Carnet de Travaux Pratiques
Système et programmation système
Licence 2 Informatique

Julien Bernard Eric Merlet

Introduction

Ces travaux pratiques vous permettent de vous entraîner à manipuler toutes les notions vues en cours. Si vous ne parvenez pas à finir le TP dans le temps encadré imparti, il sera obligatoire de le finir pour la semaine suivante.

Tout le travail demandé ne nécessite qu'un éditeur de texte (comme `Kate`) et un terminal pour le shell. En particulier, il est expressément déconseillé d'utiliser un gestionnaire de fichier graphique (comme `Nautilus`).

La langue de la programmation est l'anglais. Vous devrez donc écrire tout votre code (commentaires inclus) en utilisant l'anglais. Cela concerne aussi bien les noms (variables, fonctions, etc) dans votre code que l'affichage.

Table des matières

Travaux Pratiques de Système n°1

Exercice 1 : Prise en main de l'environnement

Cet exercice doit vous permettre de prendre en main votre environnement de travail. Pour cela, connectez-vous d'abord sur votre compte avec votre nom d'utilisateur.

→ Accès à un terminal physique ou textuel

Tapez CTRL+ALT+F3 ou CTRL+ALT+F4 ou CTRL+ALT+F5 ou CTRL+ALT+F6. Entrez votre nom d'utilisateur et votre mot de passe.

Question 1.1 Dans quel répertoire vous trouvez-vous ?

Question 1.2 À l'aide de la commande `ls(1)`, afficher le contenu de votre répertoire. Combien de fichiers et répertoires cachés avez-vous ?

Question 1.3 Effacer l'écran à l'aide de la commande `clear` ou avec la combinaison de touches CTRL+L.

Question 1.4 Fermer la session à l'aide de la commande `exit` ou avec la combinaison de touche CTRL+D.

Pour revenir à l'interface graphique, tapez CTRL+ALT+F2.

→ Commandes de base

Question 1.5 Ouvrez un terminal virtuel ou graphique. Tester les commandes vues en cours : `whoami(1)`, `uname(1)`, `uptime(1)`, `date(1)`, `cal(1)`, `echo(1)`, `man(1)`, `whatis(1)`, `apropos(1)`. En particulier, en cas d'options multiples, vous pouvez utiliser deux écritures : `-a -b -c` ou `-abc`. Le vérifier à l'aide de la commande `uname` par exemple.

Question 1.6 Comment obtenir une commande équivalente à `whoami` avec la commande `id` ?

Question 1.7 Dans quels sections pouvez-vous trouver une manpage appelée `time` ?

→ Système de fichier

Pour créer un fichier régulier, il existe la commande `touch(1)`. En fait, cette commande met à jour le `atime` et le `mtime` d'un fichier (en le "touchant"), mais si le fichier n'existe pas, il est créé. Vous utiliserez cette commande pour créer des fichiers réguliers vides. Pour rappel, la commande pour créer un répertoire est `mkdir(1)`.

En outre, pour les commandes `cp(1)`, `rm(1)`, `mv(1)`, vous testerez l'option `-i` qui permet de demander une confirmation.

Question 1.8 Créer un répertoire `SYS` dans votre répertoire personnel. Entrer dans ce répertoire puis créer un répertoire `exemple`. Entrer dans le répertoire `exemple`.

Dans la suite de cet exercice, vous pouvez utiliser l'option `-R` de la commande `ls(1)` pour visualiser le contenu d'un répertoire et de ses sous-répertoires récursivement.

Question 1.9 Dans le répertoire `exemple` que vous venez de créer, créer les répertoires `skywalker/luke` en une seule commande. Puis créer un répertoire `skywalker/anakin` et un fichier régulier `skywalker/anakin/README`.

Question 1.10 Supprimer le répertoire `skywalker/luke`. Que se passe-t-il si vous essayez de supprimer `skywalker/anakin` avec la commande `rmdir(1)` ?

Question 1.11 Déplacer le répertoire `skywalker/anakin` pour le positionner dans le répertoire `exemple`, tout en le renommant `darth_vader`.

Question 1.12 Créer un répertoire `yoda` et un fichier régulier `yoda/README`. Copier le fichier `yoda/README` dans le fichier `yoda/README.old`. Créer un lien physique appelé `yoda/README2` sur l'inode du fichier `yoda/README`. Ecrire la chaîne de caractères "bonjour" dans `yoda/README`. Afficher le contenu de `yoda/README2`. Visualiser les numéros d'inode et le nombre de liens physiques (pointant sur les inodes) des fichiers `yoda/README` et `yoda/README2`.

Question 1.13 Créer un répertoire `.private` dans `yoda`. Créer, dans `.private`, un lien symbolique nommé `LREADME` sur le fichier `yoda/README`. Afficher le contenu de `yoda/.private/LREADME`.

Question 1.14 Supprimer le lien physique `yoda/README2`. Un fichier sur disque a-il été supprimé ? Supprimer `yoda/.private/LREADME`. Un fichier sur disque a-il été supprimé ? Supprimer le lien physique `yoda/README`. Un fichier sur disque a-il été supprimé ?

Question 1.15 Supprimer récursivement le répertoire `yoda` (c'est-à-dire le répertoire et tout ce qu'il contient) à l'aide de l'option `-r` de `rm(1)`.

Continuez à créer des fichiers et des répertoires et à les déplacer, copier, supprimer.

Exercice 2 : Archives

Cet exercice consiste à manipuler des archives (compressées ou non) à l'aide de la commande `tar(1)` (*Tape ARchive*). À l'origine, la commande `tar(1)` servait à créer des sauvegardes sur des bandes magnétiques (*tape*). Les bandes magnétiques avaient une plus grosse capacité de stockage que les disques durs mais avait un accès linéaire (c'est-à-dire que le temps pour accéder à une donnée était fonction de sa position sur la bande). La commande `tar(1)` a évolué pour gérer des archives de fichiers.

La commande `tar(1)` prend en option :
— Une option de compression parmi :
  — aucune option si on ne veut pas de compression
  — `z` pour compresser avec `gzip`
  — `j` pour compresser avec `bzip2`
— Une action parmi :
  — `c` pour créer une archive avec des fichiers
  — `x` pour extraire les fichiers d'une archive
  — `t` pour lister le contenu d'une archive
  — `r` pour ajouter des fichiers dans une archive
  — `u` pour mettre à jour des fichiers dans une archive
— L'option `f` qui indique qu'on utilise un fichier dont le nom est indiqué
— L'option `v` (non-obligatoire) si vous voulez afficher le déroulement des opérations

Par exemple :

```
tar cf archive.tar repertoire
tar cf archive.tar fichier1 fichier2
```

Question 2.1 Créez une archive `exemple.tar` avec le répertoire `exemple` de l'exercice de prise en main.

Question 2.2 Allez chercher le fichier des "figures du cours" sur MOODLE. Quels sont les fichiers contenus dans cette archive ?

Question 2.3 Extrayez l'archive puis créez une archive `inodes.tar.bz2` avec les quatre figures concernant les inodes.

