Introduction aux systèmes d'exploitation

Cahier de TP - printemps 2021

Table des matières

1 Révisions **3**
1.1 La ligne de commande . . . 3
1.2 Compilation de petits projets . . . 4

2 Accès à des systèmes distants (mode utilisateur) **7**
2.1 Accès distant ssh . . . 7
2.2 Accès distant extérieur . . . 10

3 Manipulation de fichiers **12**
3.1 Manipulation de fichiers, liens (mode utilisateur) . . . 12
3.2 Écriture et lecture dans des fichiers (programmation) . . . 12
3.2.1 Exercices préliminaires . . . 12
3.2.2 Implémenter la commande cp en c++ . . . 13
3.2.3 Pour aller plus loin : implémenter tee . . . 14

4 Début avec les processus **15**
4.1 Gestion des processus depuis le shell . . . 15
4.2 Les débuts avec fork . . . 16

5 Processus et mini-shell **18**
5.1 Processus, signaux (1h grand maximum) . . . 18
5.2 Mini-shell . . . 18

6 Signaux, Tuyaux **21**
6.1 Signaux . . . 21
6.2 Tuyaux (*pipe*) et Signaux . . . 21

7 Sockets, et logiciels client-serveur **23**
7.1 Jouons avec nc . . . 23
7.2 Un serveur qui bégaie . . . 24
7.3 Le contrôle TP de l'année dernière . . . 25

8 Un serveur HTTP **28**
8.1 Introduction . . . 28
8.1.1 Objectif du TP . . . 28
8.1.2 Un peu de "protocole" . . . 28
8.2 Un serveur web minimal . . . 29
8.3 Gestion de la terminaison des clients, et du serveur . . . 30

TP 1

Révisions

Environnement de TP Pour les TPs il est **obligatoire** d'utiliser un système Linux et de faire le maximum de manipulations à la ligne de commande. Un système autre avec une machine virtuelle est autorisé MAIS il ne sera fait aucun support dessus.

Il est aussi **fortement recommandé** d'arrêter d'utiliser `gedit` comme éditeur de code, nous vous laissons déterminer lequel convient le mieux (mais il faut au moins qu'il indente et qu'il colore).

1.1 La ligne de commande

EXERCICE 1 ► **Le minimum vital**

Juste quelques questions pour réviser des commandes shell utiles. On aura l'occasion de revoir plus de choses en cours de semestre, mais on se concentre pour l'instant sur le strict nécessaire :

— `mkdir` de l'anglais *make directory*,

— `rmdir` pour *remove directory*,

— `ls` pour *list on screen*,

— `cd` pour *change directory*,

— `cp` pour *copy*,

— `mv` pour *move*,

— `grep` (acronyme de *global regular expression print*!),

— `cat` (pas évident, mais ça vient de *concatenate*. . .),

— `less` (successeur de `more`. . .),

— `apropos` et `man`.

Commencez en ouvrant un terminal de commandes. Ensuite :

1) Créez un répertoire de travail `LIFASR5` avec `mkdir`.

. . . .

2) Vérifiez que le répertoire à bien était créé avec `ls`.

. . . .

3) Faites de `LIFASR5` votre répertoire courant avec `cd`.

. . . .

4) Que fait la commande `pwd`? De quoi est-elle l'acronyme en anglais?

. . . .

5) Créer un sous-répertoire `TP1`.

. . . .

6) Entrez la commande `cd` (sans paramètre) : que devient votre répertoire courant? (donnez le plus de réponses possible, au moins trois!)

. . . .

7) En utilisant `cd` et la complétion automatique avec la touche « tabulation », retournez dans le répertoire `~/LIFASR5/TP1`.

. . . .

8) Avec `cp` (*copy*), copiez le fichier `/etc/passwd` dans le répertoire courant.

. . . .

9) Avec `cat`, affichez le contenu du fichier sur la sortie standard.

. . . .

10) Même chose avec `less` (touche vers le haut et vers le bas pour faire défiler le texte, `q` pour quitter).

....

11) Renommez votre copie de `passwd` en `mdp` avec `mv`.

....

12) Affichez toutes les lignes du fichier `mdp` qui contiennent la chaîne `nologin` avec `grep`.

....

13) Déplacez le fichier `mdp` dans le répertoire `/tmp/`.

....

14) Supprimez le fichier `/tmp/mdp` avec `rm`.

....

15) Que retourne la fonction C `getenv`? Répondez en consultant le page de manuel de cette fonction (`man getenv`, puis touche `q` pour quitter).

....

16) Cherchez de l'aide sur la fonction C `write`, qui permet d'écrire sur un descripteur de fichier (tentez `man write`, puis utilisez `apropos write` pour identifier la bonne page).

....

1.2 Compilation de petits projets

EXERCICE 2 ► **Compilation et Makefile**

Pour les TP qui viennent, il est important de savoir compiler et exécuter des programmes depuis la ligne de commande : sans cela, vous ne pourrez pas travailler sur la plupart des sujets. Le but de cet exercice est de rappelez comment utiliser simplement un compilateur C++ standard, `g++` pour fixer les idées (mais vous pourriez aussi utiliser `clang++`), et comment créer un `Makefile` simple pour compiler de petits projets.

1ère partie, compilation en ligne de commande :

1) Commencez par créer un fichier source `test.cpp`, qui contient un programme quelconque, par exemple :

```
#include <iostream>
using namespace std;
int main(void) {
  cout << "Hello!" << endl;
  return 0;
}
```

Compilez ce programme avec `g++ test.cpp` : quel fichier a été créé (`ls`)? Comment voyez-vous qu'il s'agit d'un fichier exécutable (`ls -l`)? Comment faire pour l'exécuter?

....

....

2) Supprimer le fichier exécutable généré à la question précédente (avec `rm`), re-compiler avec `g++ -o toto test.cpp`, puis faites un `ls -l` : que permet de faire l'option `-o`?

....

....

3) D'abord, supprimez le fichier généré à la question précédente avec `rm toto`. Quelle commande utiliser pour que le source `test.cpp` soit compilé en l'exécutable `test`?

....

4) Supprimer le fichier généré à la question précédente. Dans la catégorie des classiques du rire, on va ajouter une fonction dans un fichier séparé. Déclarez la fonction suivante dans un fichier `fonction.cpp` :

```
double cube(double x) {
  return x*x*x;
}
```

Créez le fichier d'entête `fonction.h`, qui contient le *prototype* de la fonction `cube`. Dans le `main` de `test.cpp`, ajoutez la ligne

```
cout << "Le cube de 33.0 est " << cube(33.0) << endl;
```

Tentez de générer un exécutable à partir de `test.cpp` : de quoi se plaint le compilateur ?

....

....

5) Ajouter la ligne `#include "fonction.h"` au début du fichier `test.cpp`, puis tentez à nouveau de compiler : de quoi se plaint maintenant le compilateur ? Expliquez.

....

....

6) On peut se débrouiller en créant un fichier objet à partir de `fonction.cpp` : utilisez la commande `g++ -c fonction.cpp`, et vérifiez qu'un fichier `fonction.o` a bien été créé. Pour comprendre :

— vérifiez que `fonction.o` ne contient pas le code C de la fonction cube (`cat fonction.o`).

— testez la commande `nm fonction.o` (`man nm` pour regarder la signification de `T` dans le manuel).

— désassembler le fichier objet avec `objdump -d fonction.o` : le code en langage machine, et le code en langage d'assemblage de votre fonction doivent s'afficher ! Jouons « à Champollion » : quelle instruction est utilisée pour les deux multiplications effectuées dans la fonction `cube` ?

....

Compilez maintenant `test.cpp` avec `g++ -o test test.cpp fonction.o` et vérifiez que vous obtenez bien l'exécutable `test`. Expliquez pourquoi, enfin, tout se passe bien !

....

....

....

7) Avec `rm *.o test`, supprimez les fichiers de la question précédente. On va essayer maintenant une approche plus directe, avec la commande :
`g++ -o test test.cpp fonction.cpp`
A nouveau, vérifiez que tout se passe bien, et expliquez. Quel peut être l'inconvénient de cette méthode ?

....

....

2ème partie, utilisation simplifiée d'un `Makefile` : Supprimez les fichiers `test` et `fonction.o` de la partie précédente. Un `Makefile` est un fichier qui contient des recettes (règles) pour fabriquer de nouveaux fichiers à partir de fichiers sources. Essentiellement, une recette est de la forme suivante :

```
fichier à faire: liste des fichiers nécessaires
	méthode pour fabriquer le fichier d'après les fichiers nécessaires
```

Insistons sur l'importance d'insérer le caractère de tabulation avant la méthode de fabrication (sinon, le `Makefile` plante quand on l'invoque avec `make`). Par exemple, voici une règle pour fabriquer un fichier objet :

```
fonction.o: fonction.cpp fonction.h
	g++ -c fonction.cpp #ceci est un commentaire
```

Ici, on se propose d'écrire un `Makefile` tout simple pour le programme de la partie précédente.

1) Commencez par écrire un `Makefile` de deux lignes qui contient une seule recette pour fabriquer l'exécutable `test` de la partie précédente (approche « directe »).

....

....

Exécutez votre recette en entrant que la ligne de commande `make test` : vérifiez que le programme est bien compilé. Entrez à nouveau `make test` : que constatez-vous ? Modifiez le fichier `fonction.h`, en ajoutant un commentaire sur ce que fait la fonction `cube`, puis entrez encore `make test` : que constatez-vous ?

....

....

....

2) En général, on ajoute (au moins) deux règles spéciales au `Makefile` :

— `all: <liste des fichiers à fabriquer>`. Comme ça, on peut se contenter d'entrer `make` au lieu de faire suivre la commande des noms des fichiers à fabriquer.

— une règle pour faire le nettoyage, en supprimant les exécutables, les `.o` et autres fichiers temporaires. Cela peut être :

```
clean:
	rm -f *.o *~ test
```

Ajoutez une recette `all` et une recette `clean` à votre `Makefile`. Notez que lorsqu'un `Makefile` contient plusieurs recettes, chaque règle est séparée de ses voisines par une ligne vide. Vérifiez que tout fonctionne bien.

3) Ajoutez la ligne suivante dans le `main` de `test.cpp` :

```
cout << "La racine cubique du cube de 33.0 est " << cbrt(cube(33.0)) << endl;
```

Tentez ensuite de compiler. Logiquement, il doit y avoir deux problèmes :

— le prototype de la fonction `cbrt` n'est pas connu à la compilation,

— le code de la fonction n'est pas disponible à l'édition des liens.

En vous aidant du `man cbrt` (touche `q` pour quitter le `man`), expliquez ce deux problèmes, et corrigez-les ! Bonus : localisez la bibliothèque vis-à-vis de laquelle vous faites l'édition des liens.

....

....

....

....

Pour aller plus loin : à la recherche de la librairie perdue En regardant sur le web comment dessiner des graphes, on tombe sur l'exemple de la page :

`https://www.graphviz.org/documentation/`

1) Récupérer l'adresse de l'exemple `demo.c` et télécharger l'exemple au bon endroit avec `wget`.

....

2) Essayer de compiler. On vous laisse vous débrouiller. On pourra lire la doc, utiliser find, ...

....

....

....

....

....

3) Pour l'exécution, en cas de :

```
There is no layout engine support for "a.out"
Use one of: circo dot fdp neato nop nop1 nop2 osage patchwork sfdp twopi
Error: Layout was not done.  Missing layout plugins?
```

On lancera plutôt :

```
./a.out -Kdot
```

4) Une fois que vous avez obtenu le graphe sur la sortie standard, vous pouvez obtenir le pdf avec :

```
./a.out -Kdot -Tpdf -o mygraph.pdf
```

TP 2
Accès à des systèmes distants (mode utilisateur)

2.1 Accès distant ssh

Pour ces exercices, vous allez vous connecter à l'un des serveurs utilisables au département Info dans le cadre de l'UE : commencez par aller regarder dans TOMUSS le nom du serveur que l'on vous demande d'utiliser, dans la case `lifasr5-k`. Ce nom est de la forme `lifasr5-k`, où `k` est égal à `1`, `2`, ..., ou `6`. Dans la suite, on parlera toujours de `lifasr5-k`, mais il faudra utiliser le nom que vous avez trouvé dans TOMUSS. De plus, `pNUMETU` désigne votre login étudiant.

