Introduction aux systèmes d'exploitation (IS1)
TP n° 7 : les entrées / sorties

Télécharger depuis Moodle le script `tp7.sh`, puis l'exécuter depuis `~/Cours/2022/IS1`. Placez-vous dans le répertoire `TP7` qui vient d'être créé et répondez aux questions marquées par ✍ dans le fichier `reponses_TP7.txt`.

Redirections

Une commande `UNIX` est une « boîte noire » à laquelle on peut fournir, en plus de ses arguments, un fichier logique nommé *entrée standard* et qui renvoie deux types de réponses éventuelles sur des fichiers logiques appelés respectivement *sortie standard* et *sortie erreur standard*, comme illustré ci-dessous. On pourra aussi parler de *flots* (ou *flux*) d'entrée et de sortie.

Quelques exemples :

- « `cat` » écrit sur la sortie standard ce qu'elle reçoit sur l'entrée standard.
- « `date` » ne prend rien sur le flot d'entrée et écrit sur le flot de sortie.
- « `cd` » ne prend rien en entrée et ne renvoie rien en sortie.

Toutes ces commandes peuvent renvoyer des informations sur la sortie erreur standard pour signaler un problème quelconque (mauvaise utilisation de la commande, arguments inexistants, problèmes de droits, etc.).

Il est souvent nécessaire de sauver les données en entrée ou en sortie d'un processus dans des fichiers afin de pouvoir ensuite les réutiliser, les voir en totalité ou les traiter. Par défaut, les trois flots standard correspondent au terminal depuis lequel la commande est lancée : l'entrée standard est saisie au clavier et les sorties (standard et erreur) s'affichent à l'écran. Cependant, on peut changer la destination de ces flots à l'aide du mécanisme de *redirection*.

> À chaque flot d'entrée/sortie est associé un entier, appelé *descripteur* : `0` correspond à l'entrée standard, `1` à la sortie standard et `2` à la sortie erreur standard.

Redirection de la sortie standard : les opérateurs >, >| et >>

Lors des précédents TP, on a utilisé les opérateurs de redirection > et >> pour créer des fichiers avec la commande « `echo` » ; ce mécanisme de redirection peut en fait s'utiliser avec toutes les commandes.

Exercice 1 – *rediriger la sortie standard*

1. Dans un terminal, exécuter la commande « `set +o noclobber` ».[1]

2. Exécuter la commande « `date > fic` », puis afficher le contenu de `fic`. Exécuter ensuite « `cal > fic` » et comparer le contenu de `fic`.
   ✍ Quel est le rôle de > ?

3. Refaire les manipulations précédentes en utilisant l'opérateur `>|` à la place de l'opérateur >.
   ✍ Observez-vous une différence entre les opérateurs > et `>|` ?

4. Exécutez maintenant la commande « `set -o noclobber` ».
   ✍ Quelle est désormais la différence entre les opérateurs > et `>|` ?

5. Refaire les manipulations précédentes en utilisant maintenant l'opérateur >> plutôt que >.
   ✍ Que fait l'opérateur >> ? Est-il affecté par « `set -o noclobber` » ? Par « `set +o noclobber` » ?

Redirection de la sortie erreur standard : les opérateurs `2>` et `2>>`

En shell POSIX (`sh`, `bash`, `zsh`...), les opérateurs `2>`, `2>|` et `2>>` redirigent la **sortie erreur standard**.

Exercice 2 – *rediriger les messages d'erreur*

1. Dans `~/Cours/2021/IS1/TP7/Shadocks`, exécuter la commande « `cat gabuzomeu` » (nom qui n'est pas censé exister). Vous devez obtenir un message d'erreur. Refaire la même opération en redirigeant la sortie standard dans un (nouveau) fichier `gabuzo`. Que constatez-vous ?

2. Exécuter ensuite `ls gabuzomeu 2> gabuzo` et comparer. Recommencer en remplaçant `2>` par `2>>`. Que constatez-vous ?

3. Exécuter la commande « `meuzobuga` » (qui n'existe probablement pas plus que le fichier précédent) en redirigeant la sortie d'erreur vers un fichier `meuzobu`, puis afficher le contenu de ce fichier. La commande « `meuzobuga` » n'existe pas, pourtant le message d'erreur a été redirigé. Quel est le programme dont le message d'erreur a été redirigé ?

1. Cette commande désactive l'option `noclobber` ; sur les machines du SCRIPT, ce devrait déjà être le cas, et `set +o noclobber` ne fait initialement rien.

