Carnet de Travaux Libres
Système et programmation système
Licence 2 Informatique

Julien Bernard Eric Merlet

Introduction

Le carnet de travaux libres est une compilation de sujets de travaux dirigés et de travaux pratiques. Il peut vous servir à réviser, à vous auto-évaluer, à mieux comprendre certains concepts, etc. Il contient aussi les anciens sujets d'examen, sous forme d'exercices séparés.

Table des matières

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. Puis, ouvrez une console (`Konsole` par exemple) qui vous donne accès à un interpréteur de commande.

→ Connexion à votre compte

Avant de vous connecter sur votre compte, tapez CTRL+ALT+F1. 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'écran de login graphique, tapez CTRL+ALT+F7.

→ Commandes de base

Question 1.5 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, 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 vide. Pour rappel, la commande pour créer un répertoire est `mkdir`.

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 utilisateur. Entrer dans ce répertoire puis créer un répertoire `exemple`. Entrer dans le répertoire `exemple`.

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 `skywalker/anakin/README`.

Question 1.10 Supprimer le répertoire `skywalker/luke`. Que se passe-t-il si vous essayez de supprimer `skywalker/anakin` ?

Question 1.11 Renommer (en fait, déplacer) le répertoire `skywalker/anakin` en `darth_vader`.

Question 1.12 Créer un répertoire `yoda` et un fichier `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 un lien symbolique nommé `LREADME` dans `.private` 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`.

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 créer des archives dans des 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 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 trois 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.

Exercice 3 : Fichiers et ligne de commande

→ Arborescence et accès

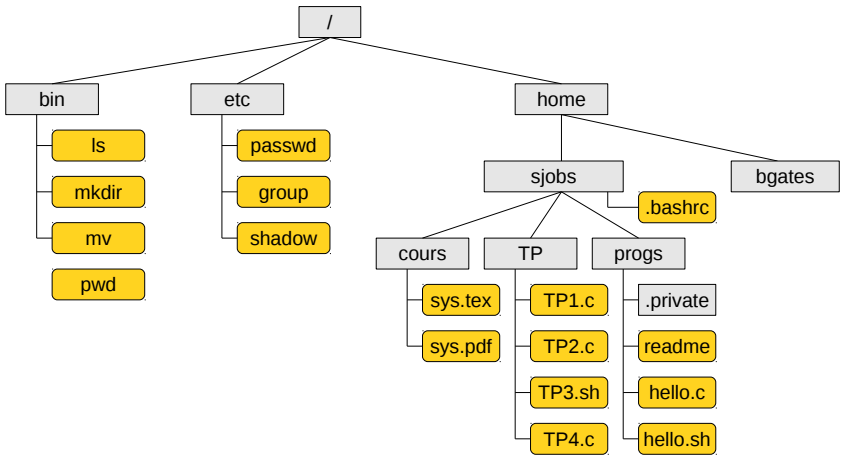

FIGURE 1 – Exemple d'arborescence

Question 3.1 En considérant que l'on se trouve dans le répertoire `sjobs` (fig. 1), quel est le résultat des commandes suivantes ?

1. `ls`
2. `ls ..`
3. `ls -a`
4. `ls -a ..`
5. `ls -a progs`

Question 3.2 En considérant que l'on se trouve dans le répertoire `TP` (fig. 1), quelles sont les commande nécessaires pour effectuer les opérations suivantes ?

1. Aller dans le répertoire `/etc` (deux solutions).
2. Copier `sys.tex` dans `.private` sous le nom `sys2.tex`
3. Déplacer `hello.c` et `hello.sh` dans `.private`.
4. Créer en une seule opération les répertoires `TD` et `TD/TD1` dans le répertoire `sjobs`.
5. Effacer le répertoire `TP`.

→ Liens physiques et de liens symboliques

L'option `-d` de la commande `ls` permet d'afficher les répertoires avec la même présentation que les fichiers, sans lister leur contenu.

Question 3.3 On considère l'arborescence fig. 1 initiale, et on repart du répertoire `TP` pour toutes les questions.

1. Donner 2 commandes permettant de visualiser le numéro d'inode du répertoire `TP`.
2. Donner le nombre de liens désignant ou pointant sur l'inode du répertoire `TP`. Justifier. Donner une commande permettant de vérifier votre résultat.
3. Mêmes questions pour les inodes de répertoire `sjobs` et `progs`.
4. Donner une commande permettant de créer dans `.private` un lien physique nommé `hello2.sh` vers l'inode du fichier `hello.sh`. Quel est alors le nombre de liens physiques pointant sur l'inode de `hello.sh` ? Un nouveau fichier sur disque est-il créé ?
5. On exécute la commande `echo "echo bonjour;pwd" > ../progs/hello.sh`. Quel est l'effet de cette commande ? Donner la commande permettant d'afficher le contenu du fichier `hello2.sh`. Qu'affiche-t-elle ?
6. Donner 2 commandes permettant de supprimer `hello.sh`. Quel est l'effet de l'exécution d'une de ces commandes ?
7. Donner une commande permettant de créer dans `sjobs` un lien symbolique nommé `hello3.sh` pointant sur le fichier `hello2.sh`. Un nouveau fichier sur disque est-il créé ? Si oui que contient-il ?
8. Donner la commande permettant d'afficher `hello3.sh`. Quel est le résultat obtenu ?
9. Que se passe-t-il si on supprime `hello2.sh` ?
   Quel est l'effet de la commande `unlink hello3.sh` (depuis `sjobs`) ?

→ Permissions

Question 3.4 Donnez l'équivalent octal des permissions suivantes :

1. `rw-r--r--`
2. `rwxr-xr-x`
3. `rwxrwxrwt`
4. `rwsr-xr-x`
5. `r--------`

Question 3.5 Donnez l'équivalent symbolique des permissions suivantes :

1. `660`
2. `555`
3. `700`
4. `4755`
5. `2644`

Question 3.6 Quel umask définir pour que les permissions par défaut des fichiers soit 600 ? Quelles seront les permissions par défaut des répertoires ?

Question 3.7 En admettant que l'on soit l'utilisateur `sjobs` et que les fichiers et répertoires situés dans le répertoire utilisateur aient été créé avec un umask 0022 :

1. Interdire le répertoire `.private` en lecture, écriture et accès à tout le monde (autre que `sjobs`)
2. Autoriser l'écriture du fichier `readme` au groupe
3. Interdire la lecture du fichier `hello.c` aux autres
4. Autoriser l'exécution de `hello.sh` à tout le monde

→ Inodes

Question 3.8 Admettons que chaque bloc mémoire fasse 2048 octets et que chaque adresse de bloc soit codée sur 4 octets. Quelle est la taille maximale d'un fichier ?

→ Sticky Bit

Question 3.9 L'utilisateur `eleve` n'appartient pas au groupe `eric`. Pour chaque question, on considère que le répertoire `rep` contient les 2 fichiers `ficeleve` et `ficeric` avec les propriétaire, groupe propriétaire et droits indiqués ci-dessous.

```
$ ls -la rep
total 8
drwxrwxr-x. 2 eric  eric  4096 18 janv. 11:11 .
drwxr-xr-x. 4 eric  eric  4096 18 janv. 11:12 ..
-rw-rw-r--. 1 eleve eleve    0 18 janv. 11:11 ficeleve
-rw-rw-r--. 1 eric  eric     0 18 janv. 11:11 ficeric
```

1. `eric` a-t-il le droit de supprimer `ficeleve` et `ficeric` ?
   `eleve` a-t-il le droit de supprimer `ficeleve` et `ficeric` ?

2. `eric` autorise l'ecriture aux autres sur le répertoire `rep`.
   ```
   $ ls -ld rep
   drwxrwxrwx. 2 eric  eric  4096 18 janv. 11:11 rep
   ```
   `eric` a-t-il le droit de supprimer `ficeleve` et `ficeric` ?
   `eleve` a-t-il le droit de supprimer `ficeleve` et `ficeric` ?

3. `eric` positionne le sticky bit sur le répertoire `rep`.
   ```
   $ ls -ld rep
   drwxrwxrwxt. 2 eric  eric  4096 18 janv. 11:11 rep
   ```
   `eric` a-t-il le droit de supprimer `ficeleve` et `ficeric` ?
   `eleve` a-t-il le droit de supprimer `ficeleve` et `ficeric` ?

Exercice 4 : Makefile et LaTeX

Le but de cet exercice est de créer un Makefile pour LaTeX. LaTeX est un logiciel de mise en forme de document à base de balises, très pratique pour écrire des TD de système par exemple.

Avec LaTeX, on ne voit pas le document final, on écrit un document texte (avec l'extension `.tex`), puis on le compile avec la commande `pdflatex(1)` qui va produire un document PDF. Seulement, une seule compilation ne suffit parfois pas. En effet, il est possible en LaTeX de faire des références vers d'autres parties du document. À la première compilation, l'ensemble des références (et leur page) va être sauvegardé dans un fichier avec l'extension `.aux`, les références apparaissent alors avec un point d'interrogation dans le document PDF. Puis, à la deuxième compilation, ce fichier `.aux` servira pour résoudre les références et produire le document PDF final.

Question 4.1 Écrire un fichier `Makefile` qui permet de produire un document `TD.pdf` à partir d'un fichier `TD.tex`.

Question 4.2 Écrire des règles génériques qui permettent de compiler un fichier LaTeX en document PDF.

Au cours de la compilation, la commande `pdflatex(1)` produit un fichier de log (avec l'extension `.log`) et un fichier pour la table des matières (avec l'extension `.toc`).

Question 4.3 Écrire une règle `clean`.

BibTeX est un logiciel associé à LaTeX qui permet de gérer une bibliographie. Pour cela, la commande `bibtex(1)` a besoin de lire un fichier `.bib` qui contient l'ensemble des références bibliographiques et le fichier `.aux` généré à la première compilation pour avoir une liste des références citées explicitement. Il produit alors un fichier `.bbl` qui sera inclus lors de la deuxième compilation pour afficher la bibliographie. La commande `bibtex(1)` prend en paramètre le nom du document, sans extension.

Question 4.4 Écrire un fichier `Makefile` qui permet de produire un document `article.pdf` à partir d'un fichier `article.tex` et d'une bibliographie `article.bib`.

Question 4.5 Écrire des règles génériques qui permettent de compiler un fichier LaTeX et sa bibliographie en document PDF.

Au cours de la construction de la bibliographie, la commande `bibtex(1)` produit un fichier de log (avec l'extension `.blg`).

Question 4.6 Écrire une règle `clean` et une règle `mrproper`.

Exercice 5 : `true(1)` et `false(1)`

On exécute les commandes suivantes :

```
$ whatis true
true (1)             - do nothing, successfully
$ whatis false
false (1)            - do nothing, unsuccessfully
```

Question 5.1 Quelle variable spéciale permet de lire le code de retour de la dernière commande ?