EXERCICE 1 ▶ **Connexion `ssh` sur les machines `lifasr5-k` depuis Linux**

De but de l'exercice est de réviser l'**utilisation élémentaire** de la commande `ssh` (déjà vue en LIFASR2), ainsi que quelques autres commandes (`ls`, `cd`, `pwd`...), et de voir comment se connecter sur des machines dédiées à LIFASR5.

Lorsque vous lancez un terminal de commandes sur un système Unix, celui-ci vous permet d'accéder, *via* un shell, aux ressources de la machine sur laquelle vous êtes déjà connecté.e : on appellera cette machine la **machine locale**. Il est aussi possible d'ouvrir un terminal sur une **machine distante** au travers du réseau : dans ce cas, les commandes que vous tapez sur votre clavier, et dont vous voyez le résultat sur votre écran sont en fait exécutées sur la machine distante.

Il existe différents logiciels et différents protocoles pour faire cela. Nous utiliserons le logiciel `ssh`, car il s'agit d'une solution sécurisée très souvent utilisée en pratique. Pour pouvoir ouvrir une connexion `ssh`, il faut :

— que le le serveur `sshd` s'exécute sur la machine distante,

— pouvoir exécuter le client `ssh` sur la machine locale.

Ca tombe bien, car `sshd` tourne sur les machines `lifasr5-k`, et le client `ssh` est disponible sur les ordinateurs des salles de TP.

1) Connectez-vous en utilisant la commande `ssh pNUMETU@lifasr5-k`; il est possible que le client `ssh` vous demande de lui confirmer, à la première connexion, que vous connaissez bien l'identité de la machine `lifasr5-k`. En l'occurence, vous pouvez répondre `yes`. Ensuite, entrez le mot de passe associé à votre login pour vous connecter.

2) Comment est composée la commande que vous venez d'entrer : comment lire l'arobase (`@`) pour retenir la syntaxe ?

....

....

3) Une fois connecté à `lifasr5-k`, quel le chemin absolu de votre répertoire personnel ?

....

4) Où se trouvent les fichiers de votre compte habituel de l'université ?

....

5) Rendez-vous dans le répertoire que vous avez utilisé au TP précédent pour l'exercice portant sur la compilation en ligne de commande : vérifiez que vous pouvez toujours utiliser votre `Makefile`.

....

6) Lancez un éditeur de texte en mode graphique, par exemple `geany &` ou bien `gedit &` : que se passe-t-il et pourquoi ?

....

7) Déconnectez-vous du serveur distant en entrant `exit` ou en tapant le caractère de fin de fichier EOF (Ctrl + D). Ensuite, reconnectez-vous, en ajoutant l'option `-X` et l'option `-C` : `ssh -CX pNUMETU@lifasr5-k`. Tentez de lancer à nouveau un éditeur graphique, avec `geany &` ou `gedit &` : cette fois-ci, vous devez voir apparaître la fenêtre sur votre machine locale. Quelle est le rôle de chacune des deux options `-X` et `-C` (`man ssh`) ?

....

....

8) Avec la commande `w`, affichez la liste de tous les utilisateurs connectés sur `lifasr5-k` (`man w` pour plus de détails). Mettez-vous d'accord pour travailler avec l'un de vos voisin.e.s de TP, disons Alix, et demandez lui.elle son login pour vérifier que vous le.a voyez apparaître dans la liste. Avec `mesg yes`, donnez le droit aux autres utilisateur.ice.s connecté.e.s d'écrire sur votre terminal. Demandez ensuite à Alix de vous envoyer un message : Alix doit entrer la commande `write pNUMETU` (`pNUMETU` est votre login), entrer un message, puis taper le caractère EOF (Ctrl + D) pour terminer son message : celui-ci doit apparaître sur votre terminal. Faites aussi l'expérience symétrique : envoyez un message à Alix. Une fois cette question terminée, interdisez aux autres utilisateur.ices.s d'écrire dans votre terminal, avec `mesg no`.

Pour résumer :

— Quelle machine `lifasr5-k` vous a été attribuée ?

....

— Quelle est la syntaxe de base de la commande `ssh` ?

....

— Depuis une machine de l'université sous Linux, quelle commande utiliser pour vous y connecter en mode texte ?

....

— Et pour vous y connecter en ayant la possibilité de lancer des applications graphiques ?

....

EXERCICE 2 ► Connexion `ssh` sur les machines `lifasr5-k` depuis Windows

Pour cet exercice, on suppose que vous êtes connecté.e.s sur l'une des machines de TP de l'université, sous Windows (Windows 10 Education, plus précisément). Attention : dans les chemins sous Windows, les répertoires sont séparés par des `\` (caractère *backslash*) et non des `/` (caractère *slash*) comme sous Linux.

Avec PowerShell : vous allez dans un premier temps utiliser le programme PowerShell, qui vient avec Windows : il s'agit d'une interface en ligne de commandes, un peu comme le *bash* que vous utilisez généralement sous Linux.

Dans PowerShell, vous avez des alias vers les commandes que vous utilisez habituellement pour vous déplacer dans l'arborescence des fichiers (`ls`, `cd`, `pwd`) et les manipuler (`cp`, `mv`, `rm`, `cat`). Vous pouvez (devez!) utiliser la complétion automatique, avec la touche « tabulation », un peu comme sous Linux.

Lors d'une connexion sous Windows sur les machines des salles de TP, votre répertoire personnel (les répertoires et fichiers que vous retrouvez quelle que soit la machine à laquelle vous vous connectez) est accessible sur le volume `U:` (et aussi `W:` sur certaines machines).

1) Dans Windows, avec le navigateur de fichier, créez un dossier `tmpasr5` à la racine de votre répertoire personnel (il a donc pour chemin `U:\tmpasr5`).

2) Lancez le programme PowerShell : clic droit sur le bouton « Windows », « Exécuter », puis entrer `powershell` avant de faire « OK ».

3) Dans quel répertoire vous trouvez-vous une fois le PowerShell lancé (`pwd`) ? Placez-vous dans votre répertoire personnel : retrouvez-vous bien le répertoire `tmpasr5` ?

....

....

4) Avec la commande `ssh`, connectez-vous à `lifasr5-k`.

....

5) Une fois connecté à `lifasr5-k`, quel le chemin absolu de votre répertoire personnel ? Où se trouvent les fichiers de votre compte habituel de l'université ?

....

....

....

6) Tentez de lancer une application graphique sur `lifasr5-k`, comme par exemple `gedit` : que se passe-t-il ?

....

....

....

7) Quel éditeur de texte pouvez-vous utiliser pour éditer des fichiers dans le terminal ouvert sur `lifasr5-k` ? Assurez-vous que vous savez vous débrouiller pour éditer (charger, modifier, enregistrer) les fichiers sources du précédent TP, uniquement en mode texte.

....

....

8) Comment faire pour éditer les fichiers du TP précédent directement depuis Windows ? Faites l'expérience : est-ce que le modifications faites depuis votre machine locale sont immédiatement visibles sur `lifasr5-k` ?

....

htb!

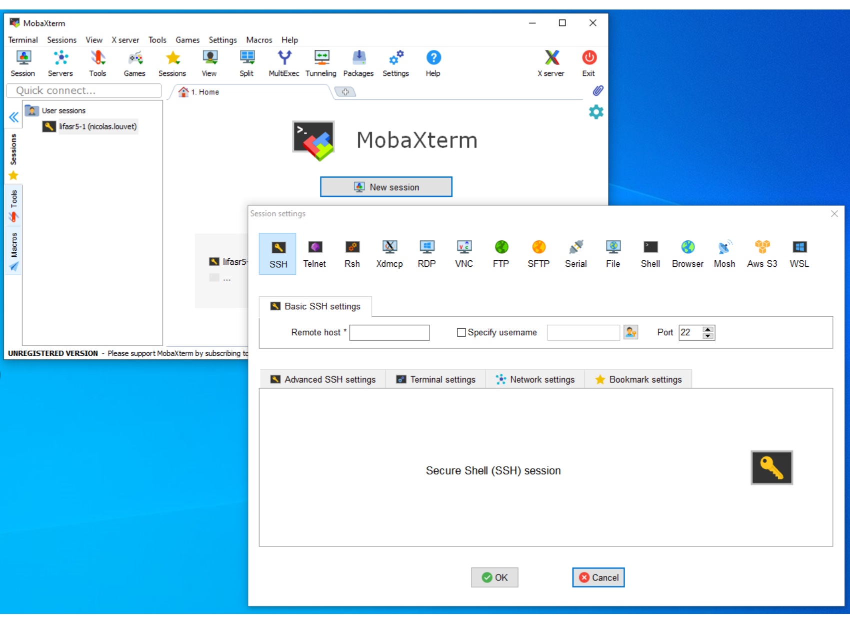

FIGURE 2.1 – Fenêtre de connexion de MobaXterm.

....

9) A l'aide d'un navigateur web, téléchargez depuis Windows le fichier PDF du 1er CM de l'UE, `cm-intro.pdf`, et enregistrez-le à la racine de votre répertoire personnel (il a donc pour chemin `U:\cm-intro.pdf`).

10) Ouvrez un second PowerShell. Avec la commande
`scp U:\cm-intro.pdf pNUMETU@lifasr5-k:~/tmpasr5`
copiez le fichier `cm-intro.pdf` vers le sous répertoire `tmpasr5` de votre répertoire personnel, sur `lifasr5-k`. En utilisant le navigateur de fichiers de Windows, vérifiez que vous retrouvez bien ce fichier dans `U:\tmpasr5`. Faites la même vérification en utilisant la connexion `ssh` sur `lifasr5-k` dont vous disposez dans le premier PowerShell.

11) Fermez la connexion `ssh` ouverte dans votre premier PowerShell avec la commande `exit`. Fermez ensuite vos deux PowerShell en utilisant à chaque fois la commande `exit`. Avec le navigateur de fichier de Windows, vous pouvez supprimer le répertoire `tmpasr5`.

Avec MobaXterm : il s'agit d'un terminal, qui peut être connecté à une machine distante *via* `ssh`, et qui a l'avantage de fournir un serveur graphique local (serveur X). L'idée est que, si vous lancez une application graphique sur la machine distante `lifas5-k`, les commandes graphiques sont envoyées sur le serveur graphique local, qui affiche donc l'interface de l'application.

1) Rendez-vous dans la section « Download » du site `https://mobaxterm.mobatek.net/`, et téléchargez « MobaXterm Home Edition v12.4 (**Portable edition**). » Désarchivez le fichier `MobaXterm_Personal_12.4.exe` (l'extension `.exe` n'est pas visible dans le navigateur de fichiers) à la racine de votre répertoire personnel, puis lancez l'exécutable. Si le « pare-feu Windows Defender » vous informe qu'il a bloqué certaines fonctionnalité de `xwin_mobax` sur tous les réseaux publics, privés et avec domaine, faites simplement « Annuler. »

2) Dans le menu *Sessions*, faites ensuite *New sessions* : une fenêtre intitulée *Session settings* apparaît (voir Figure 2.1). Sélectionnez une session SSH, en indiquant :

— *Remote host* : `lifasr5-k`,

— *Specify username* : `pNUMETU`,

— *Port* : 22 (port habituel de SSH).