Question 2.4 Les options `z` et `j` sont en fait équivalentes à appeler directement `gzip(1)` et `bzip2(1)` (pour la compression) ou `gunzip(1)` et `bunzip2(1)` (pour la décompression). Décompressez l'archive `inodes.tar.bz2` sans en extraire les fichiers. Vous obtenez l'archive `inodes.tar`.

Question 2.5 Ajoutez les figures concernant la compilation à l'archive `inodes.tar` obtenue à la question précédente.

→ Bonnes pratiques

Il existe quelques bonnes pratiques pour la gestion des archives :
— On n'archive jamais un fichier seul : ça n'a aucun sens !
— On place toujours les fichiers à archiver dans un répertoire. Ainsi, lors de l'extraction de l'archive, seul ce répertoire est ajouté dans le répertoire courant (les fichiers archivés ne sont pas dispersés dans le répertoire courant).

Exercice 3 : Chaînes de caractères et développement de la ligne de commande

L'exécutable `prog` est le résultat de la compilation du programme source `prog.c` ci-dessous :

```
int main(int argc, char *argv[]){
  printf("argc = %d\n", argc); // printf() écrit dans stdout
  fprintf(stderr, "Arguments received :\n");
  for(int i = 0; i < argc; ++i){
    fprintf(stderr,"argv%d : %s\n", i, argv[i]);
  }
  return 0;
}
```

L'exécutable `prog` est placé dans un répertoire contenant l'arborescence indiquée ci-dessous :

```
$ ls -al
total 32
drwxr-xr-x 3 eric eric  4096 févr.  3 09:51 .
drwxr-xr-x 5 eric eric  4096 févr.  3 09:51 ..
-rw-r--r-- 1 eric eric     0 janv. 21  2019 .cache
-rwxrwxr-x 1 eric eric 16072 févr.  3 09:36 prog
-rw-r--r-- 1 eric eric   231 janv. 21  2019 prog.c
drwxr-xr-x 2 eric eric  4096 janv. 21  2019 rep
$ ls -al rep
total 8
drwxr-xr-x 2 eric eric 4096 janv. 21  2019 .
drwxr-xr-x 3 eric eric 4096 févr.  3 09:51 ..
-rw-r--r-- 1 eric eric    0 janv. 21  2019 .rcache
-rw-r--r-- 1 eric eric    0 janv. 21  2019 rfile
```

Question 3.1 Tester et justifier le résultat des commandes ci-dessous :

```
$ A='one two'
$ B='ls'
$ ./prog '$A' $NOEXIST "$A" > files1 2> files2
$ ./prog > files1 2> files2 ${A}toto \'$A "$NOEXIST"
$ ./prog ${#A} ${A#* } "${A#t* }"
$ ./prog $($B)
$ ./prog "$($B)"
$ ./prog *
$ ./prog .*
$ ./prog rep/*
$ ./prog rep/.*
$ ./prog files?
$ ./prog "files?"
$ ./prog $($B files?)
$ ./prog "$($B files?)"
$ ./prog "$($B "files?")"
$ ./prog '$($B "files?")'
```

Exercice 4 : Somme des longueurs des arguments d'un script

Le but est de faire un script `arguments.sh` qui détermine et affiche :
— le nombre d'arguments d'un script
— la somme des longueurs des arguments d'un script

Indices : `$#`, `$*`, `$@`, `${#VAR}`, `shift`

Question 4.1 Ecrire un script qui détermine et affiche le nombre d'arguments transmis.

Question 4.2 Compléter le script pour calculer et afficher la somme des longueurs des arguments.

Exemple d'exécution :

```
$ ./arguments.sh aa bbb "" "cc  12"
Number of arguments = 4
Sum of the lengths of the arguments = 11
$
```

Exercice 5 : Nombre de fichiers du répertoire courant

Le but est de faire, sans utiliser la commande `ls`, un script `files.sh` qui détermine et affiche :
— le nombre de fichiers non cachés du répertoire courant
— le nombre de fichiers cachés du répertoire courant

Question 5.1 Ecrire un script qui détermine et affiche le nombre de fichiers non cachés du répertoire courant.

Question 5.2 Compléter le script pour déterminer et afficher le nombre de fichiers cachés du répertoire courant.

Exemple d'exécution :

```
$ ls -a
.  ..  arguments.sh  .bashrc  fic1  fichiers2.sh  fichiers.sh  toto
$ ./files.sh
Number of not hidden files in the current directory = 5
Number of hidden files in the current directory = 3
$
```

Travaux Pratiques de Système n°2

Exercice 6 : Jeu du «Plus petit / Plus grand»

Le but est de faire un script `more_less.sh` pour jouer au jeu «Plus petit / Plus grand». Ce jeu consiste à deviner un nombre entre 1 et 100 choisi au hasard par l'ordinateur. Pour cela, le joueur peut faire plusieurs propositions et l'ordinateur dit si la valeur qu'il a choisie est plus petite ou plus grande que la proposition du joueur.

Voici un exemple d'utilisation du jeu :

```
The computer has chosen a value between 1 and 100.
What do you propose ? 50
The value is smaller.
What do you propose ? 20
The value is greater.
What do you propose ? 35
Win ! You have found in 3 attempts
```

Question 6.1 Définir une variable appelée `MAX` avec la valeur 100. Faire tirer à l'ordinateur une valeur au hasard entre 1 et `MAX` en utilisant la commande :
`$ hexdump -n 2 -e '/2 "%u"' /dev/urandom`
Cette commande lit 2 octets (`-n 2`) dans le fichier virtuel `/dev/urandom` et affiche sur la sortie standard le résultat de la conversion binaire vers décimale des 2 octets lus (`-e '/2 "%u"'`).

Question 6.2 Demander à l'utilisateur sa proposition tant qu'il n'a pas trouvé la bonne réponse et afficher le résultat par rapport à la valeur de l'ordinateur.
Indice : `test(1)`
Remarque : `man test` permet d'accéder à la man page de la commande externe `test`. La documentation de la commande interne `test` se trouve dans la man page de `dash(1)`.

Question 6.3 Afficher le nombre d'essais nécessaires pour trouver la réponse.

On veut limiter le nombre d'essais possibles pour le joueur à 6. Si le joueur échoue 6 fois de suite, l'ordinateur affiche un message au joueur. Par exemple :

```
Lost ! The secret number was: 32
```

Question 6.4 Modifier le programme pour s'arrêter si le joueur fait 6 propositions perdantes et afficher un message adéquat.

Exercice 7 : Fichier de notes

Un enseignant garde les notes de ses étudiants dans un fichier `notes.txt` avec le format suivant :

```
nomprenom:note1:note2
```

où :
— le champ `nomprenom` contient un nombre quelconque de lettres, de caractères espace, de tirets et de simples quotes, contient au minimum 2 lettres, et doit commencer et se terminer par une lettre ;
— les notes sont des nombres **entiers** compris entre 0 et 20.