Question 5.2 Expliquez ce que font les commandes `true` et `false`.

Question 5.3 Donnez le code C des commandes `true` et `false`.

Question 5.4 Expliquez ce que fait le script shell suivant et pourquoi il fait bien ce qu'on attend.

```
while true
do
  echo -n "$RANDOM"
done
```

Question 5.5 Dans le fichier `/etc/passwd`, on peut lire :

```
mysql:x:107:112:MySQL Server:/var/lib/mysql:/bin/false
```

Expliquez l'utilisation de `false` à cet endroit.

Exercice 6 : Ouverture de fichier et permissions

Parmi les deux API vues en cours pour manipuler des fichiers (descripteur de fichier et flux), les paramètres de la fonction d'ouverture de fichier ne sont pas identiques. Pour rappel, voici les paramètres possibles pour l'API descripteur de fichier : `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_APPEND`, `O_CREAT`, `O_TRUNC`.

Question 6.1 Quelle combinaison de paramètres correspond à :
- `"r"` : Ouvre le fichier en lecture
- `"r+"` : Ouvre le fichier en lecture et écriture
- `"w"` : Tronque le fichier
- `"w+"` : Crée un fichier et l'ouvre en lecture et écriture
- `"a"` : Ouvre le fichier en ajout
- `"a+"` : Ouvre le fichier en lecture et ajout

Question 6.2 Avec un umask défini à 0026, avec quelle permission par défaut est créé un fichier ? et un répertoire ? Dans les deux cas, on donnera la représentation symbolique et la représentation octale.

Question 6.3 On exécute la commande suivante sur un fichier `foo` exécutable :

```
$ chmod g+s foo
```

Quelles sont les permissions avant la commande ? Quelles sont ses nouvelles permissions ? Dans les deux cas, on donnera la représentation symbolique et la représentation octale. Expliquer ce que signifie cette nouvelle permision accordée ?

Exercice 7 : Le dossier `/tmp`

On exécute les commandes suivantes avec l'utilisateur `alice` :

```
$ ls -ld /tmp
drwxrwxrwt 13 root root 12288 mars   5 15:49 /tmp
$ umask
0027
$ mkdir /tmp/foo
$ chmod g+s /tmp/foo
$ touch /tmp/foo/bar
$ mkdir /tmp/foo/baz
```

Question 7.1 Que signifie le `d` au début de la ligne 2 ? Que signifie le `t` avant le 13 à la ligne 2 ?

Question 7.2 Donner les permissions en octal du répertoire `/tmp` ? Pourquoi les quatre dernières commandes ne renvoie-t-elle pas d'erreur ?

Question 7.3 Quelles sont les permissions (en octal et en symbolique) du répertoire `foo` après la troisième commande ? Et après la quatrième commande ?

Question 7.4 Quelles sont les permissions (en octal et en symbolique) du fichier `bar` après la cinquième commande ? Dire en français ce que cela signifie.

Question 7.5 Quel est le propriétaire du fichier `bar` ? Pourquoi ?

Question 7.6 Quelles sont les permissions (en octal et en symbolique) du répertoire `baz` ?

Exercice 8 : Fichiers dans `/dev`

```
$ ls -l /dev/sd*
brw-rw---- 1 root disk 8, 0 mars  18 10:54 /dev/sda
brw-rw---- 1 root disk 8, 1 mars  18 09:55 /dev/sda1
brw-rw---- 1 root disk 8, 2 mars  18 09:55 /dev/sda2
brw-rw---- 1 root disk 8, 5 mars  18 09:55 /dev/sda5
brw-rw---- 1 root disk 8, 6 mars  18 09:55 /dev/sda6
```

Question 8.1 Que représentent les fichiers `sd*` ?

Question 8.2 Que signifie le `b` au début de la ligne ?

Question 8.3 Donner la représentation en octal des permissions de `/dev/sda`. Donner la signification de ces permissions dans une phrase.

Question 8.4 Que représente `root` ? Que représente `disk` ?

Question 8.5 Si on lit le fichier `/dev/sda1`, que va-t-on y trouver ? Quelle commande utiliser pour savoir si je peux le lire ?

Exercice 9 : Permissions spéciales

On exécute les commandes suivantes :

```
$ ls -l /bin/su
-rwsr-xr-x 1 root root   40168 nov.  20 23:03 /bin/su
$ ls -l /sbin/unix_chkpwd
-rwxr-sr-x 1 root shadow 35408 août   9  2014 /sbin/unix_chkpwd
```

Question 9.1 Donner les permissions en octal du fichier `/bin/su`

Question 9.2 Donner le nom de la permission spéciale associée à `/bin/su` et expliquer son principe général

Question 9.3 Sachant que la commande `su(1)` permet de changer d'identifiant utilisateur ou de devenir super-utilisateur, expliquer l'intérêt de cette permission spéciale.

Question 9.4 Donner les permissions en octal du fichier `/sbin/unix_chkpwd`

Question 9.5 À votre avis, à quoi peut servir cette commande ?

Question 9.6 À quel fichier est liée cette commande ? Quel est le propriétaire, le groupe propriétaire, ainsi que les permissions de ce fichier ? Justifier.

Exercice 10 : Avec le fichier `/etc/passwd`

Le fichier `passwd(5)` contient la description des utilisateurs. Le format du fichier est le suivant :

```
login:passwd:uid:gid:name:home:shell
```

À partir de ce fichier et des commandes vues en cours, on veut les informations suivantes.

Question 10.1 La liste des logins.

Question 10.2 Le nombre d'utilisateurs.

Question 10.3 La liste des utilisateurs (par leur nom).

Question 10.4 Le nombre d'utilisateurs de `/bin/zsh`.

Question 10.5 Le nombre d'utilisateurs dont le login débute par la lettre d.

Question 10.6 Un fichier nommé `loginD` contenant la liste triée des logins commençant par la lettre d.

Exercice 11 : Le fichier `group(5)`

Question 11.1 Qu'est-ce qu'un groupe ? Quelle relation y a-t-il avec le fichier `group(5)` ? Dans quel répertoire trouve-t-on ce fichier ?

Question 11.2 À votre avis, quels sont le groupe et l'utilisateur propriétaire du fichier `group(5)`, ainsi que ses permissions (qu'on donnera sous forme symbolique et sous forme octale) ? Justifier.

Question 11.3 Le format du fichier est le suivant :

```
nom_du_groupe:mot_de_passe:GID:liste_utilisateurs
```

où les utilisateurs sont séparés par des virgules.

Donnez la commande qui permet d'obtenir les informations suivantes :

1. Le nombre de groupes
2. Le nombre de groupes auquels appartient l'utilisateur `john`
3. La liste ordonnée des noms de groupe
4. Le deuxième utilisateur du group `users`
5. Le plus grand GID utilisé dans le fichier

Question 11.4 Écrire un script `addgroup.sh` qui prend deux paramètres, le nom du groupe et le GID, et qui ajoute un groupe avec son GID au fichier `group(5)`. On vérifiera le bon nombre de paramètres. On vérifiera que le groupe n'existe pas déjà et que le GID n'existe pas déjà. On mettra un `x` comme mot de passe. On ajoutera le groupe à la fin du fichier.

Exercice 12 : Le fichier `services(5)`

Le fichier `/etc/services` est un fichier texte ASCII fournissant une correspondance entre des noms textuels faciles à mémoriser pour les services et les numéros de ports qui leur sont assignés, ainsi que les types de protocoles (TCP ou UDP). Voici un extrait de ce fichier :

```
ftp-data        20/tcp
ftp             21/tcp
ssh             22/tcp
ssh             22/udp
telnet          23/tcp
smtp            25/tcp
time            37/tcp
time            37/udp
gopher          70/tcp
gopher          70/udp
finger          79/tcp
http            80/tcp
http            80/udp
```

Question 12.1 Donnez la commande qui permet d'obtenir les informations suivantes :

1. Le nombre de service TCP.
2. La liste des services dans l'ordre alphabétique, sans doublon.
3. Les services qui utilisent les protocoles UDP et TCP.
4. Les numéros de port des services commençant par la lettre `t`.

Question 12.2 On suppose qu'il y a des commentaires dans ce fichier sur des lignes commençant par `#`. Quelle commande ajouter à celles que vous avez données pour conserver le résultat ?

Question 12.3 Écrire un script shell `getservbyport.sh` qui prend en paramètres un numéro de port et un protocole et qui renvoie le nom du service s'il existe. On vérifiera que le nom du protocole est soit `tcp`, soit `udp`.

Question 12.4 Écrire un script shell `addserv.sh` qui prend en paramètre un nom de service, un numéro de port et un protocole et qui ajoute le service au fichier `services(5)`. On vérifiera que le service n'existe pas déjà, et s'il existe, que le numéro de port est identique mais que le protocole est différent. Sinon, on renverra un message d'erreur.

Exercice 13 : Fichier de notes

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

```
nom:prenom:note1:note2:note3
```

où chaque note correspond à une épreuve. On supposera que les notes sont entières.

Question 13.1 À partir du fichier `notes.txt` et des commandes vues en cours, on veut connaître :

1. Le nombre d'étudiants.
2. La liste des étudiants dans l'ordre alphabétique sous la forme «Nom Prénom».
3. La liste des étudiants ayant eu 20 à au moins une des trois épreuves.
4. La liste des notes de la seconde épreuve avec pour chaque note le nombre d'étudiants l'ayant obtenu.

Question 13.2 Écrire un script shell qui prend sur son entrée standard le fichier `notes.txt` et qui affiche sur la sortie standard le résultat avec le format suivant :

```
nom:prenom:moyenne
```

où `moyenne` est la moyenne des trois notes.

Exercice 14 : La commande `who(1)`

La commande `who(1)` donne la liste des utilisateurs connectés sous le format :

```
login terminal month day hour:minute
```

L'option `-w` ajoute après le nom de connexion un caractère indiquant le statut de l'utilisateur vis à vis des messages :

— `+` : messages autorisés
— `-` : messages non autorisés
— `?` : impossible de trouver le périphérique du terminal

À partir de cette commande, on veut les informations suivantes.

Question 14.1 Le nombre d'utilisateurs connectés.

Question 14.2 Le nombre d'utilisateurs connectés autorisant l'envoi de messages.

Question 14.3 Un fichier nommé `msgUser` contenant les utilisateurs connectés autorisant l'envoi de messages, trié selon l'ordre inverse de l'ordre alphabétique des utilisateurs.

Question 14.4 Le nombre d'utilisateurs connectés autorisant l'envoi de messages tout en créant un fichier nommé `msgUserCpt` trié par ordre alphabétique inverse de ces utilisateurs.

