

Carnet de Travaux Libres  
Système et programmation système  
Licence 2 Informatique

Julien BERNARD

## **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 . . . . .	4
Exercice 2 : Archives . . . . .	6
Exercice 3 : Fichiers et ligne de commande . . . . .	7
Exercice 4 : Makefile et L <sup>A</sup> T <sub>E</sub> X . . . . .	9
Exercice 5 : <code>true(1)</code> et <code>false(1)</code> . . . . .	10
Exercice 6 : Ouverture de fichier et permissions . . . . .	11
Exercice 7 : Le dossier <code>/tmp</code> . . . . .	12
Exercice 8 : Fichiers dans <code>/dev</code> . . . . .	13
Exercice 9 : Permissions spéciales . . . . .	14
Exercice 10 : Avec le fichier <code>/etc/passwd</code> . . . . .	15
Exercice 11 : Le fichier <code>group(5)</code> . . . . .	16
Exercice 12 : Le fichier <code>services(5)</code> . . . . .	17
Exercice 13 : Fichier de notes . . . . .	18
Exercice 14 : La commande <code>who(1)</code> . . . . .	19
Exercice 15 : La commande <code>history(1)</code> . . . . .	20
Exercice 16 : Résultats d'élection . . . . .	21
Exercice 17 : Qualifications de la course . . . . .	22
Exercice 18 : <code>diff(1)</code> et <code>patch(1)</code> . . . . .	23
Exercice 19 : Lister les fichiers d'un répertoire en shell . . . . .	24
Exercice 20 : Jeu du «Plus petit / Plus grand» . . . . .	25
Exercice 21 : Destruction récursive des répertoires vides . . . . .	26
Exercice 22 : Évaluation des variables en shell . . . . .	27
Exercice 23 : <code>diff(1)</code> et <code>patch(1)</code> . . . . .	28
Exercice 24 : Les types <code>union</code> en C . . . . .	29
Exercice 25 : Environnement . . . . .	31
Exercice 26 : Dépassement de pile . . . . .	32
Exercice 27 : Arguments de la ligne de commande . . . . .	33
Exercice 28 : Gestion avancée des arguments . . . . .	34
Exercice 29 : La commande <code>echo(1)</code> . . . . .	35
Exercice 30 : Chaînes de caractères en C . . . . .	36
Exercice 31 : Interface et page de manuel . . . . .	37
Exercice 32 : La commande <code>cat(1)</code> . . . . .	40
Exercice 33 : La commande <code>tee(1)</code> . . . . .	41
Exercice 34 : La commande <code>cmp(1)</code> . . . . .	42
Exercice 35 : La commande <code>wc(1)</code> . . . . .	43
Exercice 36 : La commande <code>cp(1)</code> . . . . .	44
Exercice 37 : La commande <code>tr(1)</code> ( <i>T</i> Ranslate) . . . . .	45
Exercice 38 : La commande <code>nl(1)</code> ( <i>N</i> umber <i>L</i> ines) . . . . .	46
Exercice 39 : La commande <code>expand(1)</code> . . . . .	47
Exercice 40 : La commande <code>paste(1)</code> . . . . .	48
Exercice 41 : La commande <code>cut(1)</code> . . . . .	49
Exercice 42 : La commande <code>strings(1)</code> . . . . .	50
Exercice 43 : La commande <code>fold(1)</code> . . . . .	51
Exercice 44 : Sauvegarde du PID . . . . .	52
Exercice 45 : Lanceur de commande . . . . .	53
Exercice 46 : La fonction <code>system(3)</code> . . . . .	54
Exercice 47 : La commande <code>nohup(1)</code> . . . . .	55
Exercice 48 : Processus concurrents . . . . .	56