Pour vos tests, vous pouvez utiliser le fichier `notes.txt` disponible sur Moodle.

Question 7.1 À partir du fichier `notes.txt`, écrire, dans un script, nommé `notes1.sh`, en n'utilisant qu'une seule ligne de commande pour chaque question, les commandes permettant de connaître :

1. La liste des étudiants dans l'ordre alphabétique.
2. La liste des étudiants ayant eu une première note supérieure ou égale à 10.
3. La liste des deuxièmes notes avec pour chaque note le nombre d'étudiants l'ayant obtenu.

Question 7.2 L'objectif de cette question est d'écrire un script shell, nommé `notes2.sh`, qui prend en argument un fichier `notes.txt` avec le format décrit ci-dessus, et qui affiche sur sa sortie standard :
— pour chaque étudiant, la valeur de son champ `nomprenom` et la moyenne de ses deux notes (nombre éventuellement [1] à virgule), séparées par un caractère `":"` ;
— la moyenne de toutes les premières notes et celle de toutes les deuxièmes notes (avec 2 décimales après la virgule).

Définir deux fonctions prenant en paramètre le chemin vers le fichier `notes.txt`, qui font "tout" le travail, une sans utiliser la commande `awk(1)`, l'autre qui l'utilise. Ces deux fonctions doivent produire exactement le même résultat sur la sortie standard, aux problèmes d'arrondi près pour les moyennes de toutes les premières notes, et de toutes les deuxièmes notes.

Avant d'appeler ces fonctions, il faut vérifier que :

1. le script est appelé avec un seul argument, et que l'argument transmis correspond à un fichier existant avec la permission de lecture. Dans le cas contraire, il faut afficher un message sur la bonne sortie et terminer le script.
2. chaque ligne du fichier respecte le format décrit ci-dessus. Idem, dans le cas contraire, il faut afficher un message sur la bonne sortie et terminer le script.

1. par exemple, une moyenne égale à 17.0 est égale à 17, on peut donc supprimer le ".0" à la fin

Indication : pour la version sans la commande `awk(1)`, vous pouvez utiliser la commande `bc(1)` qui permet de faire des calculs avec des nombres entiers ou décimaux. En interne, la commande `bc(1)` représente tous les nombres en décimal, et tous les calculs sont faits en décimal. La commande `bc(1)` tronque les résultats des divisions et des multiplications en utilisant la valeur de l'attribut `scale` qui fixe le nombre de chiffres après le point décimal du résultat. Dans l'exemple ci-dessous, la commande `bc(1)` calcule `5/3` et tronque le résultat en utilisant 4 chiffres après le point décimal :

```
$ echo 'scale=4;5/3' | bc
1.6666
```

Exercice 8 : Lister les fichiers d'un répertoire en shell

Le but est de faire un script `ls.sh` qui est une version très simple de `ls(1)` en shell, ... sans utliser la commande `ls(1)`. Le script prendra éventuellement un seul argument, le chemin du répertoire à lister. En absence d'argument, le script liste le répertoire courant. Dans tous les autres cas, les arguments transmis au script sont non valides.
`Usage : ./ls.sh [DIR]`

Question 8.1 Vérifier la validité des arguments. En cas d'erreur, afficher, sur la sortie d'erreur standard, un message indiquant l'origine de l'erreur et l'usage du script (synopsis), puis sortir.

Question 8.2 Déterminer le répertoire à lister.

Question 8.3 Parcourir tous les fichiers du répertoire à lister et afficher leur nom.

Question 8.4 Déterminer si c'est un fichier régulier, un lien symbolique ou un répertoire, et afficher cette information. Indice : `test(1)`

Exemples d'exécution :

```
$ ./ls.sh
Directory : complexe
Directory : cor
Regular file : fic
Symbolic link : lfic
Directory : rep1
$
$ ./ls.sh rep1/rep12/
Regular file : fic121
$
$ ./ls.sh fic
Error : fic doesn't exist or isn't a directory
Usage : ./ls.sh [DIR]
$
$ ./ls.sh rep1 rep2
Error : The number of arguments isn't correct
Usage : ./ls.sh [DIR]
$
```

Travaux Pratiques de Système n°3

Exercice 9 : Recherche des IP attribuées sur un réseau local

→ Version 1

Le but est de faire un script `ping.sh` qui détermine les adresses IPv4 attribuées dans une plage d'adresses.
Les adresses IPv4 sont codées sur 32 bits. Elles sont usuellement représentées en utilisant une notation décimale à points où chaque nombre décimal correspond à un octet.
Exemple : `172.20.128.15`
Les adresses IPv4 sont hierarchiques :

- la partie de l'adresse située à gauche désigne le réseau (*network*) auquel elle appartient
- la partie située à droite désigne l'hôte (*host*) sur le réseau

Dans nos salles de TP, la partie "réseau" de l'adresse est codée sur les 3 premiers octets (octets de poids fort) et la partie "hôte" sur un seul octet. Ainsi, dans l'exemple précédent, la partie "réseau" de l'adresse est égale à `172.20.128`, et la partie "hôte" à `15`.

Question 9.1 Déterminer l'IPv4 de votre machine en utilisant la commande `ip(1)` avec les arguments `address` et `show` :
`$ ip address show`
En déduire ses parties "réseau" et "hôte".

La commande `ping(8)` permet, grâce à l'envoi de paquets, de vérifier si une machine distante répond et, par extension, si elle est accessible via le réseau.

Question 9.2 Essayer de "pinguer" votre propre adresse IPv4, ainsi que celle de votre voisin. (utiliser `CTRL^C` pour tuer le processus qui exécute la commande `ping(8)`).

Question 9.3 Déterminer les options de la commande `ping(8)` qui permettent d'envoyer un seul paquet avec une deadline de une seconde (si l'hôte distant ne répond pas).

Le synopsis du script `ping.sh` est le suivant :
`./ping.sh NETWORK_PART HOST_PART_START HOST_PART_STOP`

Exemple : `./ping.sh 172.20.128 10 50`
Dans le cas précédent, le script doit tester si les IP comprises entre 172.20.128.10 et 172.20.128.50 (bornes incluses) répondent.
Les arguments `HOST_PART_START` (2ème argument) et `HOST_PART_STOP` (3ème argument) doivent être des entiers compris dans l'intervalle [1, 254] et avec `HOST_PART_STOP` $\geq$ `HOST_PART_START`

Quelques exemples d'exécution :

```
$ ./ping.sh 172.20.128 170
Error : the number of arguments isn't correct
Usage : ./ping.sh NETWORK_PART HOST_PART_START HOST_PART_STOP
$
$ ./ping.sh 172.20.128 170 168
Error : HOST_PART_START and HOST_PART_STOP must be integers in [1, 254]
        with HOST_PART_START <= HOST_PART_STOP
Usage : ./ping.sh NETWORK_PART HOST_PART_START HOST_PART_STOP
$
$ ./ping.sh 172.20.128 170 180p
Error : HOST_PART_START and HOST_PART_STOP must be integers in [1, 254]
        with HOST_PART_START <= HOST_PART_STOP
Usage : ./ping.sh NETWORK_PART HOST_PART_START HOST_PART_STOP
$
$ ./ping.sh 172.20.128 170 180
Assigned IP addresses on the network 172.20.128.0 in the interval [.170, .180] :
172.20.128.170
172.20.128.178
--> Number of assigned IP addresses : 2
$
```

Question 9.4 Vérifier la présence et la validité des arguments. En cas d'erreur, afficher, sur la sortie d'erreur standard, un message indiquant l'origine de l'erreur et l'usage du script (synopsis), puis sortir.