Exercice 15 : La commande `history(1)`

La commande `history(1)` affiche la liste des dernières commandes lancées par l'utilisateur, dans l'ordre chronologique, sous la forme :

```
1 ls
2 less titi
3 pwd
4 less toto
5 ls /etc
6 ps -ejH | wc -l
```

Question 15.1 Quelle commande permet de placer le résultat de la commande `history(1)` dans un fichier `history.txt` ?

Question 15.2 À partir du fichier `history.txt` précédemment obtenu, donnez la commande qui permet d'obtenir les informations suivantes :

1. le nombre de commandes dans l'historique
2. les 5 dernières commandes utilisées sans leurs paramètres (uniquement les premières commandes)
3. la liste triée des commandes (et uniquement les commandes) avec leurs paramètres
4. le nombre de fois qu'on a utilisé la commande `less` en tant que première commande de la ligne
5. la liste des 5 commandes les plus utilisées par ordre décroissant

Question 15.3 Écrivez un script shell `execn.sh` qui prend en paramètre un chemin vers le fichier `history.txt`, puis :

— affiche le nombre de commandes qu'il contient,
— demande à l'utilisateur la saisie d'un numéro n qui doit être compris entre 1 et le nombre de commandes contenues dans le fichier, bornes incluses, jusqu'à obtenir une saisie valide,
— (ré-)exécute la commande numérotée n dans l'historique.

On prendra garde de vérifier que l'argument transmis au script est un chemin vers un fichier ordinaire existant.

Exemples d'exécution :

```
$ ./execn.sh filenoexist
File filenoexist doesn't exist or isn't a regular file
Usage : ./execn.sh history_file
$
$ ./execn.sh history.txt
Number of commands : 492
Enter the number of the command : 0
The command number must be in the interval [1 , 492]
Enter the number of the command : 493
The command number must be in the interval [1 , 492]
Enter the number of the command : 3
Debug: command: pwd
```

```
Direct execution :
/home/eric/travail/Systeme/TD/TD3
```

Question 15.4 Écrivez un script shell `search.sh` qui prend en paramètre un chemin vers le fichier `history.txt` et deux nombres n_1 et n_2, tels que $n_1 < n_2$, et qui affiche les commandes numérotées entre n_1 (inclus) et n_2 (exclu). On prendra garde à vérifier la présence et la validité des paramètres.
Exemples d'exécution :

```
$ ./search.sh history.txt 2 lm
lm isn't an integer
Usage : ./search.sh history_file N1(included) N2(excluded)
$
$ ./search.sh history.txt 490 494
The file history.txt contains only 492 commands
Usage : ./search.sh history_file N1(included) N2(excluded)
$
$ ./search.sh history.txt 0 2
The two numbers must be greater than 0 with N1 < N2
Usage : ./search.sh history_file N1(included) N2(excluded)
$
$ ./search.sh history.txt 1 1
The two numbers must be greater than 0 with N1 < N2
Usage : ./search.sh history_file N1(included) N2(excluded)
$
$ ./search.sh history.txt 491 493
491 echo 'hello world'
492 sleep 10
```

Exercice 16 : Résultats d'élection

Vous recevez l'ensemble des résultats de la dernière élection qui opposait Ulfric à Tullius dans un fichier `resultats.txt` avec le format suivant :

```
nom:bureau:zone:voix
```

où `nom` désigne le nom du candidat (Ulfric ou Tullius), `bureau` désigne le bureau de vote concerné par le résultat, `zone` est la zone où se trouve le bureau de vote, et `voix` désigne le nombre de voix réalisée par le candidat. Voici un extrait de ce fichier :

```
Ulfric:Rivebois:Blancherive:18
Tullius;Rivebois:Blancherive:10
Ulfric:Blancherive:Blancherive:42
Tullius:Blancherive:Blancherive:47
Ulfric:Pondragon:Haafingard:0
Tullius:Pondragon:Haafingard:4
Ulfric:Solitude:Haafingard:23
Tullius:Solitude:Haafingard:36
Ulfric:Vendeaume:Estemarche:53
Tullius:Vendeaume:Estemarche:11
...
```

Question 16.1 À partir du fichier `resultats.txt` et des commandes vues en cours, on veut connaître :

1. l'ensemble des résultats d'Ulfric
2. la liste des bureaux de vote
3. le nombre de zones
4. le nombre de voix de Tullius dans le bureau de Rivebois à Blancherive
5. les bureaux où Tullius a obtenu 0 voix

Question 16.2 Faire un script shell qui donne le vainqueur de l'élection, c'est-à-dire celui qui a le plus grand nombre de voix

Question 16.3 Faire un script shell qui donne le nom de celui qui a remporté le plus grand nombre de bureaux de vote (on supposera qu'il n'y a jamais d'égalité).

Exercice 17 : Départements

Chaque ligne du fichier `departements` contient les champs suivants (séparés par le caractère ' :') : l'indicatif postal du département, son nom, sa préfecture, sa région d'appartenance et sa superficie. Voici un extrait de ce fichier :

```
01:Ain:Bourg-en-Bresse:Auvergne-Rhône-Alpes:5762
02:Aisne:Laon:Hauts-de-France:7369
...
24:Dordogne:Périgueux:Nouvelle-Aquitaine:9060
25:Doubs:Besançon:Bourgogne-Franche-Comté:5234
26:Drôme:Valence:Auvergne-Rhône-Alpes:6530
27:Eure:Évreux:Normandie:6040
28:Eure-et-Loir:Chartres:Centre-Val de Loire:5880
...
```

Question 17.1 À partir du fichier `departements` et des commandes vues en cours, on veut connaître :

1. Le nombre de départements.
2. Le nombre de régions.
3. Le nombre de départements de la région Bourgogne-Franche-Comté.
4. Les régions et leur nombre de départements, classées par nombre de départements dans le sens décroissant
5. Les départements ayant une superficie comprises entre 7000 et 7999 Km2.

Question 17.2 Écrire un script shell ayant le synopsis suivant :

```
./Superficie.sh chemin_vers_fichier_departements [région]
```

L'argument `chemin_vers_fichier_departements` est obligatoire. Le script doit vérifier que le fichier existe et est lisible. Le second argument est optionnel : si il est absent, le script calcule la superficie de la France, sinon il calcule la superficie de la `région` passée en argument.

Veiller à écrire les messages sur la bonne sortie, et à respecter les conventions pour le code de retour.

Exemples d'exécution :

```
$ ./Superficie.sh
usage : ./Superficie chemin_vers_fichier_departements [région]
$ ./Superficie.sh one two three
usage : ./Superficie chemin_vers_fichier_departements [région]
$ ./Superficie.sh de
de n'existe pas ou n'est pas lisible
usage : ./Superficie chemin_vers_fichier_departements [région]
$ ./Superficie.sh departements
Superficie totale de la France : 633929 km2
$ ./Superficie.sh departements 'Centre-Val de Loire'
Superficie totale de la région Centre-Val de Loire : 39145 km2
```

```
$ ./Superficie.sh departements 'Centre-Val de Loir'
La région Centre-Val de Loir n'existe pas
$
```

Exercice 18 : Qualifications de la course

Les résultats des qualifications pour la course de dimanche est fournie sous forme d'un fichier texte `qual.txt` dont le format est le suivant :

```
vitesse:numero:pilote:ecurie:constructeur
```

où `vitesse` est la vitesse obtenue par le pilote lors de la qualification, `numero` est le numéro du pilote, `pilote` est le nom du pilote, `ecurie` est l'écurie à laquelle appartient le pilote et `constructeur` est la marque du constructeur du moteur de la voiture. Voici un extrait du fichier :

```
199:11:Denny Hamlin:Joe Gibbs Racing:Toyota
201:24:Jeff Gordon:Hendrick Motorsports:Chevrolet
197:16:Greg Biffle:Roush Fenway Racing:Ford
200:48:Jimmie Johnson:Hendrick Motorsports:Chevrolet
198:14:Tony Stewart:Stewart Haas Racing:Chrevolet
...
```

Question 18.1 À partir du fichier `qual.txt` et des commandes vues en cours, on veut connaître :

1. le nombre de pilote qui ont participé aux qualifications
2. le numéro et le nom des pilotes de l'écurie *Hendrick Motorsports*
3. la liste des 43 pilotes les plus rapides qui participeront à la course
4. le nombre de pilote pour chaque constructeur
5. le nombre d'écurie pour chaque constructeur

Question 18.2 Écrire un script shell qui permet de présenter l'ensemble des pilotes par écuries et constructeur, ordonné par leur numéro au sein de chaque écurie. Par exemple, voici un extrait du format de sortie attendu :

```
Hendrick Motorsports (Chevrolet)
  24 Jeff Gordon
  48 Jimmie Johnson
  ...
Roush Fenway Racing (Ford)
  16 Greg Biffle
  ...
```

Exercice 19 : `diff(1)` et `patch(1)`

`diff(1)` et `patch(1)` sont deux utilitaires qui permettent de générer et d'appliquer des patchs, c'est-à-dire des modifications à un code source (ou tout autre fichier au format texte).

Question 19.1 Créer un fichier `toto.txt` avec plusieurs lignes, puis copier `toto.txt` dans le fichier `titi.txt`, puis modifier une ligne du fichier `titi.txt`. Ensuite, exécuter la commande :

```
diff -u toto.txt titi.txt
```

Cette commande renvoie un patch sur la sortie standard. Les deux premières lignes indiquent les fichiers concernés : le premier est celui qui sert de base (ici `toto.txt`) et le second est celui qui sert d'objectif (ici `titi.txt`). Ensuite se trouvent les modifications qu'il faut appliquer à la base pour atteindre l'objectif :

— les lignes qui commencent par un espace ne sont pas modifiées ;
— les lignes qui commencent par un `-` indiquent qu'il faut supprimer la ligne ;
— les lignes qui commencent pas un `+` indiquent qu'il faut insérer une ligne.

Dans notre exemple, il y a une ligne à supprimer (celle que vous avez modifiée dans `toto.txt`) et une ligne à insérer (la nouvelle ligne de `titi.txt`).

L'option `-u` permet de générer le patch dans le format dit «unifié» qui est le format le plus répandu. Il existe d'autres formats de patch qui peuvent faire l'objet d'extensions à cet exercice.

Question 19.2 Sauvegarder le patch dans un fichier patch.txt

Question 19.3 Appliquer le patch au fichier `toto.txt` avec la commande :

```
patch -u toto.txt patch.txt
```

Cette commande a pour effet d'appliquer le patch sur le fichier `toto.txt`. À l'issue de cette commande, le fichier `toto.txt` est complètement identique au fichier `titi.txt`.

Question 19.4 Recommencer, mais avant d'appliquer le patch, modifier la même ligne dans le fichier `toto.txt` avec un contenu différent de celui de `titi.txt`. Que se passe-t-il ?