Vous pouvez ensuite lancer la connection avec le bouton « OK. » Il est possible que le client vous demande de lui confirmer, à la première connexion, que vous connaissez bien l'identité de la machine `lifasr5-k`. En l'occurence, vous pouvez répondre `yes`. Dans le terminal qui s'ouvre, entrez votre login, et votre mot de passe : vous êtes maintenant connecté sur `lifasr5-k`.

3) Vérifiez, une fois de plus, que vous retrouvez bien votre compte personnel! Editez les fichiers du TP précédents LIFASR5 à l'aide d'un éditeur de texte en mode graphique (`geany &` ou `gedit &` par exemple).

4) Logiquement, vous avez le cours `cm-intro.pdf` qui traîne à la racine de votre répertoire personnel (il ne vous a pas été demandé de la supprimé à la fin de la partie précédente) : comment faire pour l'ouvrir ce fichier ?

....

....

5) Pour vous déconnecter proprement : `exit`. Vous pouve maintenant effacer le fichier `cm-intro.pdf` qui se trouve à la racine de votre répertoire personnel.

Remarque : Lors de prochains TP, si d'aventure vous vous retrouvez avec un ordinateur sur lequel vous n'arrivez pas à vous connecter sous Fedora, vous pouvez essayer de rebooter sous Windows, et si vous arrivez à vous loguer, de travailler tout de même sur le TP en vous connectant à `lifasr5-k` grâce à MobaXterm.

Avec Putty : `Putty` est un programme (distribué sous licence MIT), et dont des versions précompilées pour Windows sont disponibles ; il est déjà installé sous Windows sur les machines des salles de TP, mais si vous souhaitez l'utiliser sur un ordinateur personnel, il est disponible ici : `https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html`

1) Lancer le programme `Putty` : vous avez habituellement un raccourci sur le bureau, et vous pouvez aussi le trouver dans le menu *Démarrer*.

2) Dans la fenêtre de démarrage de `Putty` (voir Figure 2.2), vous devez entrer les informations nécessaires à votre connexion. Comme vous devez vous connecter en `ssh` sur `lifasr5-k` :
   - — entrez `lifasr5-k` dans le champ *Host Name*,
   - — vérifiez que le protocole sélectionné est bien *SSH*,
   - — vérifiez que le *Port* sélectionné est bien le port 22 (c'est le port TCP prévu pour les serveurs `ssh`).

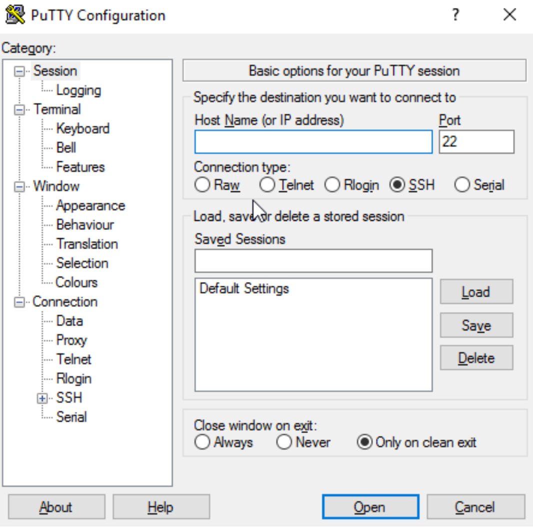

FIGURE 2.2 – Fenêtre de connexion de Putty.

Notez que vous pouvez sauvegarder ces informations en vue d'une prochaine utilisation, en entrant un nom de votre choix dans le champ `Saved Sessions`, puis en cliquant sur `Save` : vous pourrez ultérieurement recharger les informations en cliquant sur ce même nom, puis en cliquant sur `Load`.

3) Une fois que vos informations de connexion sont prêtes, vous pouvez cliquer sur *Open*. Dans le terminal qui s'ouvre, entrez votre login, et votre mot de passe : vous êtes maintenant connecté.e sur `lifasr5-k`.

2.2 Accès distant extérieur

EXERCICE 3 ▶ **(Pour aller plus loin) Et de l'extérieur alors ?**

De "l'extérieur de l'université" [1], vous avez accès à vos comptes, par exemple en vous connectant sur la machine `linuxetu.univ-lyo` accessible de l'extérieur du réseau Lyon1.

Cet exercice est à réaliser sur une machine personnelle de chez vous ou alors à partir du réseau wifi universitaire.

1. Connectez-vous en ssh à la machine `linuxetu.univ-lyon1.fr`. Observez.
2. Copiez des fichiers de votre compte vers chez vous.

1. par exemple vous êtes sur vos ordinateurs personnels en salle de TP sur un wifi quelconque

Et encore plus loin! On n'a pas tout vu, les plus avancés d'entre vous pourront aller chercher comment "monter" leur répertoire universitaire sur leur machine personnelle à distance, pour travailler avec les mêmes fichiers que leur compte universitaire. Vous pouvez aussi utiliser le protocole *ftp* pour récupérer vos fichiers ... et donc vous n'avez plus d'excuse pour ne pas pouvoir rendre un TP à partir de chez vous...

TP 3
Manipulation de fichiers

La commande du jour Glissons ici trois façons d'accéder à l'historique du shell :

— flèche du haut;

— la commande `history`;

— (Ctrl + R) : commencez par taper le début de la commande recherchée, puis utilisez Ctrl + R et Ctrl + S pour parcourir les commandes correspondantes dans l'historique.

3.1 Manipulation de fichiers, liens (mode utilisateur)

EXERCICE 1 ► **Liens physiques et symboliques**

Expérimentez les liens en Linux/UNIX :

1. Créez un fichier `foo` contenant la ligne `foo is foo`, puis affichez son numéro d'inode (voir le man de `ls`);
2. Créez une copie `bar` de ce fichier avec `cp`;
3. Créez un lien physique puis un lien symbolique vers le fichier `foo` (commande `ln`);
4. Affichez le contenu de ces quatre fichiers;
5. Comparez les inodes de ces quatre fichiers et expliquez;
6. Supprimez le fichier `foo` et regardez le contenu des trois fichiers restants;
7. Créez un nouveau fichier `foo` contenant la ligne `foo is not so foo`;
8. Consultez à nouveau les inodes des quatre fichiers et leur contenu. Expliquez.

3.2 Écriture et lecture dans des fichiers (programmation)

On commencera par aller chercher l'archive contenant les squelettes de code à remplir sur la page web du cours :

— copier l'adresse dans le presse-papier;

— se placer dans un terminal à un bon endroit (`~/LIFASR5/TP3/` semble une bonne idée);

— utiliser `wget`;

— Désarchivez avec :

```
tar xvzf lifasr5_tp3.tgz
```

3.2.1 Exercices préliminaires

EXERCICE 2 ► **Paramètres de la ligne de commande**

En C/C++, on peut récupérer les paramètres passés de la ligne de commande au programme *via* les paramètres de la fonction `main`, dont le prototype peut être :

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

ou encore

```
int main(int argc, char *argv[]);
```

Dans les deux cas, `argc` donne ne nombre d'arguments entrés sur la ligne de commande **plus un** (le nom de l'exécutable compte pour un). `argv` est un tableau de chaînes de caractères :

— `argv[0]` contient le nom de l'exécutable,

— `argv[1]`, ..., `argv[argc]` contiennent chacun des arguments passés à la ligne de commande.

On vous fournit un squelette de code appelé `paraldc.cpp` que vous compilerez en un binaire `paraldc` :

1) Comment se comporte le programme fourni ? Testez avec différents arguments dans la ligne de commande :

```
$ ./paraldc
$ ./paraldc toto
$ ./paralcd titi toto 42
```

2) Modifiez le programme pour qu'il n'accepte que les lignes de commande avec 1 argument :

```
$ ./paraldc titi 65
Il faut un argument et un seul à la commande.
```

EXERCICE 3 ▶ **Une commande** `wc`

Le but est d'écrire un petit programme qui compte le nombre d'octets que contient un fichier.

1) Complétez pour cela le programme `mywc.cpp` (fourni dans l'archive venant avec le TP).

```
// ...
int main(int argc, char *argv[]) {
  int fd; // descripteur de fichier
  char c; // servira ici de buffer
  int nbrd, nbbytes;

  if(argc < 2) {
    print_usage(argv[0]);
    return -1;
  }
  // TODO HERE
  // open file in read only mode + ...

  return 0;
}
```

Vous le compilerez avec `g++ -Wall mywc.cpp -o mywc`. Testez ensuite avec `./mywc onefilename`; vous comparez votre résultat avec celui de `wc -c onefilename`

2) Que se passe-t-il si vous appelez votre programme avec comme paramètre le nom d'un fichier qui n'existe pas ? Mettez en place une gestion de l'erreur, avec un message adéquat (`man open`, `man errno`, `man perror`) :

```
if (fd < 0) {...
 if (errno == ...)
}
```

3.2.2 Implémenter la commande `cp` en c++

EXERCICE 4 ▶ **Un programme mycp**

Le but de cet exercice, long, est d'expérimenter avec les appels systèmes `read()` et `write()`. On va écrire un programme similaire à la commande `cp` pour copier des fichiers réguliers.

Les fichiers `mycp.cpp` et `in.pdf` sont disponibles dans l'archive fournie.

Copie simple (sans erreur) : On écrit un programme permettant de copier un fichier, que l'on améliorera par la suite.

1) On vous fournit le code source `mycp.cpp`; utilisez `g++ -o mycp mycp.cpp -Wall` pour obtenir l'exécutable `mycp`.

2) Commencez par vous faire un `Makefile` simple pour compiler le programme (une seule règle `all`), et effacer l'exécutable (règle `clean`). À la compilation, vous allez avoir des *warnings* et **c'est tout à fait normal** : dans le code, il y a des variables déclarées que vous n'utiliserez que plus tard.

3) Intéressez-vous au source `mycp.cpp`, et expliquez le comportement du programme (`man open` peut vous aider).

4) Donnez une première version simple de la copie pour laquelle on ne copie que les `LEN` premiers caractères, où `LEN` est une constante déjà définie dans le programme, et égale à 20. On utilisera un unique appel à `read()`, suivi d'un unique appel à `write()`. Dans cette question, on estime que le fichier comporte plus de `LEN` caractères : expérimentez par exemple avec un fichier `in.txt` qui contient une phrase d'au moins 20 caractères.

5) Que se passe-t-il si le fichier contient strictement moins de `LEN` caractères (expérimenter!) ? Faites en sorte de retourner du programme avec un code d'erreur dans ce cas, et testez. Il peut être utile de consulter la page de manuel de `read()`.

Copie d'un plus gros fichier :

1. Téléchargez dans votre répertoire de travail le fichier `in.pdf`, qui va nous servir de fichier de test pour le programme `mycp`. Quelle est la taille de ce fichier en octets (`ls -l`) ? Jetez un œil au contenu du fichier également (avec `evince in.pdf` ou `xpdf in.pdf` par exemple). Testez `./mycp in.pdf out.pdf` : quel est l'effet produit sur `out.pdf` (`ls -l` et `evince`) ?
2. Consultez la page de manuel de `read` : que signifie une valeur de retour non nulle de `read()` ? Dans quel cas sait-on que l'on a atteint la fin du fichier ? Dans quel cas doit-on considérer qu'il y a une erreur, et qu'il faut définitivement abandonner ?