Question 9.5 Compléter le script pour déterminer les IPs attribuées, ainsi que leur nombre.

Question 9.6 Sauvegarder votre script sous le nom `pingv1.sh`.

→ backréférence - référence arrière

Les expressions rationnelles disposent d'un mécanisme appelé backreference (ou référence arrière) qui permet de rechercher des sous-chaînes déjà trouvées. Un des exemples typiques du fonctionnement des références arrières consiste à rechercher si une chaîne contient le même mot répété 2 fois.
Par exemple, l'expression `"[a-z][a-z]"` correspond à deux lettres minuscules quelconques. Si on veut rechercher des chaines contenant deux lettres identiques adjacentes, on ne peut pas utiliser ce motif `"[a-z][a-z]"`. On a besoin d'un mécanisme permettant de :

1. *capturer* (ou mémoriser) une correspondance avec `"[a-z]"` ;
2. *référencer* la correspondance capturée.

Dans une expression rationnelle étendue :

— on a vu en cours que les parenthèses `(` et `)` permettent de repérer un sous-motif, mais elles permettent également de *capturer* une correspondance avec le sous-motif ;

— ensuite, un "\" suivi d'un chiffre `i` permet de référencer la `ième` correspondance capturée

Exemple 1 : l'expression régulière étendue qui correspondrait à deux lettres identiques adjacentes serait :
`([a-z])\1`.

Vous pouvez avoir jusqu'à neuf correspondances capturées : chaque occurrence de "(" commence une nouvelle capture de correspondance.

Exemple 2 : l'expression régulière étendue qui correspondrait à un palindrome de 5 lettres (par exemple "radar") serait :
`([a-z])([a-z])[a-z]\2\1`

On peut également référencer une correspondance capturée dans la chaine de remplacement de la commande `s` de `sed`.

Exemple 3 :
`sed -E 's/"([^"]*)"/\1/'` permet de remplacer, sur chaque ligne, la première suite de caractères entourée par des guillemets, par la même suite de caractères sans les guillemets.
`sed -E 's/"([^"]*)"/\1/g'` : avec l'indicateur `g`, toutes les occurrences (de suites de caractères entourées par des guillemets) sont traitées.

```
$ cat backreference.txt
bonjour, "un deux" blabla "trois quatre"
"cinq" blabla2
nous sommes mercredi ou jeudi
au revoir "six"  "sept huit"

$ sed  -E 's/"([^"]*)"/\1/' < backreference.txt
bonjour, un deux blabla "trois quatre"
cinq blabla2
nous sommes mercredi ou jeudi
au revoir six  "sept huit"

$ sed  -E 's/"([^"]*)"/\1/g' backreference.txt
bonjour, un deux blabla trois quatre
cinq blabla2
nous sommes mercredi ou jeudi
au revoir six  sept huit

$
```

Exemple 4 :
Soit un fichier contenant sur chaque ligne des champs séparés par un caractère `:`. La commande suivante permet d'inverser les deux premiers champs de chaque ligne.

```
$ cat backreference2.txt
eric:merlet:enseignant
```

```
julien:bernard:enseignant chercheur
frederic:dadeau:enseignant chercheur

$ sed  -E 's/([^:]*):([^:]*)/\2:\1/' < backreference2.txt
merlet:eric:enseignant
bernard:julien:enseignant chercheur
dadeau:frederic:enseignant chercheur
```

→ **Version 2**

On souhaite maintenant extraire du résultat produit par la commande `ip(1)` (c'est-à-dire ce qu'elle affiche sur la sortie standard) l'IPv4 de la machine, et ensuite déterminer les adresses IPv4 attribuées dans une plage d'adresses du réseau local où la machine se situe.
Sur les machines de l'université, l'interface filaire de connexion au réseau s'appelle `enp0s31f6`. La commande `ip address show enp0s31f6` permet d'afficher la configuration de cette interface `enp0s31f6`. Comme le nom de cette interface peut changer d'un système à l'autre, il sera transmis en argument au script (qui devra vérifier sa validité).
Le synopsis de lancement du script `pingv2.sh` devient donc :
`./pingv2.sh INTERFACE_NAME HOST_PART_START HOST_PART_STOP`

Exemples d'exécution :

```
$ ./pingv2.sh blabla 170 180
Error : blabla isn't a valid network interface name
Usage : ./pingv2.sh INTERFACE_NAME HOST_PART_START HOST_PART_STOP
$ ./pingv2.sh enp0s31f6 170 180
Host IP : 172.20.128.178
Assigned IP addresses on the network 172.20.128.0 in the interval [.170, .180] :
172.20.128.170
172.20.128.178
--> Number of assigned IP addresses : 2
$
```

Question 9.7 Extraire l'IPv4 de votre machine du résultat produit par la commande `ip(1)` (le nom de l'interface reçu en argument du script doit être tranmis à cette commande) :

- Sélectionner la ligne contenant la chaîne `inet` suivie d'un espace (ou d'un caractère différent du chiffre `6`[2]).
- Extraire de cette ligne uniquement l'adresse IPv4 de votre machine en utilisant soit `sed(1)`, soit `awk(1)`, soit les possibilités d'extraction de sous-chaînes offertes par `dash(1)`.

Question 9.8 Compléter le script pour répondre au cahier des charges.

2. Les adresses IP indiquées sur les lignes commençant par `inet6` sont des adresses IPv6.

Exercice 10 : Destruction récursive des répertoires vides

Le but est de faire un script `rec_rmdir.sh` qui efface récursivement les répertoires vides du répertoire courant. Pour cela, on peut définir une fonction nommée `rec_rmdir` qui fera tout le travail et uniquement appeler cette fonction avec le répertoire courant :

```
rec_rmdir .
```

La fonction `rec_rmdir` doit supprimer tous les répertoires vides contenus dans le répertoire qu'elle reçoit en paramètre, sans supprimer ce répertoire reçu en paramètre.

Question 10.1 Dans la fonction `rec_rmdir`, vérifier qu'il y a un seul paramètre et l'afficher, sinon sortir de la fonction (avec la commande `return`). Indice : `$#`

Question 10.2 Vérifier que le paramètre est bien un répertoire. Indice : `test(1)`.