Question 19.5 Recommencer en ajoutant des lignes, ou en retirant des lignes, ou en modifiant des lignes à différents endroits du fichier, que ce soit avant de générer le patch ou après l'avoir généré.

Exercice 20 : 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. 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 20.1 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 20.2 Déterminer le répertoire à lister.

Question 20.3 Parcourir tous les fichiers du répertoire à lister et afficher leur nom.

Question 20.4 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]
$
```

Exercice 21 : 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 21.1 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 21.2 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)`.

Question 21.3 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 21.4 Modifier le programme pour s'arrêter si le joueur fait 6 propositions perdantes et afficher un message adéquat.

Exercice 22 : 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 .
```

Question 22.1 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 22.2 Vérifier que le paramètre est bien un répertoire, puis, si c'est le cas, entrer dans ce répertoire. Indice : `test(1)`.

Question 22.3 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 de 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 définies dans une fonction sont visibles globalement (pour le processus en cours) et donc ne sont pas uniquement locales à la fonction. Pour résoudre ce problème, il faut soit créer un sous shell (nouveau processus) à chaque appel de la fonction, soit déclarer que la variable utilisée pour itérer est locale à la fonction.
Pour créer un sous shell, on peut 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 22.4 Terminer le script avec les indications données. On veillera notamment à supprimer le message d'erreur de `rmdir(1)` quand le répertoire n'est pas vide.

Question 22.5 Bonus : supprimer les fichiers vides

Exercice 23 : Évaluation des variables en shell

On tape la commande suivante :

```
$ V=date
```

Question 23.1 Donner le résultat des commandes suivantes :

```
$ echo V
$ echo 'V'
$ echo "V"
$ echo $(V)
$ echo $V
$ echo '$V'
$ echo "$V"
$ echo $($V)
$ echo \$V
$ echo '\$V'
$ echo "\$V"
$ echo $(\$V)
$ $V
$ $($V)
```

Question 23.2 En supposant que le répertoire courant soit `/home/toto`, donner le résultat des commandes suivantes :

```
$ pwd
$ V1=pwd
$ V2=$V1
$ V3=$($V2)
$ echo $($V1)
$ echo $V2
$ echo $($V2)
$ echo $V3
$ echo $($V3)
```

Exercice 24 : Les types `union` en C

Une union est une structure de données en C dans laquelle les champs partagent le même emplacement mémoire (contrairement aux structures où les emplacements mémoires sont consécutifs). Une union se déclare de la même manière qu'une structure, avec le mot-clef `union` à la place du mot-clef `struct`.

```
union foo {
  int i;
  double d;
  char c;
};
```

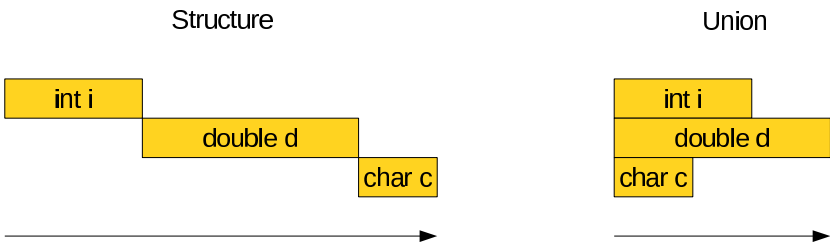

FIGURE 2 – Structure et union

La figure 2 montre l'agencement mémoire comparée d'une structure et d'une union. On remarque que la taille d'une structure est (au moins) la somme des tailles des différents champs, tandis que la taille d'une union est (au moins) le maximum de la taille de ses champs.

De manière pratique, une union ne pourra avoir qu'un seul champs ayant une valeur (soit `i`, soit `d`, soit `c` dans l'exemple). On utilisera l'un ou l'autre des champs suivant la sémantique du programme. Généralement, on associe à une union un type énuméré qui indique quel champs de l'union est utilisé. On l'appelle alors une union tagguée.

```
struct foo {
  enum { INT, DOUBLE, CHAR } tag;
  union {
    int i;
    double d;
    char c;
  } value;
};
```

Question 24.1 Définir une union d'un `int` (qu'on supposera sur 32 bits) et un tableau de 4 `char`. Quelle est la taille de cette union ?

Question 24.2 À l'aide de l'union précédente, écrire une fonction C qui permet de savoir si la machine sur laquelle s'éxécute le code est *little endian* ou *big endian* ?

Question 24.3 Dans le code suivant, en supposant qu'un `long` soit sur 64 bits, qu'est-ce qui s'affiche ? Pourquoi ?

```
union number {
  long l;
  double d;
};

int main() {
  union number n;
  n.l = 42;
  if (n.d == (double) n.l) {
    printf("Yes\n");
  } else {
    printf("No\n");
  }
  return 0;
}
```

Exercice 25 : 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 25.1 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 25.2 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 25.3 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);
```

Elle 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 25.4 Écrire un programme `readenv` qui lit et affiche les variables d'environnement suivantes : `HOME`, `LANG`, `PATH`, `PWD`, `SHELL`, `USER`. On pourra, par exemple, faire une fonction `void printenv(const char *name);` qui regarde si la variable d'environnement `name` existe et affiche son nom et sa valeur le cas échéant.

Exercice 26 : Dépassement de pile

On considère le programme suivant :

```
#include <stdio.h>

void f(int n) {
  int a[3] = { 0, 0, 0 };
  int b = 0;

  a[n] = n;

  printf("%i %i %i %i\n", a[0], a[1], a[2], b);
}

int main() {
  f(2);
  f(3);
  return 0;
}
```

Question 26.1 À votre avis, quel est l'affichage ?

Question 26.2 Expliquer ce qu'il se passe.

Question 26.3 Que faire pour éviter ce genre d'erreur ?

Exercice 27 : Arguments de la ligne de commande

Jusqu'à présent, nous n'avons pas utilisé les paramètres `argc` (*ARGument Count*) et `argv` (*ARGument Vector*) de la fonction `main`. Nous allons nous y intéresser. Pour rappel, `argv` est un tableau de `argc` chaînes de caractères dont la première indique le nom du programme.

Question 27.1 Écrire un programme `printargs` qui affiche l'ensemble des arguments passés sur la ligne de commande. On testera notamment l'utilisation des guillemets pour fournir des paramètres incluant des espaces.

Question 27.2 Écrire un programme `ints` qui prend en argument des entiers et qui en fait la somme et le produit et les affiche sur la sortie standard. On utilisera `atoi(3)` pour convertir une chaîne de caractères en entier. On utilisera le type `long` pour stocker la somme et le produit, et la séquence de contrôle `%ld` pour afficher un `long` (voir `printf(3)`).

Exercice 28 : Gestion avancée des arguments

Question 28.1 Écrire un programme `mycc` qui prend un ensemble de noms de fichier (au maximum 64) et trois options :

- `-h` qui affiche l'aide et arrête ;
- `-c` qui permet de dire qu'on veut uniquement compiler ;
- `-o output` qui permet de préciser le nom du fichier de sortie (`a.out` par défaut).

On affichera uniquement le résultat du traitement des options. Voici ce qu'on doit obtenir au final :

```
$ ./mycc
        only compile: no
        output: a.out
        inputs:
                standard input

$ ./mycc -h
Usage: mycc [-h] [-c] [-o output] [files...]

$ ./mycc -o
Error : no target specified after the '-o' option
Usage: mycc [-h] [-c] [-o output] [files...]

$ ./mycc -o test -o test2
Error: there can be only one target
Usage: mycc [-h] [-c] [-o output] [files...]

$ ./mycc -h -c
Error : no more arguments when 'h' is used
Usage: mycc [-h] [-c] [-o output] [files...]

$ ./mycc -x
Error: Option -x non supported
Usage: mycc [-h] [-c] [-o output] [files...]

$ ./mycc -c toto.c
        only compile: yes
        output: a.out
        inputs:
                file: toto.c
$ ./mycc -o toto.o toto.c -c titi.c
        only compile: yes
        output: toto.o
        inputs:
                file: toto.c
                file: titi.c
```

Exercice 29 : La commande `echo(1)`

La commande `echo(1)` affiche sur la sortie standard tous les arguments de sa ligne de commande, séparés par des espaces. Elle prend l'option `-n` qui permet de ne pas passer à la ligne à la fin.

Question 29.1 Déterminer si l'option `-n` est présente. On supposera qu'elle ne peut être qu'en première position.

Question 29.2 Afficher les arguments de la ligne de commande.

Question 29.3 Prendre en compte l'option `-n`.

Exercice 30 : Chaînes de caractères en C

La manipulation de chaînes de caractères est un élément essentiel en C et dans n'importe quel langage de programmation. Nous allons voir ici comment sont implémentés quelques fonctions de la bibliothèque standard C concernant les chaînes de caractères (`string(3)`). Voici le prototype des fonctions auxquelles nous allons nous intéresser :

```
size_t strlen(const char *s);
int strcmp(const char *s1, const char *s2);
char *strstr(const char *haystack, const char *needle);

char *strcpy(char *dest, const char *src);
char *strcat(char *dest, const char *src);
char *strdup(const char *s);
```

Question 30.1 Qu'est-ce qu'une chaîne de caractère ?

→ Fonctions d'interrogation de chaînes de caractères

Question 30.2 Implémenter `strlen(3)` qui renvoie la taille de la chaîne `s`.

Question 30.3 Implémenter `strcmp(3)` qui compare deux chaînes de caractères et renvoie -1, 0 ou 1 suivant que la première chaîne et plus petite, égale ou plus grande que la seconde (selon l'ordre lexicographique).

Question 30.4 Implémenter `strstr(3)` qui recherche une aiguille dans une botte de foin et renvoie un pointeur sur le début de la sous-chaîne cherché ou `NULL` si elle n'est pas trouvée.

→ Fonctions avec manipulation de chaînes de caractères

Question 30.5 Implémenter `strcpy(3)` qui fait une copie de `src` dans `dest`.

Question 30.6 Implémenter `strcat(3)` qui copie `src` au bout de `dest`.

Question 30.7 Implémenter `strdup(3)` qui fait une copie de `src` dans un buffer alloué via `malloc(3)`.

Exercice 31 : Interface et page de manuel

Le but de ce TP est de vous faire implémenter une interface prédéfinie en C, puis de la documenter à travers une manpage. Vous allez donc implémenter un générateur pseudo-aléatoire congruentiel linéaire.

→ Générateur pseudo-aléatoire congruentiel linéaire

Une technique pour générer des nombres pseudo-aléatoires est d'utiliser une suite récursive de la forme :

$$X_{n+1} = (a * X_n + c) \mod m$$

Où a est le multiplicateur, c l'incrément, m le module et X_0 est la graine (*seed*). Ce type de générateur, très simple, est utilisé notamment dans la `libc` dans la fonction `rand(3)` avec les paramètres $a = 1103515245$, $c = 12345$ et $m = 2^{32}$. Ces générateurs sont employés là où le besoin de hasard n'a pas besoin d'être fort statistiquement ou fort cryptographiquement. Voir la page Wikipedia «Générateur congruentiel linéaire» pour plus d'informations.

Dans cette première partie, vous devez implémenter l'interface suivante :