Exercice 49 : Le tri par endormissement . . . . .	57
Exercice 50 : <code>man(1)</code> et processus . . . . .	58
Exercice 51 : Pilote de compilation . . . . .	59
Exercice 52 : Compilation de fichier C . . . . .	60
Exercice 53 : La commande <code>xargs(1)</code> . . . . .	61
Exercice 54 : La fonction <code>sleep(3)</code> . . . . .	62
Exercice 55 : La commande <code>kill(1)</code> . . . . .	63
Exercice 56 : Zombie . . . . .	64
Exercice 57 : Copie de fichiers . . . . .	65
Exercice 58 : Signaux . . . . .	66
Exercice 59 : Calcul de $\pi$ . . . . .	67
Exercice 60 : <code>run-parts(8)</code> . . . . .	69
Exercice 61 : la commande <code>service(8)</code> . . . . .	70
Exercice 62 : La commande <code>startpar(8)</code> . . . . .	71
Exercice 63 : Fuzzing . . . . .	72
Exercice 64 : Segments mémoire . . . . .	73

## 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)`, `what(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`, puis supprimer le fichier `yoda/README`.

**Question 1.13** 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 deux 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 des liens (symboliques et durs) à l'archive `inodes.tar` obtenue à la question précédente.

### Exercice 3 : Fichiers et ligne de commande

→ Arborescence et accès

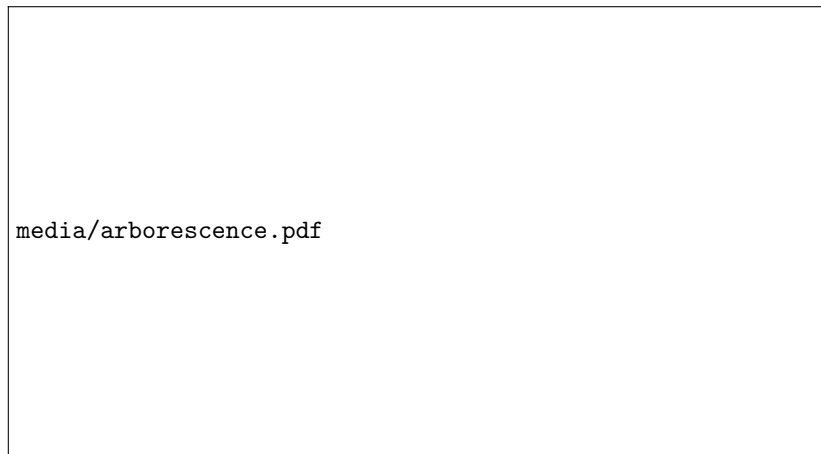


FIGURE 1 – Exemple d'arborescence

**Question 3.1** En considérant que l'on se trouve dans le répertoire L3 (fig. 1), quel est le résultat des commandes suivantes ?

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

**Question 3.2** En considérant que l'on se trouve dans le répertoire L2 (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`
3. Déplacer `hello.c` dans `.private`
4. Créer le répertoire `jbernard`
5. Effacer le répertoire `sjobs`

→ Permissions

**Question 3.3** Donnez l'équivalent octal des permissions suivantes :

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

**Question 3.4** Donnez l'équivalent symbolique des permissions suivantes :

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

**Question 3.5** Quel umask définir pour que les permissions par défaut des fichiers soit 600 ? Quelles seront les permissions par défaut des répertoires ?

**Question 3.6** En admettant que l'on soit l'utilisateur `ltorvalds` et que les fichiers et répertoires situés dans le répertoire utilisateur ait été créé avec un umask 0022 :

1. Interdire le répertoire `.private` en lecture, écriture et accès à tout le monde (autre que `ltorvalds`)
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 3.7** 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 ?

**Question 3.8** Quel est le numéro d'inode de la racine ?



## Exercice 4 : Makefile et L<sup>A</sup>T<sub>E</sub>X

Le but de ce TD est de créer un Makefile pour L<sup>A</sup>T<sub>E</sub>X. L<sup>A</sup>T<sub>E</sub>X est un logiciel de mise en forme de document à base de balises, très pratique pour écrire des TD de système par exemple.