3. Consultez la page de manuel de `write` : doit-on forcément considérer qu'un appel va permettre d'écrire le nombre d'octets passé en paramètre? Dans quel cas doit-on considérer qu'il y a une erreur, et qu'il faut définitivement abandonner?
4. Complétez le programme `mycp.cpp` de façon à copier tout le contenu du fichier d'entrée dans le fichier de destination. Pour cela, inspirez vous du « brouillon » suivant :

```
char buf[LEN]; // Buffer à utiliser pour les lectures/écritures.
char *p;       // Pointeur pour avancer dans le tampon buf.
int nbrd;      // Nombre d'octets lus par read.
int nbwr;      // Nombre d'octets que l'on arrive à écrire en une fois.
int nbrem;     // Nombre d'octets restants à écrire dans le tampon buf.
do {
  // Lire (read) au plus LEN caractères sur fdin en les rangeant dans buf.
  // Soit nbrd le nombre de caractères lus par read dans fdin.
  // Prendre en compte une possible erreur lors du read.
  nbrem = nbrd;
  p = buf;
  while(nbrem > 0) { // Il reste des octets à écrire.
    // On tente d'écrire les nbrem octets restants à écrire
    // (à partir de l'adresse p) vers fdout.
    // Soit nbwr le nombre d'octets que l'on réussi à écrire.
    // En cas d'erreur lors du write terminer le programme en retournant 1.
    // Mettre à jour nbrem en fonction de nbwr.
    // Mettre à jour p pour avancer dans le tampon.
  }
} while(nbrd > 0);
```

   Utilisez `man read` et `man write` pour bien vérifier les paramètres de `read` et de `write`. N'oubliez pas de bien gérer les cas d'erreurs de `read`, et d'afficher les messages correspondant au code présent dans `errno` (à l'aide de `strerror()` par exemple). Tous les messages d'erreurs doivent être envoyés vers la sortie d'erreur standard (utilisez `cerr`).
5. Testez votre programme avec `./mycp in.pdf out.pdf`; assurez-vous que le fichier copié fait bien la même taille en octets que le fichier initial (`ls -l`). Assurez-vous également que les deux fichiers contiennent bien le même document avec un lecteur de pdf (`evince out.pdf` par exemple).
6. Pourquoi ne pas avoir copié le fichier octet par octet? En d'autres termes, pourquoi s'est-on embêté à utiliser un buffer `buf`?

3.2.3 Pour aller plus loin : implémenter tee

EXERCICE 5 ▶ **Commande** `tee`
En utilisant tee (man tee) :

1. Écrire le résultat de la commande `ls` sur la sortie standard et dans un fichier en même temps.
2. Écrire le résultat de la commande `ls` dans trois fichiers de noms différents.
3. Écrire le résultat de la commande `ls` et dans le même temps, chercher un fichier précis dans la liste rendue par `ls`.

EXERCICE 6 ▶ **Programme** `mytee`

Sur le modèle des exercices précédents :

1. Programmer en C un clone du programme `tee` qui lit sur l'entrée standard et écrit sur la sortie standard, qui se lancera avec `./mytee`.
2. Ajouter l'écriture dans un fichier passé en paramètre.
3. Implémenter l'option `-a` de `tee`.

N'oubliez pas de tester les valeurs de retour des appels système pour vérifier que tout s'est bien passé!

TP 4
Début avec les processus

On fera attention à la différence entre le chiffre `1` et la lettre `l`. La section 4.1 ne doit pas prendre plus d'une heure.

4.1 Gestion des processus depuis le shell

EXERCICE 1 ► **Découverte des processus**

Un processus est un programme en cours d'exécution sur le système. L'un des rôles du système d'exploitation est de permettre à un ensemble de processus de progresser dans leur exécution en même temps sur votre système : il doit partager les ressources disponibles (processeurs, mémoire, périphériques) entre les différents processus. Ainsi, à chaque instant de nombreux processus sont soit en cours d'exécution, soit bloqués en attente d'exécution, mais vous avez toujours l'impression qu'ils s'exécutent en « même temps ».

1) Par défaut, la commande `ps` liste les processus rattachés au terminal (`TTY`) dans lequel elle est lancée. Ouvrez un terminal, et entrez la commande `ps` : parmi les informations listées, interprétez les colonnes `CMD`, `PID` et `TTY`.

....

....

Dans la suite, on va considérer un exemple particulier de processus dont l'exécution “se voit à l'oeil nu”, c'est `xclock`. Évidemment ce n'est qu'un exemple. . . .

— Si ce logiciel n'est pas disponible sur votre machine personnelle vous pouvez l'installer (paquet `x11-apps`).
— Si ce logiciel n'est pas disponible sur les machines de TP, vous vous connecterez avec `ssh -X` sur une des machines `lifasr5-xx` ($xx \in \{1,6\}$). Il vous faut deux terminaux (le plus simple est de faire deux connections).

2) Lancez deux fois la commande `xclock -update 1 &`, puis entrez à nouveau `ps` : qu'observez-vous ?

....

....

3) Maintenant, entrez la commande `bash`, puis lancez à nouveau une horloge à l'arrière plan avec `xclock -update 1 &`. Utilisez maintenant la commande `ps -l` pour afficher plus d'informations. Comment interprétez-vous la colonne `PPID` ?

....

....

4) Dans le résultat de la commande `ps -l` précédente, comment interprétez-vous la colonne `UID` ? Pour vous aider, vous pouvez aussi vous intéresser au résultat de la commande `id`.

....

....

5) Vous avez dû comprendre que les processus sont organisés d'une façon arborescente. Pour la visualiser, vous pouvez utiliser `pstree` : à l'aide de `ps`, repérer le `PID` du premier processus `bash` lancé dans votre terminal ; on note n ce `PID` ; entrez la commande `pstree -p` n. Que constatez vous ?

....

....

6) Maintenant, ouvrez un *nouveau* terminal, puis entrez la commande `ps` : vous ne retrouvez pas dans la liste les processus que vous aviez lancés dans le premier terminal. . . Comment afficher la liste de tous les processus que vous avez lancés sur le système ?

....

7) Comment afficher tous les processus lancés sur le système ?

....

8) Fermez tous ces terminaux utilisés pour l'exercice (clic sur la croix). Cela supprime les processus `xclock` lancés et les éventuelles connections ssh. On fera “plus propre” dans l'exercice suivant.

4.2 Les débuts avec fork

EXERCICE 2 ► **Les débuts avec fork()**

La fonction `fork()` permet à un processus, dit « père », de se dupliquer : à la suite d'un appel réussi à `fork()`, le système comporte un nouveau processus, dit « fils », qui est une copie du père. **Il est important de comprendre qu'après l'appel à fork(), le code écrit est *commun* au père et au fils.**

1) Consultez le manuel de la fonction `fork()`. Quelles valeurs retourne-t-elle ?

2) Consultez le manuel des fonctions `getpid()` et `getppid()`. Quelles valeurs retournent-t-elles ?

3) On considère le source `unfils.cpp` (que vous devez télécharger sur la page de l'UE) suivant :

```
10  int main(void) {
11    int pid;
12    int a = 0;
13
14    cout << "Je suis le père, de PID " << getpid()
15         << ". Je vais créer un fils..." << endl << flush;
16
17    pid = fork();
18
19    if(pid == -1) {
20      cerr << "Erreur, aucun fils n'a été créé : " << strerror(errno)
21           << "." << endl << flush;
22      return 1;
23    }
24
25    if(pid > 0) {
26      cout << "Je suis le père, de PID " << getpid()
27           << ". J'ai un fils dont le PID est " << pid << "." << endl << flush;
28    }
29    else {
30      cout << "Je suis le fils, de PID " << getpid() << ". "
31           << " Mon père a pour PID " << getppid()
32           << "." << endl << flush;
33    }
34
35    cout << "Je suis le processus de PID " << getpid() << ". "
36         << "Mon père a le PID " << getppid() << "."
37         << endl << flush;
38
39    if(pid > 0) {
40      cout << "Je suis le père, de PID " << getpid() << ". "
41           << "Je vais me terminer..." << endl << flush;
42    }
43    else {
44      cout << "Je suis le fils, de PID " << getpid()
45           << ". Mon père a pour PID " << getppid() << ". "
46           << "Je vais me terminer..." << endl << flush;
47    }
48
49    cout << "Variable a : " << a << endl << flush;
50
51    return 0;
52  }
```

Compilez avec `g++ -Wall -o unfils unfils.cpp` ce programme, et exécutez-le plusieurs fois **dans un premier terminal**. Ouvrez un **second terminal**, qui vous servira pour « observer » les procesus que vous lancez (débrouillez vous pour pouvoir afficher les deux terminaux en même temps sur votre écran).

4) Dans le programme, comment fait-on pour différencier le code qui va être exécuté uniquement par le père, et celui qui va être exécuté spécifiquement par le fils ?

....

....

5) L'affichage des lignes 35 à 37 est effectué par le père et par le fils : pourquoi ?

....

....

6) Quels processus exécutent le `return 0` de la ligne 51 ?

....

7) Quel est le PID du bash dans lequel vous exécutez votre programme (`ps`) ?

. . . .

8) Quel est le processus dont le PID est retourné par `getppid()` pour le père ?

. . . .

9) En utilisant `sleep(60)`, modifiez le programme de façon à ce que le père se termine avant que le fils n'affiche son dernier message : quel est le père du processus fils lorsqu'il affiche ce dernier message ? Utilisez pour répondre, depuis votre terminal « d'observation », la commande `ps -l PID_DU_FILS` pour obtenir le PID du père, puis cherchez le nom de ce processus avec `ps -a`.

. . . .

. . . .

. . . .

10) Modifiez le programme de façon à ce que le père se termine longtemps après que le fils n'affiche son dernier message (en utilisant `sleep(60)` par exemple) : à l'aide de `ps -l`, observez quel est l'état du processus fils avant que son père se termine.

11) Modifiez le programme pour que le père attende, avec l'appel système `waitpid(...)` (que nous avons vu en cours, et sur lequel vous pouvez vous documenter avec `man waitpid`) que son fils se termine avant de se terminer lui-même. **Remarquer en particulier que vu que le père dispose du pid du fils, qui est la valeur retournée par l'appel à** `fork` **pour lui, il peut sans souci attendre le changement d'état de ce processus avec** `waitpid(piddufils,NULL,0)` Faites en sorte que le fils reste en sommeil 60 secondes avant de se terminer. Vérifiez que le père attend bien que son fils termine : pour cela, utilisez dans votre terminal « d'observation » la commande `pstree -p PID_DU_BASH`, pour afficher l'arborescence des processus en partant du bash dans lequel vous avez lancé votre programme.

12) Si le fils modifie le contenu de la variable `a`, est-ce que cette modification va être visible pour le père ? Inversement, si le père modifie le contenu de la variable `a`, est-ce que cette modification va être visible pour le fils ? Expérimentez pour répondre à ces questions.

. . . .

. . . .

TP 5
Processus et mini-shell

Ce TP dure trois heures, et il est demandé de travailler effectivement 3 heures.

5.1 Processus, signaux (1h grand maximum)

EXERCICE 1 ▶ **Signaux en ligne de commande**

Les signaux sont une manière simple de communiquer avec les processus, sans passer par les entrée et sortie standards. On expérimente ici simplement pour pratiquer les bases. Un signal est un code entier `sig`, que l'on peut envoyer à un processus identifié par son `pid` : la commande `kill -sig pid` envoie le signal `sig` au processus `pid`. En pratique, on n'utilise pas les codes entiers, mais des noms plus faciles à retenir :

— `TERM` pour demander aimablement au processus de se terminer : certains processus peuvent l'ignorer.

— `KILL` pour demander énergiquement au processus de se terminer : aucun processus ne peut l'ignorer.

— `STOP` pour demander au processus de s'arrêter.

— `CONT` pour demander au processus de reprendre son exécution.

Par exemple, pour envoyer le signal `CONT` au processus 69007, la commande à utiliser est `kill -CONT 69007`. On ne va travailler qu'avec ces quatre signaux. En fait, il en existe une (très) longue liste : tapez `kill -l` pour afficher toute la liste.

Ouvrez deux terminaux. Dans l'un, lancer `xclock -update 1` (sans éperluette - `&`). Dans l'autre terminal, à l'aide de la commande `ps -e | grep xclock` (il y a aussi plus direct, mais pas forcément facile d'y penser : `pgrep xclock`), déterminez le `pid` du processus `xclock` que vous venez de lancer, et notez-le.