```
struct lcg {
  /* to be defined */
};

/* initialize the parameters of self */
void lcg_param(struct lcg *self, int mult, int incr, int mod);

/* initialize self (similar to srand()) */
void lcg_seed(struct lcg *self, int n);

/* get the next random number (similar to rand()) */
int lcg_next(struct lcg *self);

/* get the max that self can return (similar to RAND_MAX) */
int lcg_max(struct lcg *self);
```

Question 31.1 Écrire un fichier `lcg.h` contenant la structure de données et les interfaces des fonctions décrites précédemment.

Question 31.2 Implémenter les interfaces du fichier `lcg.h` dans des fichiers séparés (une fonction par fichier).

Question 31.3 Écrire un `Makefile` qui compile les sources en une bibliothèque statique `liblcg-static.a` d'une part, et en une bibliothèque dynamique `liblcg.so` d'autre part.

Question 31.4 Écrire un fichier `lcg_demo.c` avec une fonction `main` qui affiche les 10 premiers nombres aléatoires d'un générateur de paramètres $a = 137$, $c = 187$, $m = 256$, initialisé à 42. Lier le fichier à la bibliothèque dynamique `liblcg.so`. Modifier le `Makefile` pour automatiser la construction.

Question 31.5 Vérifier qu'on obtient la sortie suivante en tapant la commande `./lcg_demo` :

```
53
24
147
102
81
20
111
34
237
144
```

→ Écriture de manpage en langage `groff`

Dans cette deuxième partie, vous allez écrire la manpage (en anglais) des fonctions de l'interface à l'aide d'un langage appelé `groff(1)`. La manpage `groff_man(7)` donne toutes les indications de formatage pour une manpage (dans la section `USAGE`).

Vous pouvez vous inspirer librement des manpages `uptime(1)` et `rand(3)` que vous trouverez respectivement dans `/usr/share/man/man1/uptime.1.gz` et `/usr/share/man/man3/rand.3.gz`. Ces fichiers sont des fichiers textes compressés que vous pouvez lire avec `zmore(1)` ou décompresser dans votre répertoire local avec la commande : `gunzip -c file.gz > file`.

Pour voir votre manpage, il est inutile d'appeler directement `groff(1)`, vous pouvez utiliser `man(1)` en lui fournissant le nom de votre fichier en paramètre.

Question 31.6 Dans quelle section va se trouver votre manpage ? Quel nom va-t-elle avoir ?

Question 31.7 Écrire la manpage en mettant au moins les sections `SYNOPSIS`, `DESCRIPTION`, `RETURN VALUE` et `SEE ALSO` (qui renverra vers `rand(3)`). Voici un squelette qui peut vous servir de base :

```
.TH lcg 3 1970-01-01
.SH NAME
lcg_param, lcg_seed, lcg_next, lcg_max \- Linear congruential random generator

.SH SYNOPSIS
.B #include <lcg.h>

.BI "void lcg_seed(struct lcg *" self ", int " n );

etc.

.SH DESCRIPTION

The
.BR lcg_param ()
function ...

.SH AUTHOR
Your name <your.name@edu.univ-fcomte.fr>
```

```
.SH "SEE ALSO"
.BR rand (3)
```

Exercice 32 : 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 32.1 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 buffer statique (d'une taille de 1024 octets) ;
3. écrire les données du buffer sur la sortie standard ;
4. tant qu'il y a des données dans le fichier, aller à 2. ;
5. fermer le fichier.

Exercice 33 : La commande `tee(1)`

La commande `tee(1)` lit son entrée standard et écrit sur la sortie standard et sur un fichier.

Question 33.1 Implémenter la commande `tee(1)`.

Question 33.2 Améliorer votre code en considérant qu'il peut y avoir plusieurs fichiers à écrire sur la ligne de commande.

Exercice 34 : La commande `cmp(1)`

La commande `cmp(1)` permet de comparer deux fichiers octets par octets. Son code de retour permet de savoir si les deux fichiers sont identiques ou non. S'ils ne le sont pas, un message d'erreur est affiché et indique où se trouve l'erreur.

Question 34.1 Implémentez la commande `cmp(1)`

Exercice 35 : La commande `cmp(1)` - version 2

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

```
cmp [-i NB] file1 file2
```

La commande `cmp(1)` compare 2 fichiers octet par octet. Les deux fichiers constituent des arguments obligatoires. L'option `-i` permet d'ignorer les NB premiers octets des deux fichiers. Cette option ne peut apparaître qu'en première position. Si elle est absente, la comparaison des deux fichiers commence à la position 0.

- En cas d'erreur (synopsis non respecté, un des deux fichiers n'existe pas ou n'est pas lisible, ...), la commande affiche un message d'erreur et renvoie un code de retour égal à 2.
- Si les deux fichiers sont identiques à partir de la position NB, la commande n'affiche rien et renvoie un code de retour nul.
- Si les deux fichiers sont différents à partir de la position NB, la commande affiche la position du premier octet qui diffère d'un fichier à l'autre, et renvoie un code de retour égal à 1.

Exemples d'exécution :

```
$ echo abcdef > fic1
$ echo aBcDefg > fic2
$ ./cmp fic1 fic2
Les fichiers fic1 et fic2 sont différents --> premiere différence en position 1
$ echo $?
1
$ ./cmp -i 2 fic1 fic2
Les fichiers fic1 et fic2 sont différents --> premiere différence en position 3
$ ./cmp -i 7 fic1 fic2
Les fichiers fic1 et fic2 sont différents --> premiere différence en position 7
$ ./cmp -i 8 fic1 fic2
$ echo $?
0
```

Question 35.1 Contrôler la validité des arguments. Si ils sont valides, déterminer le nombre d'octets à ignorer.

Question 35.2 Réaliser le programme en traitant les erreurs lors des ouvertures et des fermetures de fichiers.

Exercice 36 : La commande `wc(1)`

Il s'agit implémenter la commande `wc(1)` avec deux options : `-c` (pour compter les octets) et `-l` (pour compter les lignes). Si aucun fichier n'est précisé, on utilisera l'entrée standard.

Question 36.1 Analyser la ligne de commande pour vérifier la présence des différentes options et du nom du fichier. S'il n'y a aucune option, on affichera les deux nombres (d'octets et de lignes) par défaut. Indice : `strcmp(3)`.

Question 36.2 Implémenter le comptage des octets.

Question 36.3 Implémenter le comptage des lignes. On supposera que les lignes sont terminées par `"\n"`.

Question 36.4 Comparer le résultat de votre commande avec le résultat de `wc(1)`.

Exercice 37 : La commande `cp(1)`

Dans cet exercice, nous allons réimplémenter la commande `cp(1)` dans un programme en C. Cette commande prendra deux arguments : le nom du fichier à copier et le nom de la copie. Par exemple :

```
./cp example.txt example.copy.txt
```

Question 37.1 Vérifier qu'il y a deux arguments sur la ligne de commande, sinon afficher un message sur l'erreur standard et quitter avec un code d'erreur.

Question 37.2 Ouvrir le fichier source donné en premier argument en lecture seule, et le fichier destination donné en second argument en écriture seule. Indice : «r» et «w».

Question 37.3 Copier le contenu du fichier source dans le fichier destination.

Question 37.4 Fermer les deux fichiers.

Exercice 38 : La commande `tr(1)` (*TRanslate*)

La commande `tr(1)` permet de remplacer un caractère par un autre ou un ensemble de caractères par un autre ensemble. Elle lit l'entrée standard et écrit le résultat sur la sortie standard. Par exemple :

— `tr 'a' 'A'` remplace tous les «a» par un «A».

— `tr 'abc' 'ABC'` remplace tous les «a» par un «A», tous les «b» par un «B» et tous les «c» par un «C».

Question 38.1 Vérifier que le programme a bien deux arguments et que les deux chaînes de caractères ont bien la même taille. Sinon, envoyer un message d'erreur.

Question 38.2 Définir un buffer (dans l'espace d'adressage du processus) et lire l'entrée standard.

Question 38.3 Parcourir le buffer et, pour chaque caractère, s'il est égal à un des caractères du premier argument, on le replace par le caractère correspondant du second argument.

Question 38.4 Écrire le buffer sur la sortie standard.

Question 38.5 Faire la boucle principale.

Exercice 39 : La commande `nl(1)` (*Number Lines*)

La commande `nl(1)` permet de numéroter les lignes d'un fichier en ajoutant le numéro de la ligne au début de chaque ligne. Elle lit le fichier spécifié sur la ligne de commande ou l'entrée standard s'il n'est pas spécifié. Elle écrit le résultat sur la sortie standard.

Question 39.1 Déterminer le fichier à lire.

Question 39.2 Définir un buffer et lire le fichier.

Question 39.3 Écrire le numéro de la ligne courante (1 au départ) sur la sortie standard puis parcourir le buffer à la recherche de la prochaine fin de ligne. On supposera que les lignes sont terminées par `"\n"`. Écrire la ligne sur la sortie standard.

Question 39.4 Faire la boucle principale (et fermer le fichier si besoin).

Exercice 40 : La commande `expand(1)`

La commande `expand(1)` permet de convertir les tabulations `'\t'` en espaces, en écrivant sur la sortie standard. Elle prend un argument : un nom de fichier. L'entrée standard est lue quand il n'y a aucun argument ou que l'argument est «`-`», sinon, l'argument désigne le fichier à lire. Chaque tabulation est remplacée par 4 espaces.

Question 40.1 Donner le code en C de la commande `expand(1)`.

Exercice 41 : La commande `paste(1)`

La commande `paste(1)` permet de regrouper les lignes de plusieurs fichiers, séparés par des tabulations.

Par exemple, on suppose que `names.txt` est un fichier texte qui contient les informations suivantes :

```
Mark Smith
Bobby Brown
Sue Miller
Jenny Igotit
```

et que `numbers.txt` est un autre fichier texte qui contient les informations suivantes :

```
555-1234
555-9876
555-6743
867-5309
```

En appelant `paste(1)` avec `names.txt` et `numbers.txt`, on obtient le résultat suivant :

```
$ paste names.txt numbers.txt
Mark Smith      555-1234
Bobby Brown     555-9876
Sue Miller      555-6743
Jenny Igotit    867-5309
```

Question 41.1 Implémenter la commande `paste(1)` en supposant que les fichiers sont de même taille

Question 41.2 Adapter le code pour traiter les cas où les fichiers sont de tailles différentes. On remplacera l'information manquante par une chaîne vide.

Quand on utilise l'option `-s`, la sortie de `paste(1)` est présentée de manière horizontale.