Avec L<sup>A</sup>T<sub>E</sub>X, 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 L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X 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`.

BIB<sub>T</sub>E<sub>X</sub> est un logiciel associé à L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X 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 permission 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 auxquels appartient l'utilisateur `john`
3. La liste ordonnée des noms de groupe
4. Le deuxième utilisateur du groupe `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 :

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

**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 more titi
3 pwd
4 more toto
5 ls /etc
```

**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 15 dernières commandes utilisées (et uniquement les commandes) sans leurs paramètres
3. la liste triée des commandes (et uniquement les commandes) avec leurs paramètres
4. le nombre de fois qu'on a utilisé la commande `more` 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 nombre  $n$  et qui ré-exécute la commande numérotée  $n$  dans l'historique. On prendra garde à vérifier la présence du paramètre. Par exemple :

```
$ ./execn.sh 3
/home/jbernard
```

**Question 15.4** Écrivez un script shell `search.sh` qui prend en paramètre deux nombres  $l$  et  $u$ , tels que  $l < u$ , et qui affiche les commandes numérotées entre  $l$  (inclus) et  $u$  (exclus). On prendra garde à vérifier la présence et la validité des paramètres. Par exemple :

```
$ ./search.sh 2 4
more titi
pwd
```

## 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 : 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:Chevrolet
...
```

**Question 17.1** 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 17.2** 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 18 : diff(1) et patch(1)

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

**Question 18.1** 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 18.2** Sauvegarder le patch dans un fichier `patch.txt`

**Question 18.3** 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 18.4** 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 18.5** 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 19 : 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 paramètre, le nom du répertoire à lister (le répertoire courant par défaut).

**Question 19.1** Déterminer le répertoire à lister.

**Question 19.2** Parcourir tous les fichiers du répertoire courant et afficher leur nom. Indice : `for`

**Question 19.3** Déterminer le type de chaque fichier et afficher cette information dans le cas où c'est un fichier régulier ou un répertoire. Indice : `test(1)`



## Exercice 20 : Jeu du «Plus petit / Plus grand»

Le but est de faire un script `plus_moins.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 :

```
L'ordinateur a choisi une valeur entre 1 et 100.  
Quelle est votre proposition ? 50  
La valeur est plus petite  
Quelle est votre proposition ? 20  
La valeur est plus grande  
Quelle est votre proposition ? 35  
Gagné ! Vous avez trouvé en 3 essais
```

**Question 20.1** Définir une variable appelée `MAX` avec la valeur 100. Faire tirer à l'ordinateur une valeur au hasard entre 1 et `MAX` en utilisant `$RANDOM`

**Question 20.2** 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 20.3** 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 :

```
Perdu ! La bonne réponse était : 32
```

**Question 20.4** Modifier le programme pour s'arrêter si le joueur fait 6 propositions perdantes et afficher un message adéquat

## Exercice 21 : Destruction récursive des répertoires vides

Le but est de faire un script `rec_rmdir.sh` qui efface récursivement les répertoires vides. 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 21.1** 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 21.2** Vérifier que le paramètre est bien un répertoire, puis, si c'est le cas, entrer dans ce répertoire. Indice : `test(1)`.

**Question 21.3** 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, toutes les variables définies dans une fonction sont visible globalement (pour le processus en cours) et donc ne sont pas uniquement locale à la fonction. Pour résoudre ce problème, il est nécessaire de créer un nouveau processus puisque les variables qui n'appartiennent pas à l'environnement ne sont pas visibles pour les processus créés. Pour créer un nouveau processus, 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 entoure la fonction de manière à ce que toute la fonction s'exécute dans un nouveau processus

**Question 21.4** 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 21.5** Bonus : supprimer les fichiers vides

## Exercice 22 : Évaluation des variables en shell

On tape la commande suivante :

```
$ V=date
```

**Question 22.1** 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)
```

**Question 22.2** En supposant que le répertoire courant soit `/home/toto`, donner le résultat des commandes suivantes :