Depuis le terminal qui n'est pas monopolisé par `xclock`, effectuez les actions suivantes.

1) Envoyez le signal `STOP` à `xclock` : qu'observez-vous ?

....

2) Envoyez le signal `CONT` à `xclock` : qu'observez-vous ?

....

3) Envoyez le signal `TERM` à `xclock` : qu'observez-vous ?

....

4) Faites les mêmes manipulations avec `gedit` : lorsque vous envoyez `STOP` à `gedit`, que constatez-vous ?

....

....

5) A votre avis, quand dans un terminal vous tapez `Ctrl+C` pour fermer un processus, que se passe-t-il ?

....

5.2 Mini-shell

Prérequis d'installation. Téléchargez *dans votre répertoire de travail* pour l'UE l'archive `mini-shell.tar.gz` à partir du site de l'UE. Désarchivez-la (avec `tar xvzf mini-shell.tar.gz` par exemple) dans un répertoire judicieusement nommé.

Dans cet exercice comme au TP précédent, on utilise le programme `xclock`, qui peut éventuellement ne pas exister sur vos machines de TP.

- vous pouvez vous connecter sur les machines `lifasr5-xx` comme au TP précédent. Votre binaire compilé en local (dans les machines de TP du Nautibus) doit pouvoir être exécuté directement sur ces machines, pourquoi ?
- vous pouvez essayer d'utiliser le script `clock.py` fourni dans l'archive ;
- vous pouvez utiliser une autre commande qui lance une fenêtre graphique, comme par exemple `gedit` ou bien `parole fichier.m` (`fichier.mp3` est un mp3 que vous aurez téléchargé) ;
- vous pouvez essayer le programme `clock` fourni, avec `xterm -e ./clock` ou `gnome-terminal -- ./clock` pour le lancer dans un nouveau terminal ;

- vous pouvez l'installer (machines perso).

EXERCICE 2 ▸ Le mini-shell

Dans le répertoire produit par l'archive, vous devez voir un sous-répertoire nommé `mini-shell`. Dans ce répertoire, vous trouverez le fichier `shell.cpp` dans lequel vous allez travailler, ainsi qu'un `Makefile` pour générer l'exécutable nommé `shell`.

Le shell ("le programme du Terminal") est un programme qui lit une commande sur l'entrée standard et l'exécute. L'objectif est de reproduire cette fonctionnalité. *Grosso modo*, vous pouvez imaginer pour l'instant que le fonctionnement d'un shell suit l'algorithme suivant :

```
 1: fini ← False
 2: tant que non fini faire
 3:     Afficher une invite
 4:     Lire une commande
 5:     Analyser la ligne de commande
 6:     si la commande est quit alors
 7:         fini ← True
 8:     sinon
 9:         si la commande se termine par '&' alors
10:             Exécuter la commande à l'arrière-plan
11:         sinon
12:             Exécuter la commande au premier plan
13:         fin si
14:     fin si
15: fin tant que
```

L'une des premières difficultés est d'analyser la ligne de commande. Dans le programme fourni, cette difficulté est escamotée : le code effectue une analyse basique [1], qui décompose la ligne de commande dans un format qui sera utilisable par l'appel système `execvp`, et sort de la boucle si la commande est « `quit` ».

Il vous reste donc à gérer correctement l'exécution des commandes. En l'état, le programme qui vous est fourni se contante d'afficher « `L'exécution d'une commande n'est pas encore implantée.` » à chaque fois que vous entrez une commande à l'invite. Mais vous pouvez quand même taper `quit` pour quitter `:D`

Vous allez devoir modifier le source `shell.cpp`, surtout dans la partie « `Traitement de la ligne de commande` », en suivant les « `TODO` ». Suivez les consignes de l'énoncé, et aidez-vous aussi des commentaires présents dans le code.

1) Compilez, testez le programme. *Pour l'instant, la commande de compilation vous donne deux* warning *de type "unused variables", c'est normal!*

2) Modifiez le programme façon à ce que votre shell permette d'exécuter une commande simple en premier plan. Cela se passe au niveau du commentaire `TODO 1` (le commentaire `TODO 2` est là aussi, mais c'est pour la question suivante). Vous utiliserez pour cela l'appel système `execvp()`, avec pour paramètres ceux fournis par `splitToChar`. Assurez-vous que :

- — la commande lancée par `execvp` soit bien un fils du shell (utiliser `fork()`),
- — le shell attend bien que son fils se termine (utiliser un `waitpid()` bloquant),
- — le shell continue d'exister après la fin de la commande,
- — si la commande lancée est erronée, le shell le signale et retourne dans un état convenable.

Pour la gestion des erreurs, vous pouvez utiliser deux fonctions fournies :

- — `void msg_error(const std::string &msg = "");` qui affiche l'éventuel message passé en argument, puis le message d'erreur donné par errno, le tout sur std : :cerr.
- — `void exit_error(const std::string &msg = "");` qui appelle `msg_error()` avec l'éventuel message passé en argument, puis termine le processus appelant avec `exit(EXIT_FAILURE)`.

Les fonctions `msg_error()` et `exit_error()` affichent leurs messages en commençant par `##` : il vous est recommandé de faire la même chose dans tout le TP, de façon à bien distinguer les affichages produits par le shell de ceux produits par les commandes que vous lancez :

Voici deux exemples de comportements attendus :

```
Entrez une commande > ls
## Lecture de la commande "ls"
## Lancement de ls
clock    clock.py  Makefile.etu prev   shell.cpp
clock.cpp  Makefile  mini-shell.tar.gz shell
```

1. En compilation, on appelle ceci analyse lexicale et syntaxique, et il existe des outils pour cela, que vous verrez en M1.

```
Entrez une commande > pouet
## Lecture de la commande "pouet"
## Lancement de pouet
## ERREUR : lors de l'exécution de la commande, No such file or directory
```

3) Modifiez votre shell pour pouvoir exécuter les commandes au premier plan ou en tâche de fond (en fonction de la variable booléenne `attend`) :

```
Entrez une commande > xclock -update 1 &
## Lecture de la commande "xclock -update 1 &"
## Ligne de commandes lancée en tâche de fond.
Entrez une commande > ## Lancement de xclock -update 1
```

Pour l'instant, contentez-vous d'utiliser un `waitpid()` bloquant de façon à ce que le shell attende la fin de son fils si la commande est lancée au premier plan; lorsqu'elle est lancée à l'arrière-plan, le shell n'attend pas pour revenir à l'invite de commandes. Les modifications doivent avoir lieu au niveau du commentaire `TODO 2`.

4) Testez votre shell dans deux situations :

— lancez `xclock -update 1` au premier plan; dans un autre terminal, vérifiez que cette commande est bien fille de votre shell (`ps -ef`); fermez `xclock`; dans l'autre terminal, vérifiez que le processus correspondant a bien disparu du système (`ps -ef`).

— lancez `xclock -update 1` à l'arrière-plan; vérifiez que cette commande est bien fille de votre shell; fermez `xclock` : quel est l'état du processus `xclock` (`ps -elf` pour avoir aussi l'état)?

5) Pour gérer (un peu) plus proprement la fin des tâches lancées à l'arrière-plan, une solution est de mettre en place dans le shell un gestionnaire pour le signal `SIGCHLD`. Terminez l'installation du gestionnaire `traitement_signal_fils()` dans le `main()`, au niveau du commentaire `TODO 3`. Recompilez votre shell, et vérifiez que votre gestionnaire de signal à bien été mis en place (pour tester, mettez un processus en arrière plan et le fermer violemment.)

6) Complétez le gestionnaire de signal `traitement_signal_fils()`, de façon à ce que, lorsqu'un des fils du processus courant se termine, le gestionnaire prenne en compte la fin de ce processus avec `waitpid()`. De plus, si le PID du processus qui vient de se terminer est le même que celui qui se trouve dans la variable globale `pid_attendu`, alors le gestionnaire mettra cette variable à 0 (on utilisera plus tard cette variable).

7) Testez votre shell comme précédemment, en lançant puis en fermant un processus, d'abord en tâche de fond, puis au premier plan : quel problème se produit-il quand un fils lancé au premier plan se termine?

8) Modifiez le programme de façon à ce que seul le gestionnaire du signal `SIGCHLD` intervienne pour prendre en compte la fin d'une tâche lancée au premier plan. Vous utiliserez l'appel système `pause()`, qui permet de suspendre le processus courant en l'attente d'un signal. En utilisant la variable globale `pid_attendu`, insérer dans le gestionnaire de signal `traitement_signal_fils()` un test pour vous assurer que le processus qui se termine est bien celui que vous attendiez (avec un message d'erreur dans le cas contraire).

9) Maintenant, on veut ajouter au shell la possibilité d'exécuter deux commandes liées par un tuyau (rappelons que l'on dit *pipe* en australien), par exemple `ls -l | less`. Pour cela, vous allez compléter la partie « cas où il y a exactement un '|' sur la ligne de commandes« du code. Dans ce cas, la ligne de commandes est de la forme `com1 | com2` : le shell doit d'abord créer un tuyau `p` avec l'appel système `pipe()`, puis deux fils :

— dans le premier fils (pour `com1`), `STDOUT_FILENO` est remplacé par une copie du descripteur de fichier `p[1]` (descripteur pour l'écriture dans le tuyau),

— dans le deuxième fils (pour `com2`, `STDIN_FILENO` est remplacé par une copie du descripteur de fichier `p[0]` (descripteur pour la lecture dans le tuyau).

Ainsi, tout ce que le premier fils écrit sur sa sortie standard (`STDOUT_FILENO`) est en fait écrit dans le tuyau (`p[1]`), et tout ce que le second fils lit sur son entrée standard (`STDIN_FILENO`) est lu depuis le tuyau (`p[0]`).

Pour qu'un descripteur de fichier `newfd` soit remplacé par une copie d'un descripteur de fichier `oldfd`, il faut utiliser l'appel système suivant :

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

En cas d'erreur, l'appel retourne -1 et `errno` prend une valeur adéquat; en cas de succès, l'appel retourne un entier positif ou nul (qui est `newfd`, mais on s'en fiche).

Dans chaque fils, l'extrémité du tuyau qui n'est pas utilisée doit être fermée avec `close`. Comme précédemment, `execvp` est utilisé pour lancer chaque commande.

Tout cela doit se passer au niveau du `TODO 5`. Il faudra vous assurer que les deux commandes se terminent proprement (pas de processus zombie), qu'elles soient lancées à l'avant ou à l'arrière-plan.

10) *Pour aller plus loin. . .* Ajoutez la possibilité d'exécuter une commande avec redirection de la sortie standard vers un fichier (par exemple, `ls -l > sortie.txt`), puis la possiblité de gérer les autres redirections (>>, <), ou encore d'exécuter des commandes avec plusieurs tuyaux (par exemple, `ls -l | grep UCL82 | wc -l`).

TP 6
Signaux, Tuyaux

Avant-propos On fournit sur la page web du cours une archive contenant les fichiers nécessaires à ce TP, ainsi qu'un Makefile.

6.1 Signaux

EXERCICE 1 ▶ **Signal périodique**

Vous devez ajouter un timer au programme `signal_temps.cpp` (dans l'archive du TP). Ce dernier fait une boucle infinie dans laquelle vous lancerez le signal SIGALRM toutes les secondes. Le signal déclenchera l'appel de la fonction `top` (une fois par seconde).

Pour cela vous utiliserez les fonctions POSIX suivantes :

— `unsigned int alarm(unsigned int s)` qui permet d'envoyer un signal au bout de `s` secondes,

— L'appel système `pause` (lire le manuel). Cet appel système est obsolète, on devrait utiliser `sigsuspend`, mais tant pis.

— La structure de donnée `struct sigaction` et son appel système associé `sigaction`.