Redirection de l'entrée standard

L'opérateur < sert à rediriger le flot d'entrée. En exécutant *cmd* < *fic*, on passe à la commande « `cmd` » le **contenu** du fichier « `fic` » (et pas le fichier brut).

> « `tee` » sans argument, permet de **dupliquer** le flot de sortie, qui continue d'être affiché dans le terminal, mais est également recopié dans un ou plusieurs fichiers dont les références sont passées en paramètres.

Exercice 3 – *copies multiples simultanées*

Depuis le répertoire `SnowWhite`, exécuter la commande `tee Charmant < Prince`. Comparer les deux fichiers « `Prince` » et « `Charmant` ». Que remarque-t-on ?

✍ Quelle ligne de commande utilisant « `tee` » permet de créer simultanément plusieurs copies du fichier `Nain`, respectivement appelées `Atchoum`, `Dormeur`, `Grincheux`, `Joyeux`, `Prof`, `Simplet` et `Timide` ?

Syntaxe *shell*

Le *shell* est capable de manipuler des variables, tout comme les autres langages de programmation. Un certain nombre est défini à l'avance et définit le comportement du *shell* : ce sont les **variables d'environnement**. Parmi les exemples importants, citons « `LC_COLLATE` », déjà rencontrée, ou bien « `PATH` » ou « `LANG` ». Une liste des variables d'environnement définies est accessible en invoquant la commande « `env` ».

> Dans une commande, la chaîne « `$var` » est remplacée par le contenu de la variable « `var` ». On peut définir une variable, ou modifier sa valeur, en invoquant « `var=toto` » dans le terminal. Pour faire d'une variable une variable *d'environnement*, on rajoute « `export` » : « `export var=toto` ».

Exercice 4 – *variables d'environnement*

1. ✍ Quel est le résultat de la commande « `echo $HOME` » ?

2. ✍ Que contiennent les variables « `USER` », « `SHELL` », « `TERM` », « `PATH` », « `PWD` », « `OLDPWD` » ? Déplacez-vous dans l'arborescence des répertoires. Comment change la valeur de « `PWD` » et « `OLDPWD` » ?

3. Exécutez l'instruction *shell* « `PATH=` », qui donne une valeur nulle à la variable d'environnement « `PATH` ».
   ✍ Arrivez-vous à exécuter les commandes « `echo` » ? « `ls` » ? « `/bin/ls` » ?

4. **Fermez votre terminal et ouvrez-en un nouveau**.
   ✍ Que contient maintenant la variable « `PATH` » ?

5. Copiez le fichier « `/bin/ls` » à l'adresse « `~/bin/my-ls` », en créant le dossier « `~/bin` » au besoin. Exécutez l'instruction « `PATH="$PATH:~/bin"` ».
   ✍ Que se passe-t-il si vous tentez d'exécuter « `my-ls` » ?

Exercice 5 – *variables et chaînes*

Dans cet exercice, on investiguera les différents types de chaînes de caractères en *shell*.

1. ✍ Qu'affiche la commande « `echo "Plus ne suis ce que i'ay esté"` » ? Que se passe-t-il si vous enlevez les guillemets ?

2. ✍ Idem pour « `echo 'Et ne le sçaurois iamais estre'` » ?

3. Exécutez l'instruction « `SPRING=printemps` ».
   ✍ Qu'affiche la commande « `echo "Mon beau $SPRING & mon esté"` » ?

4. Exécutez l'instruction « `WINDOW=fenestre` ».
   ✍ Qu'affiche la commande « `echo 'Ont faict le sault par la $WINDOW'` » ? Remplacez les guillemets simples par des guillemets doubles. Qu'est-ce qui change ? Qu'en déduisez-vous sur les chaînes définies par « `'` » et « `"` » ?

5. ✍ Modifiez le contenu de la variable « `SPRING` » en lui donnant la valeur « `hiver` » et affichez sur votre terminal la phrase « J'ai vu l'hiver par la fenestre » sans utiliser les mots « `hiver` » et « `fenestre` ».

Fichiers spéciaux

`/dev/null` est un fichier spécial qui se comporte comme un « puits sans fond » : on peut écrire dedans tant qu'on le veut et les données sont alors perdues. Il peut par exemple servir à jeter la sortie erreur quand on n'en a pas besoin.

Exercice 6 – *Le fichier `/dev/null`*

1. Afficher le contenu de tous les fichiers appartenant à un sous-répertoire du répertoire `LesCowboysFringants`, et dont le nom contient au moins une majuscule.
   ✍ Faire en sorte que les messages d'erreur ne s'affichent pas pour obtenir un affichage lisible.