```
$ pwd
/home/foo
$ V1=pwd
$ V2=$V1
$ V3=$(V2)
$ echo $($V1)
$ echo $V2
$ echo $(V2)
$ echo $V3
$ echo $(V3)
```

## Exercice 23 : diff(1) et patch(1)

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

**Question 23.1** 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 23.2** Sauvegarder le patch dans un fichier `patch.txt`

**Question 23.3** 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`.

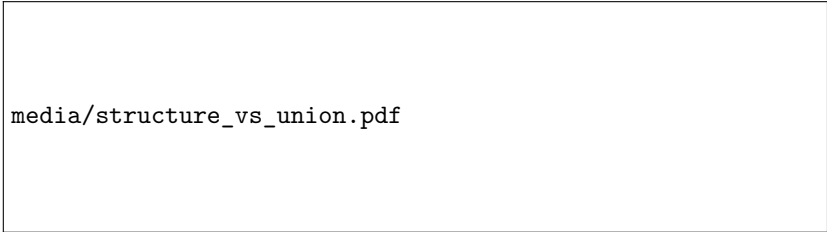
**Question 23.4** Recommencer, mais avant d'appliquer le patch, modifier la même ligne dans le fichier `toto.txt` avec un contenu différent de celui de `titi.txt`. Que se passe-t-il ?

**Question 23.5** 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 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;  
};
```



media/structure\_vs\_union.pdf

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'exécute 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

L'environnement (voir `environ(7)`) est l'ensemble des variables d'environnement. Dans un programme en C, il est possible d'accéder aux variables d'environnement via la fonction `getenv(1)`.

**Question 25.1** 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.

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

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

La variable `envp` est un tableau de chaînes de caractères. Le dernier pointeur du tableau est `NULL`, ce qui permet de connaître la fin du tableau.

**Question 25.2** Écrire un programme `env` qui lit l'ensemble des variables d'environnement et les affiche. 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 25.3** 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`.

## 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 (pour éviter de boucler), 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
      compile: no
      output: a.out
      inputs:
          standard input

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

$ ./mycc -o
Error: No output given after -o
Usage: mycc [-h] [-c] [-o output] [files...]

$ ./mycc -c toto.c
      compile: yes
      output: a.out
      inputs:
          file: toto.c

$ ./mycc -o toto.o toto.c -c titi.c
      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 est 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 cherchée 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.

**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 `wc(1)`

Il s'agit d'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 35.1** 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 35.2** Implémenter le comptage des octets.

**Question 35.3** Implémenter le comptage des lignes. On supposera que les lignes sont terminées par `"\n"`.

**Question 35.4** Comparer le résultat de votre commande avec le résultat de `wc(1)`.

### Exercice 36 : 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 36.1** 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 36.2** 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 36.3** Copier le contenu du fichier source dans le fichier destination.

**Question 36.4** Fermer les deux fichiers.

### Exercice 37 : 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 37.1** 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 37.2** Définir un buffer et lire l'entrée standard.

**Question 37.3** Parcourir le buffer et, pour chaque caractère, s'il est égal à un des caractères du premier argument, on le remplace par le caractère correspondant du second argument.

**Question 37.4** Écrire le buffer sur la sortie standard.

**Question 37.5** Faire la boucle principale et fermer le fichier.

### Exercice 38 : La commande `nl(1)` (*Number Lines*)

La commande `nl(1)` permet de numéroté 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 38.1** Déterminer le fichier à lire.

**Question 38.2** Définir un buffer et lire le fichier.

**Question 38.3** 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 38.4** Faire la boucle principale et fermer le fichier.

### Exercice 39 : 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 39.1** Donner le code en C de la commande `expand(1)`.