Vous avez un exemple de tel gestionnaire de signal dans la correction du TP mini-shell, sur la page web du cours.

6.2 Tuyaux (*pipe*) et Signaux

Dans un exercice de TD « processus et communications par tubes », nous avons insisté sur la nécessité de *fermer dès que possible les descripteurs de fichiers qui ne sont pas utilisés par le processus courant.* Les fichiers nécessaires à cet exercice sont dans l'archive fournie.

EXERCICE 2 ▶ **Bloquons!**

Nous vous fournissons un programme `desc_blocage.cpp` qui crée deux processus, le processus père écrit dans un tube pour envoyer des données à son fils, et le fils lit à partir du tube et affiche seulement ceux qui sont multiples de 3. On rappelle que *tant qu'il reste un écrivain sur le tube, même s'il est vide, l'appel à read est bloquant.*

1) Décommentez le « `close(p[1]);` » du père, en laissant la même ligne commentée pour le fils, compilez, puis testez : expliquez le comportement du programme (regarder du côté du fils).

....

....

....

2) Re-commentez le « `close(p[1]);` » du processus père, et décommentez la même ligne du côté du fils; compilez, puis testez : expliquez le comportement du programme.

....

....

....

3) Décommentez « `close(p[1]);` » pour le père et pour le fils, et vérifiez que les deux processus se terminent normalement.

....

EXERCICE 3 ▶ **Signaux et tuyaux**

Cette fois on utilise le fichier `desc_sigpipe.cpp`. Dans la page `man 7 pipe`, on peut lire :

```
If all file descriptors referring to the write end of a pipe have been closed, then an attempt to read from the pipe
will see end-of-file (read will return 0). If all file descriptors referring to the read end of a pipe have been closed,
then a write will cause a SIGPIPE signal to be generated for the calling process.
```

Dans le programme précédent, nous avons utilisé la valeur de retour 0 pour arrêter le travail du processus fils lorsque le processus père ferme son descripteur de fichier en écriture sur le pipe. Maintenant, nous allons tester le deuxième comportement indiqué par la page du manuel : s'il n'y a plus de lecteur potentiel sur le pipe, le processus qui tente d'écrire dessus reçoit le signal SIGPIPE. Nous allons utiliser un gestionnaire de signal pour cela et l'appel `sigaction()` pour l'installer.

1) Quelle est la valeur entière du signal `SIGPIPE` (commande `kill -l`)?

....

2) Observez le code source :

- quel est le nom du gestionnaire de signal pour `SIGPIPE` que l'on installe dans le programme?
- par quel message doit se terminer le processus père?
- le processus fils?

....

....

3) Lancez plusieurs fois le programme `desc_sigpipe` :

- comment se termine le processus fils?
- comment se termine le processus père?
- la suite des entiers affichés est-elle toujours la même? Pourquoi?
- le numéro du signal affiché est-il bien celui de la question précédente?
- quel processus reçoit le signal `SIGPIPE`?

....

....

....

....

4) Observez à nouveau le code source :

- expliquez plus en détail comment se termine le programme?
- pour quels processus le nouveau gestionnaire de signal est-il installé?

....

....

5) Modifier le programme de façon à ce que tous les descripteurs de fichiers soient fermés proprement lorsqu'on reçoit le signal.

TP 7

Sockets, et logiciels client-serveur

Troisième TP non présenciel, il est recommandé d'avoir une solution pour faire le TP : machine personnelle sous linux, machine virtuelle, ... avant le début du TP mardi 31/03.

7.1 Jouons avec nc

EXERCICE 1 ▶ **Netcat**

`netcat`, en abrégé `nc` est une commande qui permet de créer des connexions réseau en mode TCP (aussi en mode UDP, mais cela ne nous concerne pas dans ce cours). La commande permet de créer une socket de dialogue vers un serveur, puis d'envoyer et recevoir des données via cette socket. Elle permet également de créer facilement un serveur, en mettant en place une socket d'écoute sur un certain port. Le but de l'exercice est de comprendre comment utiliser (de façon assez élémentaire) la commande `nc`, qui nous sera utile lors des TP suivants de programmation en C/C++.

Il existe (au moins) deux variantes de commande `nc` en circulation sur les différentes distributions GNU/Linux ou sur Mac :

- la variante **OpenBSD**, installée par exemple avec Ubuntu (l'aide affichée par `nc -h` commence par `OpenBSD netcat`);
- la variante **GNU**, installée par exemple sur Arch Linux ou sur Mac (l'aide donnée par `nc -h` commence par `GNU netcat`).

Les commandes sont données ci-dessous pour la variante **OpenBSD**. Si vous utilisez la variante **GNU, il ne faut pas utiliser l'option -N** pour forcer la fermeture de la socket après l'envoi de `EOF` : cette option n'existe pas, mais de tout façon c'est le comportement GNU par défaut! **De toute façon, il est conseillé de lire le manuel de la commande (man nc) sur votre machine.**

1) Depuis un terminal, vous allez créer un serveur TCP en mettant `nc` à l'écoute sur un certain port, par exemple 8084; pour cela, utilisez la commande :

```
nc -l 8084
```

L'option `-l` (pour *listen*) permet d'indiquer que l'on veut créer une socket d'écoute; comme on ne précise rien, cette socket sera à l'écoute sur toutes les interfaces de la machine. Dans un autre terminal, connectez un client au serveur qui se trouve à l'écoute sur la machine locale (`localhost`), sur le port 8084, avec la commande :

```
nc -N localhost 8084
```

L'option `-N`[1] permet d'indiquer que l'on veut que la socket soit fermée après l'envoi du caractère de fin de fichier `EOF` (obtenu en tapant Ctrl + D). Que se passe-t-il quand vous entrez une ligne, côté client ou côté serveur?

....

....

Expliquez le fonctionnement de ce client-serveur : comment les données sont-elles lues, envoyées, reçues et écrites?

....

....

....

Que se passe-t-il quand vous tapez Ctrl + D côté client?

....

....

....

2) Mettez en place un serveur prêt à recevoir des octets et à les afficher sur la sortie standard :

```
nc -l 8084
```

Comment faire, à l'aide de `echo` et `nc -N`, pour envoyer la chaîne de caractères `"hello"` au serveur, et pourquoi?

....

....

3) En vous inspirant de la question précédente, mettez en place un serveur pour recevoir un fichier, et tentez de lui envoyer un fichier binaire, `tp3.pdf` par exemple.

1. Rappelons que l'option `-N` est absente dans la version GNU de `nc`, sous Mac notamment, mais que le comportement voulu est automatique.

....

....

Vérifiez bien que votre fichier est toujours lisible une fois reçu !

4) **2020 : à distance, ça va être compliqué de faire cet exercice, mais je le laisse pour info**. Si vous êtes connecté.e sur une machine des salles de TP : déterminez le nom `petitnom` de votre machine sur le réseau : faites `hostname -a`, ou regardez le nom qui est écrit dessus; son nom est alors *le-nom-écrit-dessus*`.univ-lyon1.fr`). Mettez vous d'accord avec un étudiant connecté sur une autre machine des salles de TP :
   - l'un crée un serveur avec `nc -l 8084`,
   - l'autre se connecte à ce serveur avec `nc -N petitnom 8084`,

   et reproduisez l'expérience de la première question.

Dans les exemples précédents, c'est le client qui a mis fin à la connexion (grâce à l'option `-N`). Mais côté serveur, il ne semble pas aussi facile de fermer la connexion une fois que toutes les données sont envoyées. Nous ne tenterons donc pas d'établir un serveur permettant d'envoyer des données avec `nc`. Mais vous savez maintenant tester facilement le fonctionnement d'un serveur ou d'un client de votre cru avec `nc`.

7.2 Un serveur qui bégaie

L'archive fournie pour le TP contient un répertoire `echoserv_etu` contenant :

- les sources de la bibliothèque `socklib`, `socklib.h` et `socklib.cpp`.
- le source `echoserv.cpp` dans lequel vous allez travailler.

Le but est d'écrire un serveur `echoserv` qui se met en attente d'une connexion en mode texte. Dans les grandes lignes, lorqu'un client établit une connexion :

- le serveur doit commencer par répondre « `Bonjour !` » au client;
- puis se mettre à lire *ligne-par-ligne* le texte qu'il reçoit de la part du client;
- ensuite, il doit afficher chaque ligne lue sur sa sortie standard, entre guillements droits;
- si la ligne lue est `quit`, alors il dit “au revoir” au client, ferme la connexion, et se termine;
- sinon, il envoie au client « `Vous avez envoyé "..."` »;
- puis le serveur doit se remettre en attente de lecture de la ligne suivante.

Évidemment, il y a un peu de travail avant d'en arriver là ! **Encore une fois, on écrit le serveur.**

EXERCICE 2 ▶ **Lecture dans une socket**

Dans cet exercice, on va écrire une fonction permettant de lire sur une socket jusqu'à ce qu'un *délimiteur* soit rencontré : les lignes lues sont séparées par ce caractère; ce sera par défaut le retour à la ligne `'\n'`, mais cela pourrait être un autre caractère. Le prototype de la fonction sera :

```
int recv_line(int s, std::string &line, char c = '\n');
```

On suppose que lors de l'appel à cette fonction, `s` est une socket de dialogue, avec un hôte à l'autre bout qui envoie des caractères; `c` désigne le caractère de fin de chaîne, qui sera stocké dans la chaîne lue dans `line`. La fonction retourne le nombre de caractères lus en cas de succès, 0 si la socket a déjà été fermée par le client lorsque la fonction est appelée, -1 en cas d'échec.

1) On vous fournit un Makefile. Tester que tout compile bien chez vous. Initialement, il y a des *warnings* à la compilation : *c'est normal, vous n'avez pas à vous en occuper pour l'instant.*

2) Une fois le serveur compilé, lancez-le avec `./echoserv 8085` (par exemple), puis tentez de vous connecter depuis un autre terminal avec `nc localhost 8085` ou bien `telnet localhost 8085` : logiquement, le serveur doit simplement vous répondre « `Bonjour !` » puis fermer la connexion.

3) Dans `echoservserv.cpp`, complétez la fonction `recv_line1()`, en lisant simplement un par un les caractères sur la socket `s` avec `recv()`. Prenez bien en compte les cas d'erreurs (`man recv`). Pour ajouter le caractère `t` à la fin de la chaîne `line` de type `string`, utilisez `line.push_back(t)`. **On commencera par écrire sur papier un pseudo-code ...**

4) Complétez le serveur (dans la fonction `main()`) pour qu'il effectue le traitement attendu du client. Pour la lecture d'une ligne de texte sur la socket, utilisez la fonction `recv_line1()` que vous venez d'écrire. Pour l'envoi d'une chaîne de caractères au client, utilisez la fonction suivante, qui vous a été fournie gracieusement avec la `socklib`.

   ```
   int send_str(int s, const std::string &str);
   ```

   Vous consulterez les commentaires concernant cette fonction dans `socklib.h` de façon à bien gérer les cas d'erreurs. Attention à bien prendre son temps et à compiler et tester au fur et à mesure.

5) Testez votre serveur avec `nc` ou `telnet`. Logiquement, à chaque fois que vous tapez la touche `Entrée` côté client, la ligne que vous venez de taper doit être affichée par votre serveur... mais vous avez des problèmes de retour à la ligne à l'affichage : si c'est le cas, passez à la suite. Sinon, déboguez!

6) Pour bien faire les choses, il faut supprimer le retour à la ligne qui se trouve à la fin de la chaîne reçue par le serveur (notez qu'il se peut aussi la chaîne soit vide, ou que la connexion ait été coupée avant que le serveur ait reçu `'\n'`). En plus, certains clients (comme `telnet`) n'envoient pas simplement `'\n'` en fin de ligne, mais les deux octets CRLF (`"\r\n"`). Supprimez donc tous ces caractères éventuels en fin de chaîne; cela se fait bien en utilisant les méthodes `empty()`, `back()` et `pop_back()` de la classe `string`.