2. ✍ Donner une ligne de commande qui utilise les messages d'erreur de « `ls` » pour déterminer la liste des répertoires non accessibles dans l'arborescence de racine `LesCowboysFringants` (sans donc afficher les fichiers et les répertoires accessibles).

Sous `UNIX`, les périphériques sont considérés comme des fichiers spéciaux, accessibles par des liens situés dans la sous-arborescence `/dev` du système de fichiers. Il existe deux types de tels fichiers spéciaux : « caractère » `'c'` (terminaux etc.) ou « bloc » `'b'` (disques etc.). Ces termes désignent la manière de traiter les lectures et les écritures : soit caractère par caractère, soit par blocs.

Exercice 7 – *Le terminal, un fichier très spécial...*

1. La commande « `tty` » renvoie la référence absolue du fichier spécial correspondant au terminal dans lequel elle est exécutée. Afficher les caractéristiques (droits, etc.) de ce fichier. ✍ quel est son type : bloc ou caractère ?

2. ✍ Dans un autre terminal, exécuter la commande `date` en redirigeant sa sortie standard vers le fichier spécial correspondant au premier terminal. Que se passe-t-il ?

Quelques commandes manipulant l'entrée standard

Les commandes qui manipulent l'entrée standard admettent en général un argument facultatif qui est un nom de fichier sur le contenu duquel elles peuvent s'exécuter. Il peut néanmoins être intéressant de les faire travailler directement sur l'entrée standard.

« `cat` » sans argument, recopie sur la sortie ce qui arrive dans le flot d'entrée.

Exercice 8 – *la commande « `cat` »*

1. Lancer « `cat` » sans argument, mais avec l'option « `-n` ». La ligne reste vide, le shell attend que vous lui fournissiez des données. Taper « `Arrête de répéter tout ce que je dis!` » suivis de la touche entrée et observer le résultat.
   Taper encore quelques lignes, puis presser la combinaison de touches ctrl-D au début d'une ligne vide. Cela indique au processus la fin des données à traiter.

2. Chercher dans la page `man` ce que fait l'option « `-s` », et tester son effet ; pour mieux le visualiser, il pourra être utile de la combiner avec l'option « `-n` ».
   ✍ Créer ensuite un fichier `Reine`, lisible sans devoir utiliser la barre de défilement, à partir du fichier `Sorciere`.

« `head` » **et** « `tail` » lisent sur leur entrée standard et conservent les premières lignes (pour « `head` ») et dernières (pour « `tail` »). Le nombre de lignes conservées est 10 par défaut, ou l'entier passé en argument après l'option « `-n` ».

Exercice 9 – *les commandes « `head` » et « `tail` »*

1. Dans `~/Cours/2021/IS1/TP7/Brassens`, tester les commandes « `head` » et « `tail` » en redirigeant leur entrée standard sur le fichier « `BancsPublics` ». ✍ Afficher ensuite seulement ses 6 premières lignes.

2. Lancer la commande « `tail -n 3` » (ou « `tail -3` ») en redirigeant la sortie sur un autre terminal. Comme dans l'exercice précédent, le shell attend que vous fournissiez des données, que la commande traitera ligne par ligne. Taper cinq lignes de texte, terminées par ctrl-D. ✍ Qu'affiche votre commande ? Quand ? Pourquoi ?

3. Faire de même avec la commande « `head -3` ». ✍ Expliquer la différence de comportement.

4. ✍ À quoi sert l'option « `-c` » de ces commandes ? La tester.

5. Pour la commande « `tail` », le nombre passé avec les options « `-n` » ou « `-c` » peut être précédé d'un signe « + », pour indiquer qu'il est compté à partir du début et non à partir de la fin. Tester « `tail -n +2` » et taper cinq lignes sur le flot d'entrée, pour vérifier que vous avez bien compris. ✍ Faire de même en lui passant comme paramètre le fichier « `BancsPublics` ». Qu'obtient-on ?

6. Lancer la commande « `head` » en lui donnant les deux paramètres « `BancsPublics` » et « `Fernande` ». Que peut-on remarquer ? ✍ Déterminer quelle option de « `head` » (qui fonctionne également avec « `tail` ») permet de supprimer l'affichage des noms.

7. ✍ À partir du répertoire `~/Cours/2021/IS1/TP7/` créer un fichier `LeGorille` à partir des 10 premières lignes de chaque fichier contenu dans le répertoire « `Brassens` ».
   De façon similaire, créer un fichier « `LesPassantes` » en utilisant les 6 dernières lignes de chaque fichier.