Question 10.3 Parcourir tous les fichiers de ce répertoire et pour chaque fichier qui est lui-même un répertoire, afficher son nom. Indice : `for`

Arrivé à ce stade, il ne reste plus qu'à appeler récursivement la fonction `rec_rmdir` sur le répertoire trouvé puis qu'à le supprimer avec `rmdir(1)`.
Seulement, si la fonction s'appelle récursivement sans précaution, la variable utilisée pour itérer sur les fichiers sera écrasée : en effet, par défaut, toutes les variables sont globales, y compris celles définies dans une fonction. Pour résoudre ce problème, il faut soit créer un sous shell (qui disposera d'une *copie* de toutes les variables du shell) à chaque appel de la fonction, soit déclarer que la variable utilisée pour itérer est locale à la fonction (c'est la meilleure solution).
Pour créer un sous shell, il faut entourer le groupe de commandes concerné par des parenthèses (semblables aux accolades). Deux manières de faire sont possibles :

- soit placer les parenthèses autour de l'appel récursif uniquement
- soit placer les parenthèses à la place des accolades qui délimitent le corps de la fonction

Question 10.4 Compléter le script en supprimant les répertoires vides avec les indications données. On veillera notamment à supprimer le message d'erreur de `rmdir(1)` quand le répertoire n'est pas vide.

Question 10.5 Compléter le script en supprimant également les fichiers ordinaires vides.

Exemple d'exécution sans la suppression des fichiers ordinaires vides (le script `create_files.sh` ci-dessous permet de créer une arborescence de fichiers ; les affichages du script `rec_rmdir.sh` ont été supprimés) :

```
$ cat create_files.sh
#!/bin/dash

mkdir -p rep1/rep11/rep111
mkdir    rep1/rep11/rep112
mkdir -p rep1/rep12/rep121
mkdir    rep1/rep12/rep122

mkdir -p rep2/rep21/rep211
echo 'hello' > rep2/rep21/fic212
mkdir    rep2/rep21/rep213
mkdir -p rep2/rep22/rep221
mkdir    rep2/rep22/rep222
mkdir    rep2/rep22/rep223

mkdir -p rep3/rep31/rep311
touch    rep3/rep31/fic312
mkdir    rep3/rep31/rep313
mkdir -p rep3/rep32/rep323

$ ./create_files.sh
$ ./rec_rmdir.sh
$ ls -R .
.:
create_files.sh  rec_rmdir.sh  rep2 rep3

./rep2:
rep21

./rep2/rep21:
fic212

./rep3:
rep31

./rep3/rep31:
fic312
```

Si le script `rec_rmdir.sh` supprime les fichiers ordinaires vides, alors les fichiers `fic312`, `rep31` et `rep3` sont également supprimés.

Exercice 11 : Calculer le factoriel d'un nombre

Le but est de faire un script `factorielle.sh` qui calcule la factorielle d'un nombre. Ce script doit donc toujours prendre un argument, et cet argument doit être un entier supérieur ou égal à 0.
Le script doit contenir une fonction récursive.

Le synopsis du script `factorielle.sh` est le suivant :
`./factorielle.sh N`

Quelques exemples d'exécution :

```
$ ./factorielle.sh 15 5
Error : the number of arguments isn't correct
Usage: ./factorielle.sh N
$ ./factorielle.sh lm
Error : the argument lm isn't an integer
Usage: ./factorielle.sh N
$ ./factorielle.sh 3.6
Error : the argument 3.6 isn't an integer
Usage: ./factorielle.sh N
$ ./factorielle.sh -3
Error : the argument -3 is a negative integer
Usage: ./factorielle.sh N
$ ./factorielle.sh 0
0!=1
$ ./factorielle.sh 1
1!=1
$ ./factorielle.sh 5
5!=120
$ ./factorielle.sh 13
13!=6227020800
$ ./factorielle.sh 15
15!=1307674368000
```

Question 11.1 Vérifier la présence et la validité de l'argument transmis.

Question 11.2 Compléter le script en définissant et en appelant une fonction récursive.

Travaux Pratiques de Système n°4

Exercice 12 : Environnement

Les variables d'environnement sont définies sous la forme de chaînes de caractères contenant des affectations du type `NOM=VALEUR`. Ces variables sont accessibles aux processus, tant dans les programme en C que dans les scripts shell. Lors de la duplication d'un processus avec un `fork()`, le processus fils hérite d'une copie des variables d'environnement de son père.

Un certain nombre de variables sont automatiquement initialisées par le système lors de la connexion de l'utilisateur. D'autres sont mises en place par les fichiers d'initialisation du shell, d'autres enfin peuvent être utilisées temporairement dans des scripts shell avant de lancer une application.

→ La variable globale `environ`

Un programme C peut accéder à son environnement (voir `environ(7)`) en utilisant la variable globale `environ`, à déclarer ainsi en début de programme :

```
extern char **environ;
```

Cette variable `environ` pointe sur un tableau de pointeurs de caractère. Chaque pointeur de caractère contient l'adresse du premier caractère d'une chaîne, de la forme `NOM=VALEUR`. Le dernier pointeur de ce tableau vaut NULL, ce qui permet de connaître la fin du tableau.

Question 12.1 Écrire un programme `env` qui lit l'ensemble des variables d'environnement et les affiche (sans modifier la variable globale `environ`). Comparer la sortie du programme `env` avec celle de la commande `env(1)` qui fait la même chose quand on l'appelle sans argument.

Question 12.2 Définir une variable `FOOBAR` dans le shell et lui donner la valeur `Hello World!`. Vérifier qu'elle n'apparaît pas à l'appel de `env`. Exporter la variable `FOOBAR` dans l'environnement via la commande `export`. Vérifier qu'elle apparaît à l'appel de `env`.

→ Le 3ème paramètre (`envp`) de la fonction `main()`

Certains systèmes d'exploitation (dont Linux et Windows) permettent de définir une fonction `main()` avec un troisième paramètre permettant l'accès à l'ensemble des variables d'environnement. Cette fonction `main()` a pour prototype :

```
int main(int argc, char **argv, char **envp);
```

Comme la variable globale `environ`, le paramètre `envp` pointe sur un tableau de pointeurs de caractère. Le dernier pointeur du tableau vaut `NULL`.

Question 12.3 Écrire un programme `env2` qui fait la même chose que le programme `env` en utilisant le paramètre `envp`.

→ **La fonction `getenv(3)`**

```
char *getenv(const char *name);
```

La fonction `getenv(3)` (appartenant à la bibliothèque standard) permet de rechercher une variable d'environnement. On lui donne le nom de la variable désirée, et elle renvoie un pointeur sur le caractère suivant immédiatement le signe = dans l'affectation `NOM=VALEUR` (Le pointeur renvoyé constitue une chaîne de caractères car le dernier caractère de la "VALEUR" est suivi par un '\0'). Si la variable n'est pas trouvée, elle renvoie `NULL`.

Question 12.4 Écrire un programme `readenv` qui lit et affiche les variables d'environnement suivantes : `HOME`, `LANG`, `PATH`, `PWD`, `SHELL`, `USER`.
On pourra, par exemple, définir une fonction `void printenv(const char *name);` qui regarde si la variable d'environnement `name` existe et affiche son nom et sa valeur quand elle existe, et dans le cas contraire que la variable d'environnement n'existe pas.