```
$ paste -s names.txt numbers.txt
Mark Smith      Bobby Brown     Sue Miller      Jenny Igotit
555-1234        555-9876        555-6734        867-5309
```

Question 41.3 Écrire un script shell `paste-s.sh` qui fait la même chose.

Exercice 42 : La commande `cut(1)`

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

La commande à implémenter prendra 2 arguments obligatoires et un argument optionnel. Les deux arguments obligatoires sont, dans cet ordre, le séparateur et le numéro du champs à sélectionner (le premier champs étant numéroté 1). L'argument optionnel sera le nom 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 `:` comme séparateur. Vous noterez que, contrairement à l'outil que vous avez utilisé, le nom des options n'apparaît pas.

Question 42.1 Déterminer le séparateur et le numéro du champs. On affichera un message d'erreur si le séparateur indiqué contient plus d'un caractère, ou si le numéro du champ est strictement inférieur à 1.

Question 42.2 Déterminer le fichier à lire. On affichera un message d'erreur si le fichier donné en paramètre ne peut pas être ouvert.

Question 42.3 Afficher le champs sélectionné sur chaque ligne. On veillera à ne pas afficher le séparateur et à prendre en compte les passages à la ligne correctement.

Question 42.4 Fermer le fichier.

Exercice 43 : 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, il 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.

Par exemple :

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

Question 43.1 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 43.2 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 43.3 Lire le fichier et afficher ses caractères imprimables suivant les spécifications données. On utilisera la fonction `isprint(3)` qui permet de savoir si un caractère est imprimable.

Question 43.4 Fermer le fichier.

Exercice 44 : La commande `fold(1)`

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

La commande `fold(1)` permet de couper les lignes trop longues en plusieurs lignes. Si une ligne dépasse n caractères, alors elle est coupée après n caractères, peu importe si on se trouve au milieu d'un mot. Par défaut, n vaut 80. Les tabulations (`\t`) comptent pour quatre espaces, les caractères backspace (`\b`) comptent pour un retour en arrière, c'est-à-dire pour un caractère en moins. La commande peut prendre une option `-w` suivi d'un nombre qui indique n. On supposera que l'option, si elle est présente, est directement après le nom de la commande. La commande prend également en paramètre le nom du fichier à lire. Si le fichier est absent, on lira l'entrée standard. Le résultat est envoyé sur la sortie standard.

Par exemple :

```
fold toto.txt
fold -w 64
fold -w 120 titi.txt
```

Question 44.1 Déterminer n, la longueur maximale d'une ligne.

Question 44.2 Déterminer le fichier à lire. En cas d'erreur à l'ouverture, afficher un message et arrêter la commande.

Question 44.3 Lire le fichier et couper les lignes trop longues en tenant compte des tabulations et des backspaces.

Question 44.4 Fermer le fichier.

Exercice 45 : 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 45.1 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 46 : Lanceur de commande

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

Question 46.1 Récupérer l'heure courante à l'aide de `gettimeofday(2)`. La fonction `gettimeofday(2)` permet de récupérer l'heure courante à l'aide d'une structure de type `timeval` :

```
struct timeval {
  time_t      tv_sec;     /* secondes */
  suseconds_t tv_usec;    /* microsecondes */
};

int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Question 46.2 Forker et exécuter la commande passée en paramètre 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 46.3 Récupérer l'heure courante puis calculer la durée (très approximative) du processus fils.

Exercice 47 : 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 bloqué (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)
- Sinon, elle renvoie le status du processus fils récupéré dans le père par `wait(2)` ou `waitpid(2)`

Question 47.1 Implémenter cette fonction (vous pouvez la nommer `my_system()` par exemple). On justifiera précisément la version de la famille `exec` utilisée à l'aide d'un commentaire dans le code.

Question 47.2 Dans la fonction `main()` (qui appelle votre fonction `my_system()`), utiliser les macros adéquates (WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG) pour afficher comment le processus fils s'est terminé (indice : `wait(2)`).

Exemples d'exécution à tester :

```
$ ./system23 "sleep 10"            --> avec ou sans ^C
$ ./system23 "ps -lH"
$ ./system23 "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 `wait(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 et pour exécuter l'interpréteur.

Exercice 48 : La commande `nohup(1)`

La commande `nohup(1)` permet d'exécuter une commande en la rendant insensible aux déconnexions, c'est-à-dire qu'elle permet d'ignorer le signal `SIGHUP` envoyé à tous les processus d'une session quand le leader de session se termine. De plus, si la sortie standard est un terminal (indice : `isatty(3)`) :

- elle est redirigée dans un fichier `nohup.out` (placé dans le répertoire de travail courant du processus),
- ou, si ce n'est pas possible, dans `$HOME/nohup.out`,
- si ce dernier fichier n'est pas non plus accessible en écriture, la commande n'est pas exécutée.

Si la sortie d'erreur standard est un terminal, elle est redirigée vers la sortie standard.

Question 48.1 Si nécesaire, ouvrir le fichier `nohup.out` avec les flags `O_RDWR | O_CREAT | O_APPEND` et le mode `S_IRUSR | S_IWUSR`.

Question 48.2 Si nécesaire, établir les redirections nécessaires

Question 48.3 Installer le gestionnaire de signal adéquat pour `SIGHUP`.

Question 48.4 Exécuter la commande passée en paramètre. On utilisera `execvp(3)`.

Exercice 49 : Processus concurrents

Des processus concurrents sont des processus qui s'exécutent en même temps. S'il n'y a pas de mécanismes de synchronisation, les actions de deux processus concurrents peuvent s'entremêler dans n'importe quel ordre. Par exemple, le programme suivant peut afficher «`12`» ou «`21`».

```
int main() {
  if (fork() == 0) {
    printf("1");
  } else {
    printf("2");
    wait(NULL);
  }
  return 0;
}
```

Question 49.1 Quels sont tous les affichages possibles du programme suivant qui contient plusieurs processus concurrents ?

```
int main() {
  printf("1");
  if (fork() == 0) {
    if (fork() == 0) {
      printf("2");
    } else {
      printf("3");
      wait(NULL);
      printf("4");
    }
    exit(0);
  } else {
    printf("5");
    wait(NULL);
  }
  printf("6");
  return 0;
}
```

Exercice 50 : Le tri par endormissement

On veut implémenter ici une commande qui effectue un tri par endormissement (*sleep sort*). La commande prend en paramètre une liste d'entier. Pour chaque entier i, on crée un processus qui s'endort pendant i secondes en utilisant la fonction `sleep(3)` puis affiche l'entier i sur la sortie standard. Ainsi, l'affichage obtenu sera trié car les processus qui auront dormi le moins longtemps seront ceux qui auront le plus petit i.

Question 50.1 Implémenter la commande décrite en langage C.

Question 50.2 Implémenter la commande décrite en shell.

Question 50.3 Quel problème potentiel voyez-vous ?

Exercice 51 : `man(1)` et processus

Nous allons ici implémenter une version très primitive de `man(1)`. Nous allons supposer que les manpages sont toutes stockées de manière compressées (via `gzip(1)`) dans le répertoire `/usr/share/man/`, chaque section ayant son propre répertoire de la forme `man?`, et que le numéro de section est obligatoirement indiqué sur la ligne de commande. La commande `man(1)` réalise alors plusieurs opérations successives :

1. Elle appelle `gunzip(1)` pour décompresser la manpage avec l'option `-c` qui permet d'envoyer le résultat sur la sortie standard.
2. Elle appelle l'utilitaire de formatage de texte `groff(1)` avec l'option `-man` qui indique que le fichier contient des directives du paquet `an.tmac` et l'option `-T utf8` qui indique que le format de sortie doit être du texte brut encodé en UTF-8.
3. Elle appelle le *pager* (`more(1)` ou `less(1)` généralement) contenu dans la variable d'environnement `PAGER`.

Ainsi, la commande `man 1 uptime` est équivalente à :

```
$ gunzip -c /usr/share/man/man1/uptime.1.gz | groff -man -T utf8 | $PAGER
```

Question 51.1 Représenter les fils d'exécution des processus à implémenter.

Question 51.2 Récupérer le numéro de section et le nom de la manpage en arguments et construire le chemin vers le fichier contenant la manpage.

On utilisera la variante suivante de la famille `exec(3)` :

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

Question 51.3 Implémenter la commande.

Exercice 52 : Implémentation en C d'une commande

Soit la commande suivante :

```
ls -R -1 path 2>&1 | wc -l
```

L'option `-1` de la commande `ls` permet d'afficher un fichier par ligne.

Question 52.1 Sachant que les redirections liées au tube sont effectuées en premier, que réalise l'*opération* `2>&1` ?

Question 52.2 Quel est le résultat de cette commande lorsque :

1. `path` désigne un répertoire ? Justifier.
2. `path` désigne un fichier régulier ? Justifier.
3. `path` désigne un fichier inexistant ? Justifier.

Question 52.3 Représenter les fils d'exécution des processus à implémenter.

Question 52.4 `path` doit être transmis en argument au programme à implémenter. Vérifier la validité du nombre d'arguments. En cas d'erreur, afficher le synopsis et sortir.

Question 52.5 Implémenter la commande sans utiliser `sh` avec l'option `-c`.

Question 52.6 Ce programme constitue-t-il une bonne solution pour remplir sa fonction ? Justifier.

Exercice 53 : Pilote de compilation

Le but de cet exercice est d'implémenter en C `mycc`, un pilote de compilation. On suppose que l'on dispose d'un ensemble d'outils qu'on va appeler à la chaîne pour produire un code objet :

- L'outil `mycpp` est un préprocesseur qui prend un fichier sur son entrée standard et qui écrit le résultat sur sa sortie standard.
- L'outil `mycc1` est un compilateur qui prend un fichier issu du préprocesseur sur son entrée standard et qui écrit un fichier assembleur sur sa sortie standard.
- L'outil `myas` est un assembleur qui prend un fichier assembleur sur son entrée standard et qui écrit un fichier objet sur sa sortie standard.

L'outil `mycc` prend en paramètre un fichier source et un fichier destination. L'appel `mycc input.c output` est équivalent à :