EXERCICE 3 ▶ **Facultatif**
Jusqu'à présent, votre serveur ne traite qu'un seul client, puis se termine : modifiez le de façon à ce qu'il puisse accepter plusieurs clients simultanément. Pour cela, une solution est que le serveur crée un processus fils à chaque fois qu'un nouveau client demande à se connecter. Chaque fils se charge du dialogue avec un client. Pendant ce temps, le serveur revient se mettre en attente sur la socket d'écoute. Notez que vous avez déjà étudié ce mode de fonctionnement lors du dernier TD.

Vous serez confrontés aux petits tracas habituels lorsqu'un processus crée des fils multiples : à chaque fois qu'un fils se termine, il reste dans l'état zombie tant que son père n'a pas pris en compte sa mort avec `waitpid()`. Pour éviter une accumulation de zombies digne d'un film d'horreur (et surtout pénalisante pour le système), vous devrez donc mettre en place un gestionnaire pour le signal `SIGCHLD`. Il faut aussi prévoir la terminaison propre du serveur, par exemple à la réception du signal `SIGTERM` ou `SIGQUIT`.

7.3 Le contrôle TP de l'année dernière

Description du programme. Votre programme doit implémenter un serveur réseau TCP qui utilise un protocole déterminé. Le programme prend un seul argument, qui est le numéro de port TCP sur lequel il doit se mettre en écoute.

Le serveur doit accepter les clients qui demandent à s'y connecter, et doit être capable d'en prendre en charge plusieurs à la fois. À chaque fois que le serveur accepte une nouvelle connexion :

- il affiche sur sa sortie standard le message
  `## Nouveau client connecté <adresse> <port>`
- il envoie au client la réponse (sans rien de plus)
  `Bienvenue\r\n`
- il crée un fils qui sera chargé du dialogue avec le client, selon le protocole qui sera décrit plus loin,
- il revient se mettre en attente de nouvelles demandes de connexion.

C'est le client qui décide quand la connexion se termine, en fermant lui-même la socket de dialogue. Lorsqu'un client ferme la connexion, le serveur doit afficher sur sa sortie standard :
`## Client déconnecté`

Le protocole. Il peut être décrit de la façon suivante :

- Le client envoie une commande et des données.
- La commande est une ligne de texte terminée par un `\r\n`.
- Les données sont
  - soit des chaînes de caractères terminées par un `\r\n`,
  - soit des données binaires dont la taille `t` en octets est fournie par la commande concernée.
- Les commandes comportent deux parties :
  - la commande de base, soit `REPEA`, soit `PRINT`;
  - La description de la donnée : `STR` pour une chaîne de caractères, `BIN t` pour des données binaires de taille `t`.

Les commandes prises en charge par le serveur sont :

- `REPEA` signifie que le serveur doit répéter la donnée sur la socket,
- `PRINT` signifie que le serveur doit afficher la donnée précédée de `## ` sur sa sortie standard. Dans le cas d'un `PRINT`, après avoir reçu les données, le serveur envoie au client un acquittement sous la forme du texte `DONE\r\n`.

Consignes et script de test. L'archive fournie contient tout ce dont vous avez besoin, dans le répertoire `tpnote2019`.

- une version de `socklib` qui va bien pour ce TP;
- le code source `server.cpp` à compléter;
- un `Makefile`, pour fabriquer l'exécutable `server`;
- le script de test `test.py`, que vous utiliserez pour tester votre serveur avec `./test.py server`

Le script `test.py` sera utilisé également pour vous noter : **vous devez suivre scrupuleusement les consignes sur l'affichage et l'envoi de message qui vous sont données pour pouvoir obtenir des points**. Si on vous demande d'afficher `## toto` sur la sortie d'erreur standard, mais que vous l'affichez sur la sortie standard, vous n'aurez pas les points correspondants...

`test.py` effectue quatre séries de tests (vérification des arguments, mise à l'écoute du serveur, vérification des connexions, vérification du protocole). Les explications sur chaque test sont précédées de `***` (et s'affichent normalement en jaune). Quand vous lancez `./test.py server`, les tests réussis s'affichent précédés de `OK` (sur fond vert), et les problèmes sont précédés de `Not OK` (sur fond rouge). Pour comprendre d'où vient le problème, remontez à l'explication précédant le `Not OK`.

À vous de jouer! Notez que la question `i` ($1 \leq i \leq 11$) correspond au `TODOi` repéré par un commentaire dans le code source.

1. Votre programme ne doit prendre comme argument que le port sur lequel le serveur doit se mettre à l'écoute. Il doit vérifier le nombre d'arguments qui lui sont passés, et si ce nombre n'est pas bon, afficher
   `## usage <commande> PORT`
   sur la sortie d'erreur standard (`<commande>` doit être le nom effectif de la commande). Complétez le code au niveau du `TODO1` de façon à respecter ces consignes.

   Vous devez, comme pour toutes les questions suivantes, tester votre programme : utilisez le script fourni avec la commande `./test.py server`.
2. Au niveau du `TODO2`, complétez le code en définissant une variable `port` de type `const char *` initialisée pour que, dans la suite du programme, elle pointe vers la chaîne de caractères qui contient le port passé en argument du programme. Le programme doit ensuite afficher sur la sortie standard (`<port>` est remplacé par sa valeur) :
   `## le port utilisé est <port>`

   À ce stade, la première série de tests doit être parfaite.
3. `TODO3` : en utilisant la fonction `create_server_socket()` de la `socklib`, créer une socket d'écoute `s` sur le port passé en argument du programme. En cas d'erreur (si le port demandé ne peut pas être ouvert), votre programme doit afficher sur la sortie d'erreur standard :
   `## erreur à la création du serveur sur le port <port>`
   (en remplaçant `<port>` par sa valeur) et se terminer ne retournant `EXIT_FAILURE`.
4. Dans la boucle d'attente des clients, complétez le code au niveau du `TODO4`, de façon à ce que le serveur se mette en attente d'une demande de connexion sur la socket d'écoute `s`. Quand un client vient se connecter avec succès, on appelle `sd` la socket de dialogue obtenue. En cas d'échec, le serveur affiche
   `## erreur lors de la connexion d'un client`
   puis se termine en retournant `EXIT_FAILURE`.

   À ce stade, la deuxième série de tests doit être parfaite.
5. `TODO5` : le serveur envoie le message `Bienvenue` au client. Veillez à ce que le message envoyé se termine bien par `\r\n`.
6. `TODO6` : modifiez le code de façon à ce que le processus principal crée un fils. Le code exécuté spécifiquement par le fils est compris entre l'accolade ouvrante et l'accolade fermante marquées par le commentaire `processus fils`. Le père quant à lui doit revenir se mettre en attente du client suivant.
7. Jusqu'à présent, on a ignoré la terminaison des fils quand un client ferme la connexion. Complétez les deux parties du code où se trouve le commentaire `TODO7`, de façon à ce que le père prenne en compte la mort de ses fils. Spécifiez l'option `SA_RESTART` dans la structure `struct sigaction` que vous utiliserez pour gérer correctement le signal `SIGCHLD`.

   À ce stade, la troisième série de tests doit être parfaite.
8. Dans la boucle de traitement des commandes (voir les commentaires dans le code), le processus fils créé s'occupe de traiter chacune des commandes envoyées par le client. Lorsque le client ferme la connexion, on sort de la boucle : le fils ferme alors la socket de dialogue, et se termine. Le code pour la réception d'une commande `com` vous est fourni à titre d'exemple (voir le `README1`). Ensuite, en fonction de la commande envoyée, le fils doit réagir différemment. Commencez par traiter le cas d'une commande `REPEA STR`, au niveau du `TODO8`. Dans ce cas, le fils doit :
   - recevoir une ligne de texte avec la fonction `recv_line()`,
   - traiter la valeur de retour (utilisez la variable `res`) de `recv_line()` : le client peut décider de fermer la connexion, ou bien il peut y avoir une erreur; dans les deux cas, il faut sortir de la boucle de traitement des commandes.
   - enlever les `\r` ou `\n` intempestifs par lesquels peut se finir cette ligne avec la fonction `strip_line()`,
   - renvoyer la ligne au client en la terminant par `\r\n` avec la fonction `send_str()`,

   puis il vient se remettre en attente de la commande suivante.
9. `TODO9` : traitez le cas d'une commande `PRINT STR`. Le fils doit :
   - recevoir une ligne de texte avec `recv_line()`,
   - traiter la valeur de retour (encore `res`) de `recv_line()`,

— envoyer au client l'acquittement constitué du texte `DONE\r\n`,
— enlever les `\r` ou `\n` intempestifs par lesquels peut se finir la ligne reçue,
— l'afficher sur sa sortie standard en la faisant précéder de `## `, et en ajoutant un retour à la fin (utilisez `endl` ou `'\n'`).

Le fils vient ensuite se mettre en attente de la commande suivante.

10. `TODO10` : traitez le cas d'une commande `REPEA BIN t`. Notez que le nombre d'octets à recevoir vous est fourni dans le source : utilisez juste la variable `t`. Le fils doit :
    — recevoir `t` octets depuis la socket de dialogue avec la fonction `recv_all()`,
    — traiter la valeur de retour (toujours `res`) de `recv_all()`, comme dans le cas de `recv_line()`,
    — envoyer ces `t` octets sur la socket de dialogue avec la fonction `send_all()`,

    puis se remettre en attente de la commande suivante.
11. `TODO11` : traitez le cas d'une commande `PRINT BIN t`. Le nombre d'octets `t` à recevoir vous est à nouveau fourni dans le code. Le fils doit :
    — recevoir `t` octets depuis la socket de dialogue,
    — traiter la valeur de retour (définitivement `res`) de `recv_all()`,
    — envoyer au client l'acquittement `DONE\r\n`,
    — envoyer les `t` octets reçus sur la sortie standard,

    puis se remettre en attente de la commande suivante.

    À ce stade, la quatrième série de tests doit être parfaite. Vous devez maintenant avoir le maximum de points.

TP 8
Un serveur HTTP

8.1 Introduction

8.1.1 Objectif du TP

Dans ce TP, nous allons programmer un *serveur web* simplifié. Pour être précis, un tel serveur est en fait un *serveur HTTP* : les clients sont les *navigateurs web* qui viennent se connecter, et les échanges qui ont lieu suivent le *protocole HTTP*. Nous nous restreignons dans ce TP à une très petite partie du protocole HTTP version 1.1 :

— le navigateur (le client, donc) se connecte en TCP/IP au serveur, puis adresse [1] une *requête* au serveur pour demander l'envoi d'un fichier;

— si le serveur possède le document demandé, il envoie une *réponse* au client, qui contient notamment la taille du document en octets (de façon à ce que le client connaisse le nombre d'octets à recevoir), puis il envoie le document;

— si le serveur ne possède pas le document demandé, il en informe le serveur par une *réponse* appropriée;

— dans tous les cas, le serveur ferme ensuite la connection.

En outre, si le client envoie une requête qui n'a pas été implémentée dans le serveur, celui-ci l'indique aussi au client.

8.1.2 Un peu de "protocole"

Voici un exemple de *requête GET* envoyée par Firefox [2] :

```
GET /index.html HTTP/1.1<CRLF>
Host: localhost:8080<CRLF>
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64) Gecko/20100101 Firefox/66.0<CRLF>
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8<CRLF>
Accept-Language: fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3<CRLF>
Accept-Encoding: gzip, deflate<CRLF>
Connection: keep-alive<CRLF>
Upgrade-Insecure-Requests: 1<CRLF>
<CRLF>
```

Dans le cadre de ce TP, on ne se préoccupera que de la première ligne, qui est de la forme :

```
GET path HTTP/1.1<CRLF>
```

Le `GET` indique le type de la requête : il s'agit ici d'une requête d'envoi d'un document. Le champ `path` donne le chemin vers le document demandé; ce chemin est relatif (sur le serveur) au répertoire contenant le site web hébergé. Le `HTTP/1.1` donne la version du protocole HTTP utilisée.

