
AutreDossier/Dossier42/
```

4.4 Types de Fichiers

Il est possible d'identifier le type de fichier en utilisant la commande `file`.

```
file .
file ../MonDossier
echo "Coucou !" > MonTexte
file MonTexte
file MonFichier
```

De la même manière, certaines commandes sont en réalités des sous-programmes/fonctions du shell et ne sont associées à aucun programme/binaire. Par exemple `ls` est un programme indépendant, mais `cd` est une builtin du shell. Pour identifier ces différences, plusieurs solutions existent : le manuel (voir si la manuel de la commande renvoie au manuel du shell/builtins), la commande `type` (builtin de bash), la commande `which`, la commande `whereis`.

```
man cd
type cd
which cd
whereis cd

man ls
type ls
which ls
whereis ls
```

Certaines commandes peuvent exister sous forme de binaire ET exister sous forme de builtin dans certains shells... dans ce cas, c'est la version shell qui primera (et il faudra appeler le programme avec son chemin long pour l'utiliser, par exemple : `/bin/ls`).

4.5 Redirections (et File Descriptors)

Dans le standard UNIX, pour transmettre des données en dehors d'un processus, on utilise des *file descriptors*. La philosophie UNIX se base sur un principe où *tout est fichier*, et donc, la lecture ou l'écriture se fait via les primitives (ou syscalls) de traitement de fichiers (open, read, write, close, lseek, ...). De plus, trois file descriptors sont constamment ouverts (sauf si leur manuel vous indique le contraire) : **STDIN** (0) Standard Input/l'entrée standard, **STDOUT** (1) Standard Output/la sortie standard, **STDERR** (2) Standard Error/la sortie d'erreur. Chacun permet de lire un périphérique d'entrée, ou d'écrire vers un flux de sortie dédié aux erreurs ou aux informations normales.

Un premier exemple de redirection a été vu plus haut.

```
echo "Coucou" > File1
cat File1
```

Ici, nous avons demandé à rediriger la sortie standard vers le fichier "*File1*". On remarque facilement que rien n'a été écrit dans le terminal, mais que tout a été mis dans le fichier. On peut étendre ce comportement à des commandes classiques :

```
ls -la > File1
cat File1
```

Inversement, lançez une commande qui échouera et écrira un message d'erreur (le fichier "*inex*" n'existant pas) :

```
ls -l . inex > File1
cat File1
```

Un message d'erreur pour le fichier inexistant "*inex*" s'affiche dans le terminal, et le fichier contient la sortie prévue pour `ls -l .`. En effet, par défaut, seule la sortie standard est redirigée avec l'opérateur > (chevron). On peut choisir quel file descriptor rediriger en le précisant avant la redirection.

```
ls -l . inex 1> FileOut
ls -l . inex 2> FileErr
cat FileOut
cat FileErr
```

Ici nous avons séparé les deux flux, il est possible de le faire en une seule commande :

```
ls -l . inex 1> FileOut 2> FileErr
```

Il est possible de rediriger le flux d'erreur vers le flux de sortie pour n'avoir qu'un seul flux à gérer. Cela s'effectue avec l'opérateur `2>&1` placé après la redirection.

```
ls -l . inex > File1 2>&1
cat File1
```

L'opérateur > a la particularité d'écraser le fichier vers lequel il écrit, il est cependant possible d'ajouter des données à la fin d'un fichier existant avec l'opérateur >>.

```
cat File1
echo "Ajout en Fin" >> File1
cat File1
```

Un autre opérateur complémentaire aux > et >> permet d'insérer des commandes dans l'entrée standard, il s'agit de l'opératuer <.

```
echo "Coucou !" > File1
cat < File1
```

Attention, les opérateurs chevrons >, >>, et < prennent **toujours** en paramètre un fichier, ils ne font qu'ouvrir en lecture ou écriture ce fichier pour transmettre le contenu.

Un quatrième opérateur appelé *heredoc* (Here Document) existe, il s'agit du <<. Celui-ci prend en paramètre non pas un fichier, mais une chaîne de caractère qui signifie l'arrêt de la lecture. Après avoir entré la commande, le shell va attendre que l'utilisateur tape des lignes, et il s'arrêtera lorsqu'il rencontrera la chaîne d'arrêt.

```
cat << FIN
allo allo
ceci est un test plus complexe
FIN
```

Plusieurs opérateurs de redirection peuvent être employés (l'entrée et la sortie sont redirigés).

```
cat << FIN > file1
1
2
FIN
cat file1


cat > file1 << FIN
Glop
Glop
FIN
cat file1
```

Si vous avez déjà travaillé avec les file descriptors en C, sachez qu'il est possible de rediriger d'autres valeurs que 0, 1, 2 en shell ! La commande suivante renvoie par exemple le

file descriptor 5 vers un fichier (il faut toutefois que le programme soit écrit pour envoyer des données vers ce file descriptor) :

```
./my_prog 5> filefd
```

4.6 Pipes (et Redirections)

Il est possible d'enchaîner des commandes et de transmettre la sortie de l'une vers l'entrée de la suivante. L'opérateur effectuant cette transmission est le | (pipe). Cet opérateur est extrêmement utile pour les traitements de textes formattés (extraits de bases de données, logs/journaux, ...).

```
echo -e "coucou\nallo\ntest" | grep "o" | more

echo -e "coucou\nallo\ntest" | grep "o" | grep "a" | more
```

Bien évidemment, la sortie de la dernière commande peut être redirigée :

```
echo -e "coucou\nallo\ntest" | grep "o" > file1
cat file1
```

L'entrée de la première commande peut elle aussi être détournée :

```
cat << FIN | grep "o" > file1
ouaf ouaf
beeeeeh
FIN
cat file1
```