```
mycpp < input.c | mycc1 | myas > output
```

Question 53.1 Décrire les processus et les autres dispositifs qui vont être utilisés pour implémenter cette commande. À l'aide d'un schéma, représenter les interactions entre ces éléments.

Question 53.2 Vérifier qu'il y a deux paramètres et afficher un message d'erreur sinon.

Question 53.3 Ouvrir le fichier source (Indice : `O_RDONLY`). Créer le fichier destination.

Question 53.4 Mettre en place les processus.

Question 53.5 On veut maintenant pouvoir gérer les signaux permettant d'arrêter les processus et les transmettre aux processus créés. Préciser les signaux en question. Mettre en place une gestion des signaux adéquate.

Exercice 54 : Compilation de fichier C

L'idée de ce TD est de créer un fichier C, de le compiler et de l'exécuter.

→ Création d'un fichier C

Question 54.1 Créer un fichier nommé `bye.c`. On utilisera le mode «w+» qui crée un fichier s'il n'existe pas.

Question 54.2 Écrire dans ce fichier un programme en C qui place un gestionnaire de signal sur `SIGTERM` qui dit «Bye !» et qui se met en attente d'un signal.

Question 54.3 Fermer le fichier.

→ Compilation du fichier C

Question 54.4 Créer un processus qui va exécuter la commande de compilation : `gcc -o bye bye.c`

Question 54.5 Attendre la fin de ce processus. Que se passe-t-il si on exécute la suite sans attendre ?

→ Exécution du programme

Question 54.6 Créer un processus qui va exécuter le programme `bye` nouvellement créé.

Question 54.7 Envoyer un signal `SIGTERM` au processus `bye`.

Question 54.8 Attendre la fin du processus `bye`.

Exercice 55 : Compilations séparées parallèles

Dans le cas de gros projets, il peut être intéressant de compiler séparément et parallèlement les différents programmes sources qui le constituent.

Synopsis du programme `buildpar` :

```
buildpar output file...
```

L'argument `output` correspond au nom de l'exécutable à générer. Les arguments suivants (`file...`) correspondent aux chemins des fichiers sources du projet.
Le programme `buildpar` compile séparément et parallèlement les fichiers sources qui lui sont transmis en arguments (argv[2] et les suivants) en utilisant la commande `gcc(1)` avec les options `-g -std=c99 -Wall`. Ces dernières sont fixées en dur dans le programme.
La commande `gcc(1)` renvoie 0 en cas de succès, et 1 en cas d'échec. En cas d'échec, elle écrit un message sur sa sortie d'erreur standard. Pour ne pas mélanger les différentes sorties des commandes `gcc(1)`, leur sortie d'erreur standard sera redirigée vers le fichier `/tmp/X.err` où `X` est le PID du processus.
On supposera qu'il ne peut y avoir au maximum que 16 fichiers sources sur la ligne de commande, et que la longueur maximale d'un argument est égale 255 caractères.
Si toutes les compilations séparées réussissent, le programme `buildpar` réalise une édition de liens avec l'option `-g`, également fixée en dur dans le programme.
On veut pouvoir arrêter proprement les commandes `gcc(1)` en cours à l'aide de `CTRL+C`. Dans ce cas, il faudra que le processus père élimine proprement ses fils et supprime les fichiers `/tmp/X.err`. Il faudra gérer correctement le signal correspondant dans les questions qui suivent.

Exemples d'exécutions :

```
$ ./buildpar hull hull.c lib/geometry.c lib/algorithms.c
gcc -g -std=c99 -Wall -c -o hull.o hull.c
gcc -g -std=c99 -Wall -c -o lib/geometry.o lib/geometry.c
gcc -g -std=c99 -Wall -c -o lib/algorithms.o lib/algorithms.c
Terminaison normale de 3565 avec code de retour 1
Debug : début de la lecture de /tmp/3565.err
hull.c: Dans la fonction « main »:
hull.c:62:3: erreur : expected « , » or « ; » before « fprintf »
   fprintf(stderr, "%zu\n", count);
   ^~~~~~
Debug : fin de la lecture de /tmp/3565.err
Terminaison normale de 3566 avec code de retour 0
Terminaison normale de 3567 avec code de retour 0
$
$
$ ./buildpar hull hull.c lib/geometry.c lib/algorithms.c
gcc -g -std=c99 -Wall -c -o hull.o hull.c
gcc -g -std=c99 -Wall -c -o lib/geometry.o lib/geometry.c
gcc -g -std=c99 -Wall -c -o lib/algorithms.o lib/algorithms.c
Terminaison normale de 3595 avec code de retour 0
Terminaison normale de 3596 avec code de retour 0
Terminaison normale de 3597 avec code de retour 0
gcc -g -o hull hull.o lib/geometry.o lib/algorithms.o
lib/geometry.o : Dans la fonction « length » :
/home/eric/Exam2_16_17/lib/geometry.c:39 : référence indéfinie vers « pow »
```

```
...
collect2: erreur : ld a retourné 1 code d'état d'exécution
$
```

Question 55.1 Pourquoi l'édition de liens échoue-t-elle lors de la 2ème exécution ?

Question 55.2 Définir des constantes pour le nombre maximum de fichiers sources et la longueur maximale d'un argument.

Question 55.3

```
int print_file(const char *path, FILE *out);
```

Définir la fonction `print_file()` qui ouvre le fichier de chemin `path`, le lit et affiche son contenu dans le fichier associé au flux `out`. Elle renvoie 0 en cas de succès et -1 si l'ouverture échoue.

Question 55.4

```
void print_args(char * args[], FILE *out);
```

Définir la fonction `print_args()` qui écrit sur une ligne, dans le fichier associé au flux `out`, les chaînes de caractères contenues dans `args` en les séparant par un espace. Le dernier pointeur du tableau `args` vaut NULL.

Question 55.5 Vérifier la validité du nombre d'arguments. En cas d'erreur, afficher le synopsis et sortir. (Il n'est pas demandé de vérifier que les fichiers sources ont l'extension `.c`).

Question 55.6 Lancer en concurrence les processus permettant de compiler séparément les différents fichiers sources. Pour chaque fichier source, vous devez afficher la ligne de commande utilisée pour le compiler sur la sortie standard. Ne pas oublier de faire la redirection nécessaire pour traiter la sortie d'erreur standard (les fichiers `/tmp/X.err` seront ouverts avec les flags `O_WRONLY | O_CREAT | O_TRUNC` et le mode `0644`).

Question 55.7 Éliminer proprement les processus fils. Pour afficher comment les processus fils se sont terminés, vous utiliserez les macros `WIFEXITED(status)`, `WEXITSTATUS(status)`, `WIFSIGNALED(status)` et `WTERMSIG(status)`. Ne pas oublier, quand c'est nécessaire, de lire le fichier `/tmp/X.err` et d'afficher son contenu sur la sortie d'erreur standard. Dans tous les cas, le fichier `/tmp/X.err` devra être supprimer.

Question 55.8 Si toutes les compilations séparées réussissent, afficher la ligne de commande permettant de réaliser l'édition de liens sur la sortie standard, et lancer la.

Exercice 56 : La commande `xargs(1)`

La somme des longueurs des chaînes de caractères, constituant les arguments et l'environnement, passées à un nouveau programme via un appel de la fonction système `execve(2)` est limitée. La limite est définie par la constante `ARG_MAX` :

```
$ getconf ARG_MAX
2097152
```

Le but de cet exercice est d'implémenter en C la commande `xargs(1)`. La commande `xargs(1)` lit des arguments délimités par des espaces blancs depuis l'entrée standard, et exécute une ou plusieurs fois la commande reçue dans `argv[1]` en utilisant les arguments initiaux suivis des arguments lus depuis l'entrée standard. Le nombre d'exécutions de la commande reçue en `argv[1]` est optimisé (c'est à dire minimisé), tout en respectant la contrainte liée à la constante `ARG_MAX`.

La commande `xargs(1)` peut donc avoir une utilité quand la liste des arguments à passer à une commande est trop longue. Par exemple, la commande `rm -f *` peut générer une liste de fichiers trop longue et échouer. On utilisera alors `xargs(1)` de la manière suivante :

```
ls | xargs rm -f
```

Cette commande équivaut à appeler `rm -f` en lui passant en arguments ce qui arrive sur l'entrée standard de `xargs(1)`, c'est à dire ce qui est écrit dans le tube par `ls`.

Pour simplifier, on supposera que la commande passée à `xargs(1)` est exécutée pour chaque nouvel argument lu depuis l'entrée standard.

La constante noyau `MAX_ARG_STRLEN` fixe la longueur maximale d'un argument. Elle est égale à 131072 (32 fois la taille d'une page mémoire). On supposera également que les arguments lus sur l'entrée standard sont tous de longueur inférieure ou égale à `MAX_ARG_STRLEN`.

Question 56.1 Vérifier qu'il y a au moins un argument tranmis à `xargs` et afficher un message d'erreur sinon.

Question 56.2 Lire l'entrée standard jusqu'au prochain espace blanc pour récupérer l'argument courant (indice : `isspace(3)`).

Question 56.3 Lancer un processus en ajoutant l'argument à ceux présents sur la ligne de commande.

Question 56.4 Faire une boucle pour lire l'entrée standard jusqu'à la fin.

Question 56.5 Attendre la fin de tous les processus créés.

Exercice 57 : La fonction `sleep(3)`

La fonction `sleep(3)` permet d'endormir un processus pendant une durée en secondes indiquée en paramètre. Son prototype est :

```
unsigned int sleep(unsigned int nb_sec);
```

Question 57.1 Quel est le signal envoyé par `alarm(2)` au bout du temps indiqué en paramètre ?

Question 57.2 Proposer une implémentation en C de la fonction `sleep(3)` en utilisant `alarm(2)` ainsi que d'autres fonctions dont vous préciserez l'utilité dans le commentaire de votre code. Votre fonction renverra 0 dans tous les cas.

Exercice 58 : 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 implémenter :

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

Question 58.1 Déterminer le signal à envoyer. On supposera que l'option `-s` ne peut apparaître qu'en première position.

Question 58.2 Lancer le signal à chacun des processus donné en argument. Indice : `kill(2)`

Exercice 59 : Zombie

Le but de cet exercice est d'observer un processus zombie.

Question 59.1 Commencer le programme en affichant le PID du processus principal.

Question 59.2 Compléter le programme en permettant à l'utilisateur d'observer un processus zombie. Le père se mettra en attente d'un signal. On justifiera précisément pourquoi un zombie est créé à l'aide d'un commentaire dans le code.

Question 59.3 Modifier le programme de sorte que l'envoi au processus père du signal `SIGUSR1` conduise à terminer proprement les deux processus.

Question 59.4 Quelle commande précise permet d'envoyer le signal `SIGUSR1` au processus père ?

Exercice 60 : 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 60.1 Vérifier que la commande a bien deux paramètres et écrire un message sur l'erreur standard sinon.

Question 60.2 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.

Question 60.3 Utiliser les macros adéquates (WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG) pour afficher dans le processus père comment les processus fils se sont terminés.

Question 60.4 À votre avis, est-ce que cette méthode permet d'accélérer la copie par rapport à un seul processus ? Pourquoi ?

Question 60.5 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 61 : Signaux

Question 61.1 Écrire un programme `invincible` qui dit «boom !» quand il reçoit SIGINT et qui continue de s'exécuter. Indices : `sigaction(2)`, `pause(2)`, `signal(7)`.

Question 61.2 Faites en sorte qu'il abandonne au bout de n tentatives (valeur donnée en paramètre de la commande).

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

Question 61.3 Modifiez-le pour qu'il compte des 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 !». Indice : `alarm(2)`.

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

Exercice 62 : 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 62.1 En utilisant `rand(3)` et `RAND_MAX`, écrire une fonction C qui renvoie un `double` compris dans l'intervalle $[0, 1]$.

Correction de la question 62.1

```
double get_double() {
  return (double) rand() / (double) RAND_MAX;
}
```

Question 62.2 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 62.2

```
void compute_pi() {
  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 (n et p) 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 `srand(3)` et `getpid(2)` (de manière à ce que chaque processus ait une graine différente).