Cas document non trouvé Si le document `/index.html` n'est pas disponible, le serveur va répondre au client avec une réponse 404 (« *Not Found* »), pour faire savoir au client que le document demandé n'a pas été trouvé :

```
HTTP/1.1 404 Not Found<CRLF>
Content-Type: text/html<CRLF>
Content-length: 146<CRLF>
Connection: close<CRLF>
<CRLF>
<html>
  <head>
    <meta charset="UTF-8">
    <title>404</title>
  </head>
  <body>
  <h1>Erreur 404 : page non trouvée</h1>
  </body>
</html>
```

La ligne `HTTP/1.1 404 Not Found` indique que la page demandée au serveur n'a pas été trouvée. Noté que le serveur envoie tout de même une petite page HTML qui (généralement) sera affichée dans le navigateur, et qui informe l'utilisateur de la situation. Pour cela, il envoie deux lignes d'entête, avec le type (`text/html`) et la taille en octet (`146`) du document. La ligne `Connection: close` indique le serveur fermera la connexion juste après l'envoi de ce message.

1. oui, c'est le bon usage du verbe adresser ici
2. Dans le suite, on note `<CRLF>` pour désigner les deux octets "`\r\n`"

Cas document trouvé Si le fichier `/index.html` existe, le serveur envoie une réponse 200 (« *Ok* »), suivie du contenu du fichier. Voici un exemple :

```
HTTP/1.1 200 Ok<CRLF>
Content-Type: text/html<CRLF>
Content-length: 339<CRLF>
Connection: close<CRLF>
<CRLF>
<html>
  <head>
    <title>page exemple</title>
  </head>
  <body>
    <p>Ceci est un exemple de page web :D</p>
    <img src="romance2.jpg" align="center"/>
    <p>Quelques images du poste <a href="romance.html">Schneider Romance</a>.</p>
    <p>D'autres images, du poste <a href="mouette.html">Oceanic Mouette</a>.</p>
  </body>
</html>
```

Notez que le contenu du fichier `/index.html` est le texte (*une suite d'octets*) qui va de `<html>` à `</html>` (ces deux balises comprises), et que cela représente 339 octets, comme cela est indiqué dans le champ `Content-length` de la réponse. Le champ `Content-Type` donne le *type MIME* du fichier envoyé, et `Connection: close` précise que le serveur fermera la connexion quand il aura fini l'envoi du fichier.

Si on n'a pas GET Notez finalement que si une requête effectuée par le client n'est pas supportée par le serveur, il répond avec une réponse 501 (« *Not Implemented* »).

Remarque importante sur les pages mutiples Dans le fichier HTML précédent, la balise `<img src="romance2.jpg" align="center"/>` permet d'inclure une image. Le navigateur va devoir effectuer une autre requête pour obtenir le fichier `romance2.jpg`. Le serveur lui répondra avec une réponse 200 (en précisant le type `MIME image/jpeg`) et en envoyant octet par octet le fichier en question.

Pour charger une simple page HTML contenant quelques images, le navigateur (le client) va donc devoir effectuer de multiples connexions au serveur, en effectuant une requête GET pour chaque image (ou tout autre document) en plus de la requête GET initiale. Pour être fonctionnel, notre serveur web devra donc forcément permettre des connexions multiples.

8.2 Un serveur web minimal

EXERCICE 1 ▶ **Prise en main**

1) Téléchargez l'archive pour le TP, désarchivez-la, et vérifiez que le `Makefile` du premier répertoire `step1/` est opérationnel, et que vous obtenez bien l'exécutable `http_server`[3]. Notez que, dans le répertoire que vous avez obtenu en désarchivant, il y a un sous-répertoire `web`, qui contient des pages HTML et des images, que l'on utilisera par la suite pour tester le serveur.
2) Vous disposez déjà de quelque chose qui ressemble à un serveur web. En effet, le programme principal (dans le fichier source `step1/http_server.cpp`) contient le code pour mettre en place une socket d'écoute et accepter un client ; quand un client est accepté, on prend en compte sa requête avec :

```
if(reqmsg_type(req) == "GET") {
  string u, v;
  if(!reqmsg_split_reqget(req, u, v)) {
    close(sd);
    exit_error("lors de l'analyse d'une requête GET");
  }
  logmsg("url demandée : " + u);
  resp_test(sd);
}
else resp_code(sd, 501, "Not Implemented");
```

Compilez votre serveur, et testez-le avec votre navigateur en le mettant à l'écoute sur le port 8080 (`http_server 8080`). Ensuite, tentez de vous connecter au serveur avec votre navigateur web. Pour cela, l'url que vous entrez doit ressembler à `localhost:8080`. En consultant le fichier source (il est fortement conseillé de suivre les appels de fonction à partir du *main* et en expérimentant, répondez aux questions suivantes :

— Dans votre navigateur, entrez successivement les url suivantes, en relançant à chaque fois le serveur ; `localhost:8080`, `localhost:8080/index.html`, `localhost:8080/toto/tutu.titi`. Observez aussi les message qui s'affichent dans le terminal où vous lancez le serveur. Après l'appel (supposé réussi) à `reqmsg_split_reqget(req, u, v)`, que contient la variable `u` ?

3. Si vous êtes sous MacOsX, il est possible que vous ayez une erreur de compilation qui se résout en éditant le fichier de librairie `socklib`.

— Quelle est la page envoyée quand vous demandez une url quelconque, comme `localhost:8080/tutu.html` ?
— Pensez-vous que dans un vrai serveur web les pages soient intégrées au code du serveur, ou qu'il faille relancer le serveur à chaque fois qu'un navigateur obtient une page ?
— Que faire pour quitter le serveur ?
— Quel travail allez-vous devoir effectuer pour rendre ce serveur fonctionnel ?

EXERCICE 2 ▶ **Protocole avec un unique client**

1) Commencez le travail en complétant la fonction `bool send_vec(int s, const vector<char> &v)`, pour envoyer un `vector<char>` sur une socket. Cela se passe dans le code au niveau du commentaire `TODO1`. Utilisez pour cela la fonction `int send_all(int s, const char *p, int n)`, documentée dans `socklib.h`, et que l'on a déjà utilisée en TD/TP *Encore une fois, il s'agit de récupérer les bons paramètres à passer à la fonction.*

2) Comment tester cette fonction avant de passer à la suite ? On se demandera où appeler la fonction, et comment initialiser un vecteur de `char` en C++.

3) Au niveau du `TODO2`, complétez la fonction `bool resp_file(int s, const string &u)`, qui doit permettre d'envoyer une réponse `200` sur la socket `s`, en envoyant le contenu du fichier dont le chemin est donné par la chaîne `u`. Pour cela, utilisez :
— `send_str()` pour envoyer une chaîne sur une socket (voir `socklib.h`),
— `read_file()` pour charger le contenu d'un fichier dans un `vector<char>`,
— `send_vec()` pour envoyer le contenu d'un `vector<char>` sur une socket.

Suivez les commentaires fournis dans le fichier source, et dans `socklib.h` pour la fonction `send_str()`. Aidez-vous également du code des fonctions `resp_test()` et `resp_code()`.

4) Modifier le code au niveau du commentaire `TODO3` de façon à ce que le serveur réponde à une requête GET en envoyant le fichier demandé. Le chemin vers la fichier est formé du contenu de la variable `http_root` concaténé avec l'url relative de la page fournie par `reqmsg_split_reqget(req, u, v)`. Vous utiliserez `resp_file()` pour tenter d'envoyer le fichier, mais s'il n'est pas disponible, vous enverrez une réponse `404` avec `resp_code()`. Testez la réponse de votre serveur, en utilisant le fichier `index.html` du sous-répertoire `web` : que constatez-vous ?

EXERCICE 3 ▶ **Serveur à multiples clients, version 1**

Modifiez le programme de façon à ce que, à chaque nouvelle connexion d'un client, le processus principal lance un fils pour prendre en charge ce client. Ainsi, une fois le fils lancé, le processus principal peut revenir se mettre en attente d'une nouvelle connexion. Vous ne vous préoccuperez pas de la mort des fils pour l'instant (ni du petit cheval), on verra cela plus tard !

8.3 Gestion de la terminaison des clients, et du serveur

Dans cette section, nous allons finaliser le serveur web simplifié. Il s'agit avant tout de prendre en compte la terminaison des processus fils créés par le serveur, et de fournir un moyen de terminer proprement le serveur.

EXERCICE 4 ▶ **Startup !**
Dans le répertoire `step2/` :

1) Vérifiez que le `Makefile` est opérationnel, et que vous obtenez bien l'exécutable `http_server`. Dans le répertoire que vous avez obtenu en désarchivant, il y a, comme dans la partie précédente un sous-répertoire `web`, qui contient des pages HTML que l'on utilisera pour tester le serveur. Par la suite, vous allez travailler dans le fichier source `http_server.cpp`.

2) Lancez le serveur en le mettant à l'écoute sur le port 8080 (`./http_server 8080`), et connectez-vous à ce serveur depuis votre navigateur web en entrant l'url `localhost:8080` ; testez le serveur en parcourant les pages HTML (c'est important de parcourir les liens, de recharger les pages, etc). Ensuite, en utilisant la commande `ps -e` (tous les processus du système), observez l'état des processus fils lancés, que constatez-vous ?

....
....
....
....

3) Parcourez le code source fourni, comment est-il prévu que le serveur sorte de la boucle d'attente de nouveaux clients ?

EXERCICE 5 ▶ **Gestion de la terminaison des fils**

Pour ne pas avoir de multiplication des fils, nous allons comme d'habitude demander au père d'attendre sagement ses fils. Nous allons encore utiliser un gestionnaire du signal `SIGCHLD`.

1) Au niveau du commentaire `TODO4`, complétez la fonction `void handler(int sig)`, pour qu'elle puisse (par la suite) permettre au processus père de gérer les signaux `SIGCHLD` et `SIGINT`. Il faut que, à la réception du signal `SIGCHLD`, le père prenne en compte la mort de l'un de ses fils avec `waitpid()`, et qu'à la réception de `SIGINT` il sorte de la boucle d'attente de nouveaux clients.

2) Parcourez les blocs de code signalés par les commentaires `README1` et `README2` : une fois que les deux blocs de code ont été exécutés, que contiennent les structures `sigaction` suivantes? *il peut être utile de relire le manuel de la fonction `sigaction`.*

— `oldsa_chld`

— `newsa_chld`

....

....

— `partsa_int`

....

....

— `chldsa_int` *Regarder le manuel pour savoir ce que peut bien vouloir dire `SIG_IGN`.*

....

....

3) Au niveau du `TODO5`, en utilisant un appel à la primitive `sigaction()`, rétablissez dans les fils le gestionnaire par défaut pour le signal `SIGCHLD`. Pour ce qui est de l'appel à `sigaction()`, vous pouvez vous inspirer du code fourni au niveau du commentaire `README1`.

EXERCICE 6 ► **Gestion de la terminaison du père**

Ici nous allons mettre en oeuvre un gestionnaire du signal `SIGINT` uniquement destiné au père. Les fils vont ignorer ce signal, et le père terminera proprement avec un message en cas de `CTRL-C`.

1) Pour le `TODO6`, intallez le gestionnaire pour le signal `SIGINT` qui va permettre aux fils d'ignorer ce signal. Inspirez-vous du code fourni au niveau du commentaire `README2`.

2) A la sortie de la boucle `while`, c'est-à-dire au niveau du `TODO7`, complétez le code de façon à ce que, lorsque le serveur se termine, il ferme la socket d'écoute, puis attend patiemment que chacun de ses fils se termine.

3) Tester.