Exercice 13 : La commande `cat(1)`

La commande `cat(1)` permet de concaténer des fichiers textes et d'envoyer le résultat sur la sortie standard. On ne va pas stocker le résultat, on va directement l'envoyer sur la sortie standard.

Synopsis :

```
cat file...
```

Question 13.1 Implémenter la commande `cat(1)`. Pour chaque fichier passé en ligne de commande :

1. ouvrir le fichier en lecture seule ;
2. lire les données dans un tableau de 64 caractères ;
3. écrire les données lues sur la sortie standard ;
4. tant qu'il y a des données dans le fichier, aller à l'étape 2 ;
5. fermer le fichier.

Exercice 14 : Écriture/lecture d'un fichier binaire

→ Le programme `write_int`

Synopsis :

```
write_int file N...
```

L'argument `file` représente un chemin vers le fichier à écrire. Cet argument est suivi d'un nombre d'entier compris entre 1 et l'infini.

Question 14.1 Écrire un programme `write_int` qui :
- ouvre un flux associé au fichier `file`
- convertit les arguments suivants en entiers, tout en les stockant tous dans un unique tableau alloué dynamiquement
- écrit tous les entiers dans le fichier en appelant une seule fois la fonction `fwrite(3)`
- ferme le flux

→ Le programme `read_int_v1`

Ce programme doit être capable de lire **séquentiellement** un fichier produit par le programme précédent. Les entiers lus sont affichés sur la sortie standard. Au début du programme, le nombre d'entiers stockés dans le fichier n'est pas connu.

Question 14.2 Écrire le programme `read_int_v1`.

→ Le programme `read_int_v2`

Ce programme doit faire la même chose que le programme précédent, mais en n'appelant qu'une seule fois la fonction `fread(3)` ; sans connaître à l'avance le nombre d'entiers présents dans le fichier, et sans surdimensionner le tableau utilisé.
Indice : on peut utiliser les fonctions `fseek(3)` et `ftell(3)` pour connaître la taille d'un fichier.

Exemples d'exécution :

```
$ ./write_int fic.dat 12 58 652 548 -953
$ od -A x -t x1 fic.dat
000000 0c 00 00 00 3a 00 00 00 8c 02 00 00 24 02 00 00
000010 47 fc ff ff
000014
$ ./read_int_v1 fic.dat
12 58 652 548 -953
$ ./read_int_v2 fic.dat
12 58 652 548 -953
```

Question 14.3 Écrire le programme `read_int_v2`.

→ **Le programme `read_int_v3`**

Ce programme doit permettre un **accès direct** à n'importe quel entier stocké dans un fichier écrit par le programme `write_int`. Il commence par afficher le nombre d'entiers stockés dans le fichier. Ensuite, il entre dans une boucle qui :

— demande à l'utilisateur de saisir la position d'un entier à lire

— affiche l'entier du fichier situé à la position demandée

On supposera que le premier entier du fichier est situé à la position 1. Si l'utilisateur saisit une position non valide, par exemple trop grande, il faut le lui signaler. Il faut également fournir à l'utilisateur un moyen "propre" de sortir de la boucle.

Exemples d'exécution :

```
$ ./write_int fic.dat 12 58 652 548 -953
$
$ ./read_int_v3 fic.dat
The file "fic.dat" contains 5 integer(s).

Enter the position of the integer to read (0 to go out) : 5
The integer located at position 5 is equal to -953
Enter the position of the integer to read (0 to go out) : 2
The integer located at position 2 is equal to 58
Enter the position of the integer to read (0 to go out) : 6
The file "fic.dat" only contains 5 integer(s)!
Enter the position of the integer to read (0 to go out) : -1
The position can't be negative!
Enter the position of the integer to read (0 to go out) : 0
$ echo $?
0
```

Question 14.4 Écrire le programme `read_int_v3`.

Travaux Pratiques de Système n°5

Exercice 15 : La commande `tee(1)`

Le but de cet exercice est de réimplémenter la commande `tee(1)` :

```
tee [FILE]...
```

La commande `tee(1)` écrit, tout ce qu'elle lit sur son entrée standard, sur sa sortie standard et dans tous les fichiers reçus en arguments.

Question 15.1 Implémenter la commande `tee(1)`.

Exercice 16 : La commande `strings(1)`

Le but de cet exercice est d'implémenter la commande `strings(1)` en C.

La commande `strings(1)` permet d'afficher les séquences de caractères imprimables qui sont incluses dans un fichier quelconque. Par exemple, pour un exécutable, elle permet d'afficher toutes les chaînes de caractères définies dans le programme et qui apparaissent dans le binaire final. Pour ne pas afficher des caractères qui ne feraient pas partie d'une chaîne, le programme n'affiche que les séquences d'au moins quatre caractères consécutifs (ou du nombre indiqué après l'option `-n`). Après chaque séquence, le programme passe à la ligne.

Synopsis :

```
strings [-n N] FILE
```

Exemples :

```
$ strings /bin/ls
$ strings -n 6 /bin/rm
```

Question 16.1 Déterminer le nombre de caractères consécutifs à partir duquel on doit afficher les caractères. On supposera que l'option `-n` ne peut apparaître qu'en première position.

Question 16.2 Ouvrir le fichier passé en paramètre et en cas d'erreur, afficher un message. On supposera que le nom du fichier est toujours donné en paramètre.

Question 16.3 En réalité, la commande affiche les séquences de caractères qui sont, soit imprimables, soit égaux à des tabulations. Lire le fichier et afficher les séquences de caractères, imprimables ou tabulations, suivant les spécifications données. On utilisera notamment la fonction `isprint(3)` qui permet de savoir si un caractère est imprimable.

Question 16.4 Fermer le fichier.

Question 16.5 Vérifier que votre programme donne les mêmes résultats que la vraie commande.

Exercice 17 : La commande `cut(1)`

Le but de cet exercice est de réimplémenter une version simplifiée de la commande `cut(1)` en langage C.

```
cut DELIM FIELD_NUMBER [FILE]
```

La commande à implémenter prendra 2 arguments obligatoires et un argument optionnel. Les deux arguments obligatoires sont, dans l'ordre, le caractère séparateur `DELIM` (codé sur un seul octet), et le numéro du champ à sélectionner `FIELD_NUMBER` (le premier champ étant numéroté 1). L'argument optionnel sera le chemin du fichier à lire. S'il n'y a pas de troisième argument, on lira l'entrée standard. Voici un exemple d'utilisation :

```
cut ':' 1 /etc/passwd
```

Cette commande sélectionne le premier champ du fichier `/etc/passwd` avec le caractère `:` comme séparateur. Vous noterez que, contrairement à l'outil que vous avez utilisé, le nom des options n'apparaît pas.

Question 17.1 Déterminer le caractère séparateur `DELIM` et le numéro du champ `FIELD_NUMBER`. On affichera un message d'erreur si le séparateur indiqué contient plus d'un octet, ou si le numéro du champ est strictement inférieur à 1.