Question 62.3 Écrire une commande `biprocess_pi` qui utilise deux processus et un tube. On fixera n à 10^7.

Question 62.4 Écrire une commande `interrupted_pi` qui utilise deux processus et qui calcule tant qu'elle n'est pas interrompue (par un `CTRL+C`).

Question 62.5 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`).

Exercice 63 : `run-parts(8)`

Le but de cet exercice est d'implémenter la commande `run-parts(8)`. La commande `run-parts(8)` permet d'exécuter les scripts ou les exécutables d'un répertoire. Pour simplifier, on supposera que tous les fichiers situés dans le répertoire sont exécutables.

Question 63.1 Vérifier qu'il y a un argument et afficher un message d'erreur sinon.

Question 63.2 Parcourir le répertoire indiqué en argument.

Question 63.3 Pour chaque fichier, construire le nom complet du fichier (répertoire et nom du fichier) pour pouvoir l'exécuter.

Question 63.4 Exécuter le processus et attendre sa terminaison.

Exercice 64 : la commande `service(8)`

La commande `service(8)` permet de lancer un service, c'est-à-dire un script situé dans le répertoire `/etc/init.d/`, dans un environnement aussi prévisible que possible, en supprimant toutes les variables d'environnement et en utilisant `/` comme répertoire de travail. Le nom du service est le premier paramètre. Le second paramètre est un ordre à passer au script parmi `start`, `stop` ou `restart`. Si l'ordre est `restart`, on appellera le script deux fois : une fois avec `stop` et une fois avec `start`. La commande `service(8)` a comme code de retour le code de retour du script, ou, dans le cas d'un `restart`, le code de retour de l'ordre `start`.

```
$ service mysql start
$ service mysql restart
```

Question 64.1 Déterminer le script à exécuter et l'ordre à passer au script.

Question 64.2 Exécuter le service dans les conditions requises. On justifiera très précisément la version d'`exec` utilisée.

Question 64.3 Renvoyer le bon code d'erreur.

Exercice 65 : La commande `startpar(8)`

La commande `startpar(8)` permet de lancer plusieurs processus en concurrence. Elle est notamment utilisée lors de l'initialisation du système pour lancer des services indépendants les uns des autres en parallèle. Elle prend en paramètre un ensemble de programmes à lancer, et éventuellement une option `-a` suivi d'un argument à passer à chaque programme. D'autre part, la sortie standard et l'erreur standard de chaque processus seront redirigées vers un l'entrée d'un tube pour ne pas mélanger les différentes sorties des programmes. Il faudra donc créer un tube par programme à lancer. Après avoir lancé tous les programmes, le processus père lira séquentiellement les données en sortie des tubes (remarque : le volume de données écrites sur ses sortie standard et sortie d'erreur standard par un processus démon est généralement assez faible, il y a donc relativement peu de chance qu'un processus démon remplisse son tube). On supposera qu'il ne peut y avoir au maximum que 16 programmes sur la ligne de commande et que l'option `-a`, si elle est présente, est en première position.

Voici des exemples d'utilisation de la commande `startpar(8)` :

```
$ startpar /etc/init.d/atd /etc/init.d/crond /etc/init.d/lighttpd
$ startpar -a start /etc/init.d/cups /etc/init.d/kdm
```

Question 65.1 Quelle est la capacité d'un tube sous Linux ?

Question 65.2 Définir une constante pour le nombre maximum de programmes.

Question 65.3 Déterminer si l'option `-a` est présente et, si elle est présente, déterminer le paramètre à passer aux programmes.

Question 65.4 Déterminer une structure contenant toutes les variables qui vont permettre de traiter un programme. Dans la suite, on utilisera un tableau de cette structure pour gérer les différents processus. Indication : vous aurez besoin, au moins, du PID du processus et du descripteur qui servira à lire sa sortie standard et sa sortie d'erreur standard.

Question 65.5 Quel signal permet de savoir qu'un processus fils est terminé ? Mettre en place un gestionnaire de signal pour ce signal qui permet de terminer le processus correctement.

Question 65.6 Lancer l'ensemble des processus en concurrence. On n'oubliera pas de faire les redirections nécessaires pour traiter la sortie standard et la sortie d'erreur standard.

Question 65.7 Dans le processus père, lire séquentiellement les sorties et erreurs standard des différents programmes et les afficher.

Question 65.8 Attendre l'élimination de tous les processus zombies avant de terminer le processus père.

Exercice 66 : Fuzzing

Le test à données aléatoires (ou *fuzzing*) est une technique pour tester des logiciels. L'idée est d'injecter des données aléatoires dans les entrées d'un programme. Le but de cet exercice est de créer un programme, `fuzz`, qui va tester un autre programme, et notamment son entrée standard.

Le programme `fuzz` envoie des données textuelles aléatoires au processus testé. La commande à tester est passée en paramètre. La sortie standard du processus testé sera conservé dans un fichier `fuzz.out`. Exemples :

```
$ fuzz wc -l
$ fuzz cut -f1
```

Pour attendre la fin du processus testé, le programme `fuzz` interceptera le signal `SIGCHLD`.

Question 66.1 Ouvrir le fichier `fuzz.out` en écriture (Indice : `O_WRONLY`).

Question 66.2 Mettre en place un tube qui servira à envoyer les données sur l'entrée standard du processus.

Question 66.3 Mettre en place un gestionnaire de signal pour le ou les signaux adéquats.

Question 66.4 Lancer le processus avec la commande passée en paramètre.

Question 66.5 Envoyer 10000 caractères aléatoires sur l'entrée standard du processus. On choisira au hasard à l'aide de la fonction `rand(3)` parmi un tableau de caractères qu'on précisera et qu'on jutifiera.

Question 66.6 Fermer le tube.

Question 66.7 Attendre pendant 5 secondes la terminaison du programme. On n'utilisera pas `sleep(3)`.

Question 66.8 Si le processus n'est pas terminé à ce moment là, alors envoyer un signal pour terminer le processus sans que l'utilisateur ne puisse intercepter le signal.

Question 66.9 Afficher la manière dont s'est terminé le processus et afficher les informations adéquates. On utilisera les macros suivantes :

- `WIFEXITED(status)`, `WEXITSTATUS(status)`,
- `WIFSIGNALED(status)`, `WTERMSIG(status)`.

Exercice 67 : Segments mémoire

L'objectif est d'illustrer la présence de plusieurs segments mémoires pour chaque processus. Pour cela, vous devrez d'abord concevoir un programme qui vous permettra de scruter les adresses de différents objets en mémoire, puis vous vérifierez vos résultats.

→ Objets en mémoire

Question 67.1 Écrire un programme `memory` qui affiche les adresses de différents objets en mémoire :

- une adresse de variable globale constante (définie à l'aide de `const`)
- une adresse de variable globale non-initialisée (segment `BSS`)
- une adresse de variable de la pile («stack»)
- une adresse de variable du tas (allouée dynamiquement, «heap»)
- une adresse d'une fonction (c'est-à-dire une adresse de code exécutable)

Donner pour chaque cas l'adresse de l'objet et le nom du segment supposé. Vous pourrez éventuellement classer ces adresses dans l'ordre pour une lecture plus facile

Question 67.2 Ajouter un `getchar(3)` juste avant le `return` final, de manière à suspendre le programme. La fonction `getchar(3)` attend un caractère sur l'entrée standard : tant que vous n'appuyez pas sur une touche, le programme reste en mémoire (ce qui va permettre de le scruter) et attend.

→ Vérification

Question 67.3 Lancer le programme `memory`. Vous devriez obtenir quelque chose comme ça :

```
$ ./memory
1.(        0x400739) code
2.(        0x400880) const's
3.(        0x600bd0) bss
4.(        0xc58010) heap
5.(  0x7fff61a1eda0) stack
```

Question 67.4 Pour savoir comment sont réellement placés les différents segments dans la mémoire virtuelle, on peut se servir de `/proc` :

```
$ cat /proc/$(pidof memory)/maps
00400000-00401000 r-xp 00000000 08:02 8265740                /home/jbernard/memory
00600000-00601000 rw-p 00000000 08:02 8265740                /home/jbernard/memory
00c58000-00c79000 rw-p 00000000 00:00 0                      [heap]
7f7d50c73000-7f7d50dcb000 r-xp 00000000 08:02 5701728        /lib/libc-2.11.2.so
7f7d50dcb000-7f7d50fcb000 ---p 00158000 08:02 5701728        /lib/libc-2.11.2.so
7f7d50fcb000-7f7d50fcf000 r--p 00158000 08:02 5701728        /lib/libc-2.11.2.so
7f7d50fcf000-7f7d50fd0000 rw-p 0015c000 08:02 5701728        /lib/libc-2.11.2.so
7f7d50fd0000-7f7d50fd5000 rw-p 00000000 00:00 0
7f7d50fd5000-7f7d50ff3000 r-xp 00000000 08:02 5701865        /lib/ld-2.11.2.so
7f7d511eb000-7f7d511f2000 rw-p 00000000 00:00 0
7f7d511f2000-7f7d511f3000 r--p 0001d000 08:02 5701865        /lib/ld-2.11.2.so
7f7d511f3000-7f7d511f4000 rw-p 0001e000 08:02 5701865        /lib/ld-2.11.2.so
7f7d511f4000-7f7d511f5000 rw-p 00000000 00:00 0
7fff619ff000-7fff61a20000 rw-p 00000000 00:00 0              [stack]
7fff61b68000-7fff61b69000 r-xp 00000000 00:00 0              [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0      [vsyscall]
```

Identifier dans ce tableau les différents segments et vérifier qu'ils correspondent à vos prédictions. On analysera plus particulièrement les permissions associés à chaque segments en les justifiant.

Question 67.5 Quelle remarque pouvez-vous faire concernant les variables globales constantes ?

Question 67.6 Lire la page de manuel `end(3)` (qui ne signifie pas que ce TP est terminé). Ajouter au programme `memory` les adresses de `etext`, `edata` et `end` et recommencer la vérification.