## Exercice 40 : 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 40.1** Implémenter la commande `paste(1)` en supposant que les fichiers sont de même taille

**Question 40.2** 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 40.3** Écrire un script shell `paste-s.sh` qui fait la même chose.



## Exercice 41 : 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 41.1** 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 41.2** Déterminer le fichier à lire. On affichera un message d'erreur si le fichier donné en paramètre ne peut pas être ouvert.

**Question 41.3** 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 41.4** Fermer le fichier.

## Exercice 42 : 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 42.1** 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 42.2** 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 42.3** 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 42.4** Fermer le fichier.

### Exercice 43 : 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 43.1** Déterminer  $n$ , la longueur maximale d'une ligne.

**Question 43.2** Déterminer le fichier à lire. En cas d'erreur à l'ouverture, afficher un message et arrêter la commande.

**Question 43.3** Lire le fichier et couper les lignes trop longues en tenant compte des tabulations et des backspaces.

**Question 43.4** Fermer le fichier.

## Exercice 44 : 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 44.1** Écrire un programme `myself` qui écrit son pid dans un fichier `myself.pid`.

## Exercice 45 : 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 45.1** 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 45.2** 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 45.3** Récupérer l'heure courante puis calculer la durée (très approximative) du processus fils.

## Exercice 46 : 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, les signaux `SIGINT` et `SIGQUIT` sont ignorés dans le père. la valeur renvoyée est le code de retour du processus.

**Question 46.1** Implémenter cette commande. On justifiera précisément la version de la famille `exec` utilisée à l'aide d'un commentaire dans le code.

## Exercice 47 : 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 par le leader de session. De plus, la sortie standard est redirigée dans un fichier `nohup.out` ou, si ce n'est pas possible, `$HOME/nohup.out`. L'erreur standard est redirigée vers la sortie standard.

**Question 47.1** Ouvrir le fichier `nohup.out` avec les flags `O_RDWR | O_CREAT | O_APPEND` et le mode `S_IRUSR | S_IWUSR`.

**Question 47.2** Établir les redirections nécessaires

**Question 47.3** Installer le gestionnaire de signal adéquat pour `SIGHUP`.

**Question 47.4** Exécuter la commande passée en paramètre. On utilisera `execvp(3)`.

## Exercice 48 : 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 48.1** 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 49 : 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 49.1** Implémenter la commande décrite en langage C.

**Question 49.2** Implémenter la commande décrite en shell.

**Question 49.3** Quel problème potentiel voyez-vous ?

## Exercice 50 : `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 50.1** Récupérer le numéro de section et le nom de la manpage en arguments et construire le chemin vers le fichier contenant la manpage.

**Question 50.2** Combien de processus vont être créés ? Combien de tubes vont être créés ?

**Question 50.3** Implémenter la commande à l'aide de processus et de tubes. On utilisera la variante suivante de la famille `exec(3)` :

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

## Exercice 51 : 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 51.1** 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 51.2** Vérifier qu'il y a deux paramètres et afficher un message d'erreur sinon.

**Question 51.3** Ouvrir le fichier source (Indice : `O_RDONLY`). Créer le fichier destination.

**Question 51.4** Mettre en place les processus.

**Question 51.5** 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 52 : 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 52.1** Créer un fichier nommé `bye.c`. On utilisera le mode «w+» qui crée un fichier s'il n'existe pas.

**Question 52.2** 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 52.3** Fermer le fichier.

### → Compilation du fichier C

**Question 52.4** Créer un processus qui va exécuter la commande de compilation : `gcc -o bye bye.c`

**Question 52.5** Attendre la fin de ce processus. Que se passe-t-il si on exécute la suite sans attendre ?

### → Exécution du programme

**Question 52.6** Créer un processus qui va exécuter le programme `bye` nouvellement créé.

**Question 52.7** Envoyer un signal `SIGTERM` au processus `bye`.

**Question 52.8** Attendre la fin du processus `bye`.

### Exercice 53 : La commande `xargs(1)`

Le but de cet exercice est d'implémenter en C la commande `xargs(1)`. La commande `xargs(1)` permet d'exécuter une même commande sur un ensemble de paramètres passés par l'entrée standard. Cette commande peut avoir une utilité quand la liste des paramètres à passer à une commande est trop longue. Par exemple, la commande `rm -f /path/*` peut générer une liste de fichiers trop longue et échouer. On utilisera alors `xargs` de la manière suivante :

```
ls /path/* | xargs rm -f
```