Question 17.2 Déterminer le fichier à lire. On affichera un message d'erreur si le fichier donné en argument ne peut pas être ouvert.

Question 17.3 Afficher le champ sélectionné sur chaque ligne. On veillera à ne pas afficher le caractère séparateur, et à prendre en compte les passages à la ligne correctement.

Question 17.4 Fermer le flux si besoin.

Exercice 18 : La commande `ls(1)`

Ce programme doit être compilé, comme les précédents, avec les options `-std=c99` et `-Wall`. Dans les exercices de programmation système, vous devrez parfois ajouter l'option `-D_DEFAULT_SOURCE`[3] "to get definitions that would normally be provided by default". C'est le cas ici.
Pour avoir plus d'informations à ce sujet : RTFM `feature_test_macros(7)`.

Question 18.1 Écrire un programme `ls(1)` qui liste les fichiers non cachés du répertoire courant. Indice : `opendir(3)`

Question 18.2 Gérer l'option `-l`. Indices : `stat(2)`, `getpwuid(3)`, `getgrgid(3)`

3. L'ajout de l'option `-D_DEFAULT_SOURCE` (transmise à `gcc`) est équivalente à l'ajout d'un `#define _DEFAULT_SOURCE` au tout début du fichier source, c'est à dire **avant** les inclusions des fichiers d'en-têtes système. Les fichiers d'en-têtes système contiennent des directives de compilation condionnelle qui permettent de sélectionner les déclarations et les définitions finalement "incluses" dans le fichier source après le passage du préprocesseur.

Travaux Pratiques de Système n°6

Exercice 19 : Sauvegarde du PID

Certains programmes (les daemons en particulier) sauvegarde leur PID dans un fichier, ce qui permet à des commandes extérieures de leur envoyer un signal plus facilement. Ces fichiers sont généralement sauvegardés dans le répertoire `/var/run` ou `/run`.

Question 19.1 Écrire un programme `myself` qui écrit son pid dans un **fichier texte** nommé `myself.pid`. Ce fichier sera placé dans le répertoire de travail courant du processus.

Exercice 20 : Signaux

Rappel : vous devez compiler tous vos programmes avec les options `-std=c99` et `-Wall`. Dans les exercices de programmation système, vous devrez parfois ajouter l'option `-D_DEFAULT_SOURCE`[4] "to get definitions that would normally be provided by default".
Pour avoir plus d'informations à ce sujet : RTFM `feature_test_macros(7)`.

Remarque : pour installer un handler de signal, vous n'avez pas le droit d'utiliser l'appel système `signal(2)`.
Extrait de la man page de `signal(2)` : "The behavior of signal() varies across UNIX versions, and has also varied historically across different versions of Linux. Avoid its use : use sigaction(2) instead."

Dans cet exercice, il ne faut pas créer un nouveau processus avec `fork(2)`.

Question 20.1 Écrire un programme `invincible`, qui écrit «boom !» quand il reçoit `SIGINT`, et qui continue de s'exécuter. Indices : `sigaction(2)`, `pause(2)`, `signal(7)`.

Question 20.2 Faire en sorte qu'il abandonne au bout de n tentatives (valeur donnée en argument de la commande).

```
$ ./invincible 5
^Cboom!
^Cboom!
^Cboom!
^Cboom!
^CKABOOM!
$
```

4. L'ajout de l'option `-D_DEFAULT_SOURCE` (transmise à `gcc`) est équivalente à l'ajout d'un `#define _DEFAULT_SOURCE` au tout début du fichier source, c'est à dire **avant** les inclusions des fichiers d'en-têtes système. Les fichiers d'en-têtes système contiennent des directives de compilation condionnelle qui permettent de sélectionner les déclarations et les définitions finalement "incluses" dans le fichier source après le passage du préprocesseur.

Question 20.3 Modifier le pour qu'il compte les moutons (1 par seconde) lorsqu'il n'est pas embêté. Mais faites en sorte qu'il reste inerte pendant les cinq secondes qui suivent un «boom !», s'il ne reçoit pas pendant ces 5 secondes un nouveau signal `SIGINT`. Les «boom !» doivent pouvoir se succéder "instantanément". Indice : `alarm(2)`.

```
$ ./invincible 2
1
2
^Cboom!
3
4
5
^CKABOOM!
$ ./invincible 3
1
2
3
^CBoom!
^CBoom!
^CKABOOM!
$
```

Exercice 21 : La commande `kill(1)`

La commande `kill(1)` permet d'envoyer un signal à un processus, `SIGTERM` par défaut. Le signal peut être précisé avec l'option `-s` suivi du numéro du signal voulu. Les processus destinataires du signal sont désignés par leur PID.

Synopsis du programme à écrire :

```
kill [-s SIG] PID...
```

Remarques :
- Utiliser la commande `kill -l` pour visualiser les signaux existants et leur numéro.
- Dans le programme à écrire, le signal à envoyer (i.e. l'argument `SIG`) ne pourra être transmis que sous la forme d'un entier ; les noms symboliques des signaux, comme `SIGTERM`, `SIGKILL`, etc., ne seront pas acceptés.

Question 21.1 Déterminer le signal à envoyer. On supposera que l'option `-s` ne peut apparaître qu'en première position.

Question 21.2 Envoyer le signal à chacun des processus donné en argument. Indice : `kill(2)`

Exemples d'exécution :

```
$ sleep 100 &
[1] 9356
$ sleep 100 &
```

```
[2] 9357
$ sleep 100 &
[3] 9358
$ ./kill -s SIGTERM 9356
SIGTERM : undefined signal
Usage: kill [-s signal] pid ...
$ ./kill -s 0 9356 azerty 9357 10000 9358
azerty : invalid pid
kill (10000): No such process
$ ./kill -s 3 9356 azerty 9357 10000 9358
azerty : invalid pid
kill (10000): No such process
$
[1]   Quitter                 (core dumped) sleep 100
[2]-  Quitter                 (core dumped) sleep 100
[3]+  Quitter                 (core dumped) sleep 100
$
$ ./kill 9356 azerty 9357 9358 10000
kill (9356): No such process
azerty : invalid pid
kill (9357): No such process
kill (9358): No such process
kill (10000): No such process
$
$ ./kill -s 15
Usage: kill [-s signal] pid ...
$ echo $?
1
```

Exercice 22 : Lanceur de commande

Nous allons réaliser un lanceur de commande `launch` chargé de lancer la commande passée en argument et de calculer le temps réel que la commande a mis pour s'exécuter.

Question 22.1 Récupérer l'heure courante à l'aide de `clock_gettime(2)`. Cette fonction permet de récupérer l'heure courante à l'aide d'une structure de type `timespec` :

```
struct timespec {
    time_t   tv_sec;        /* seconds */
    long     tv_nsec;       /* nanoseconds */
};
int clock_gettime(clockid_t clockid, struct timespec *tp);
```

Pour le paramètre `clockid`, vous pouvez utiliser la constante macrodéfinie `CLOCK_MONOTONIC_RAW`.

Question 22.2 Créer un nouveau processus et exécuter la commande passée en argument dans le fils, le père attendant la terminaison du fils. On utilisera la variante suivante de la famille `exec(3)` :

```
int execvp(const char *file, char *const argv[]);
```

Question 22.3 Le paramètre transmis à `wait(2)` permet de récupérer des informations sur le status de terminaison du fils. Utiliser les macros adéquates WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG (cf. manpage de `wait(2)`) pour afficher comment le processus fils s'est terminé.

Question 22.4 Récupérer l'heure courante puis calculer la durée (très approximative) du processus fils.

Exemples d'exécution :

```
$ ./launch sleep 2
8071 exited, status=0
Duration of child process : 2002239977ns

$ ./launch sleep 100 &
[1] 8131
$ ps -H
  PID TTY          TIME CMD
 7951 pts/0    00:00:00 bash
 8131 pts/0    00:00:00   launch
 8132 pts/0    00:00:00     sleep
 8133 pts/0    00:00:00   ps
$ kill -s SIGQUIT 8132
$ 8132 killed by signal 3
Duration of child process : 16700055431ns

[1]+  Fini                    ./launch sleep 100
$
```

Exercice 23 : La fonction `system(3)`

Le but de cet exercice est d'implémenter la fonction `system(3)` dont le prototype est :

```
int system(const char *command);
```

La fonction `system()` exécute la commande indiquée dans `command` en appelant `/bin/sh -c command`, et revient après l'exécution complète de la commande. Durant cette exécution, le signal `SIGCHLD` est masqué (uniquement dans le père), et les signaux `SIGINT` et `SIGQUIT` sont ignorés dans le père (ces 2 signaux seront traités conformément à leurs valeurs par défaut dans le processus fils).

Valeur renvoyée par la fonction `system()` :
- Si `command` est `NULL`, la fonction `system()` renvoie 1 (il n'y a pas de création de nouveau processus dans ce cas)
- Si un nouveau processus ne peut pas être créé, ou si son statut ne peut pas être récupéré par `waitpid(2)`, la fonction `system()` renvoie -1
- Si le recouvrement échoue, le processus fils se termine normalement avec un statut de terminaison égal à 127
- Sinon, elle renvoie le status de terminaison du processus fils récupéré dans le père par un appel de `waitpid(2)`

Question 23.1 Implémenter la fonction `system()`. On justifiera précisément la version de la famille `exec` utilisée à l'aide d'un commentaire dans le code.

Question 23.2 Dans la fonction `main()` (qui appelle votre fonction `system()`), utiliser les macros adéquates (WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG) pour afficher comment le processus fils s'est terminé (cf. manpage de `wait(2)`).

Exemples d'exécution à tester :

```
$ ./system "sleep 10"            --> avec ou sans ^C
$ ./system "ps -lH"
$ ./system "ps -lH | tee /dev/tty | wc -l"
```

Conclusion : la fonction `system()` fournit de la facilité et de la commodité : elle s'occupe de tous les détails d'appel de `fork(2)`, `execl(3)` et `waitpid(2)`, ainsi que des manipulations nécessaires des signaux ; de plus, l'interpréteur réalise les substitutions habituelles et les redirections d'entrées et sorties pour `command`. Le coût principal de `system()` est l'**inefficacité** : des appels système supplémentaires sont nécessaires pour créer le processus qui exécute l'interpréteur shell et pour exécuter ce processus.

Travaux Pratiques de Système n°7

Exercice 24 : Copie de fichiers

Le but de cet exercice est de réaliser une copie de fichier à l'aide de deux processus : un qui va lire le fichier à copier et un autre qui va écrire le nouveau fichier. Les deux processus vont communiquer à l'aide d'un tube. La commande prendra donc en paramètre deux noms de fichier : la source et la destination.

Question 24.1 Vérifier que la commande a bien deux paramètres et écrire un message sur l'erreur standard sinon.

Question 24.2 Vérifier que le fichier source est un fichier régulier, et que les liens source et destination ne pointent pas sur le même inode. Terminer le programme et écrire un message sur l'erreur standard sinon.

Question 24.3 Créer les deux processus et le tube et faire la copie comme indiqué. On veillera à ce que tous les descripteurs ouverts soient correctement fermés dans tous les processus, en contrôlant les erreurs pouvant se produire lors de ces fermetures de descripteurs.

Question 24.4 Utiliser les macros adéquates (WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG) pour afficher dans le processus père comment les processus fils se sont terminés.

Question 24.5 À votre avis, est-ce que cette méthode permet d'accélérer la copie par rapport à un seul processus ? Pourquoi ?

Question 24.6 On veut à présent pouvoir arrêter la copie en cours et supprimer le fichier destination à l'aide de `CTRL+C`. Modifier le programme pour mettre en place ce comportement. Indice : `unlink(2)`.

Exercice 25 : Calcul de π

Une méthode pour obtenir une approximation de π est de tirer aléatoirement n points (x, y) avec $x, y \in [0, 1]$. Parmi ces n points, p points appartiennent au disque unité (de centre O et de rayon 1), c'est-à-dire que $x^2 + y^2 < 1$. Or, la probabilité de tomber dans le disque unité est de $\frac{\pi}{4}$. Une approximation de π est donc $4 * \frac{p}{n}$. On utilisera le type `unsigned long` pour n et p.

Question 25.1 En utilisant `random(3)` et `INT_MAX`, écrire une fonction C qui renvoie un `double` compris dans l'intervalle $[0, 1]$.

Correction de la question 25.1

```
double get_double(void) {
  return (double) random() / INT_MAX;
}
```

Question 25.2 Écrire une commande `seq_pi` qui fait le calcul de π suivant la méthode indiquée en utilisant un seul processus. On fixera n à 10^7.

Correction partielle de la question 25.2

```
void compute_pi(void) {
  for (unsigned long i = 0; i < N; ++i) {
    double x = get_double();
    double y = get_double();
    if (x*x + y*y < 1.0) {
      ++p;
    }
  }
}
```

À partir de maintenant, on veut lancer plusieurs processus en même temps pour accélérer le calcul. Pour faire le calcul final de π, on va propager les résultats [5] avec un tube entre chacun des processus, le dernier processus réalisant le calcul final de π.
On prendra garde à initialiser le générateur aléatoire en utilisant `srandom(3)` et `getpid(2)` (de manière à ce que chaque processus ait une graine différente).

Question 25.3 Écrire une commande `biprocess_pi` qui utilise deux processus et un tube. On fixera n à 10^7.

Question 25.4 Écrire une commande `interrupted_pi` qui utilise deux processus et qui calcule tant qu'elle n'est pas interrompue (par un `CTRL+C`).

Question 25.5 Écrire une commande `parallel_pi` qui utilise q processus, q étant passé en paramètre de la commande et qui calcule tant qu'elle n'est pas interrompue (par un `CTRL+C`).

5. p pour la **question 25.3**, (n et p) pour les suivantes.