Cette commande équivaut à appeler `rm -f` sur chacun des paramètres qui arrivent sur l'entrée standard. On supposera que les paramètres sur l'entrée standard sont séparés par des espaces et que la taille maximale d'un paramètre est `PARAMLEN_MAX`.

**Question 53.1** Vérifier qu'il y a au moins un argument à `xargs` et afficher un message d'erreur sinon.

**Question 53.2** Lire l'entrée standard jusqu'au prochain espace pour récupérer le paramètre courant.

**Question 53.3** Lancer un processus en ajoutant le paramètre à ceux présents sur la ligne de commande.

**Question 53.4** Attendre la fin du processus.

**Question 53.5** Faire une boucle pour lire l'entrée standard jusqu'à la fin.

### Exercice 54 : 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 54.1** Quel est le signal envoyé par `alarm(2)` au bout du temps indiqué en paramètre ?

**Question 54.2** 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 55 : La commande `kill(1)`

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

**Question 55.1** Déterminer le signal à envoyer. On supposera que l'option `-s` ne peut apparaître qu'en première position.

**Question 55.2** Lancer le signal à chacun des processus donné en paramètre.  
Indice : `kill(2)`

## Exercice 56 : Zombie

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

**Question 56.1** Commencer le programme en affichant le PID du processus principal.

**Question 56.2** 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 56.3** Modifier le programme de sorte que l'envoi au processus père du signal `SIGUSR1` conduise à terminer proprement les deux processus.

**Question 56.4** Quelle commande précise permet d'envoyer le signal `SIGUSR1` au processus père ?



## Exercice 57 : 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 57.1** Vérifier que la commande a bien deux paramètres et écrire un message sur l'erreur standard sinon.

**Question 57.2** 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 57.3** À votre avis, est-ce que cette méthode permet d'accélérer la copie par rapport à un seul processus ? Pourquoi ?

**Question 57.4** On veut à présent pouvoir arrêter la copie en cours à l'aide de CTRL+C. Il faut alors interrompre le second processus et supprimer le fichier destination. Modifier le programme pour mettre en place ce comportement.

## Exercice 58 : Signaux

**Question 58.1** Écrire un programme `invincible` qui dit «boom!» quand il reçoit SIGINT et qui continue de s'exécuter. Indice : `signal(2)`, `pause(2)`, `signal(7)`.

**Question 58.2** 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 58.3** 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!», bloqué dans sa procédure de récupération. Indice : `alarm(2)`.

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

## Exercice 59 : Calcul de $\pi$

Une méthode pour obtenir une approximation de  $\pi$  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  $\pi$  est donc  $4 * \frac{p}{n}$ . On utilisera le type `long` pour  $n$  et  $p$ .

**Question 59.1** En utilisant `rand(3)` et `RAND_MAX`, écrire une fonction C qui renvoie un `double` compris dans l'intervalle  $[0, 1]$ . 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).

### Correction de la question 59.1

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

**Question 59.2** Écrire une commande `seq_pi` qui fait le calcul de  $\pi$  suivant la méthode indiquée en utilisant un seul processus. On fixera  $n$  à  $10^7$ .

### Correction de la question 59.2

```
void compute_pi() {
    srand(getpid());
    long i;
    for (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  $\pi$ , on va propager les résultats ( $n$  et  $p$ ) avec un tube entre chacun des processus, le dernier processus réalisant le calcul final de  $\pi$ .

**Question 59.3** Écrire une commande `biprocess_pi` qui utilise deux processus et un tube. On fixera  $n$  à  $10^7$ .

**Question 59.4** Écrire une commande `interrupted_pi` qui utilise deux processus et qui calcule tant qu'il n'est pas interrompu (par un `CTRL+C`).

**Question 59.5** Écrire une commande `parallel_pi` qui utilise  $q$  processus,  $q$  étant passé en paramètre de la commande et qui calcule tant qu'il n'est pas interrompu (par un `CTRL+C`). Chaque processus envoie ses données à son fils et interromp alors son fils en envoyant le signal `SIGINT`.

### Exercice 60 : `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 exécutable d'un répertoire. Pour simplifier, on supposera que tous les fichiers situés dans le répertoire sont exécutables.

**Question 60.1** Vérifier qu'il y a un argument et afficher un message d'erreur sinon.

**Question 60.2** Parcourir le répertoire indiqué en argument.

**Question 60.3** Pour chaque fichier, construire le nom complet du fichier (répertoire et nom du fichier) pour pouvoir l'exécuter.

**Question 60.4** Exécuter le processus et attendre sa terminaison.

## Exercice 61 : 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 61.1** Déterminer le script à exécuter et l'ordre à passer au script.

**Question 61.2** Exécuter le service dans les conditions requises. On justifiera très précisément la version d'`exec` utilisée.

**Question 61.3** Renvoyer le bon code d'erreur.

## Exercice 62 : 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 non-dépendants les uns des autres en parallèle. Elle prend en paramètre un ensemble de programme à lancer, et éventuellement une option `-a` suivi d'un paramètre à passer à tous les programmes. D'autre part, la sortie standard et l'erreur standard du processus sont mises en cache dans un buffer commun avant d'être affichées pour ne pas mélanger les différentes sorties des programmes. On supposera que la taille du buffer est de 10 Kio, 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 utilisations de la commande `startpar(8)` :

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

**Question 62.1** Définir deux constantes pour la taille maximale du buffer et le nombre maximum de programmes.

**Question 62.2** Déterminer si l'option `-a` est présente et, si elle est présente, déterminer le paramètre à passer aux programmes.

**Question 62.3** 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, du descripteur qui servira à lire sa sortie et son erreur standard, du buffer de réception et de variables annexes aux variables précédentes.

**Question 62.4** 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 62.5** Lancer l'ensemble des processus en concurrence. On n'oubliera pas de faire les redirections nécessaires pour traiter la sortie et l'entrée standard. Rappel : ces deux flux sont gérés avec un buffer commun dans la suite.

**Question 62.6** Dans le processus principal, lire les sorties et erreurs standard des différents programmes et les afficher si le buffer est rempli ou si la lecture s'arrête.

**Question 62.7** Fermer tous les descripteurs ouverts.

**Question 62.8** Expliquer les modifications à apporter à votre programme s'il y a plus de 16 programmes sur la ligne de commandes. Justifier.

## Exercice 63 : 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 63.1** Ouvrir le fichier **fuzz.out** en écriture (Indice : `O_WRONLY`).

**Question 63.2** Mettre en place un tube qui servira à envoyer les données sur l'entrée standard du processus.

**Question 63.3** Mettre en place un gestionnaire de signal pour le ou les signaux adéquats.

**Question 63.4** Lancer le processus avec la commande passée en paramètre.

**Question 63.5** 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 justifiera.

**Question 63.6** Fermer le tube.

**Question 63.7** Attendre pendant 5 secondes la terminaison du programme. On n'utilisera pas `sleep(3)`.

**Question 63.8** 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 63.9** 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 64 : 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 64.1** 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 64.2** 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 64.3** 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 64.4** 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]
ffffffff600000-ffffffff601000 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 64.5** Quelle remarque pouvez-vous faire concernant les variables globales constantes ?

**Question 64.6** 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.