TP1: getting to know a UNIX system

Exercise 1: wondering around your system

a. Open a Terminal
b. Navigate to the following directory: `//bin` (remember the `cd` command and the absolute and relative paths studied in class)
c. Respond to the following questions (hint: everything can be done using the commands studied in class)
    a. Who is the owner of file `bash`?
    b. Who can override file `chmod`?
    c. List the files whose name is composed of exactly 4 characters
    d. How many files have names composed by exactly 3 characters in which character `a` is the second character?
    e. What kind of file (e.g., executable, text) is `znew` (in some systems, e.g., OSX, this files is found at /usr/bin/)?
    f. Is file `znew` viewable as text? If yes: what does the first line of the file say?
d. During your exploration of the `//bin` you probably notice that some files have some 'weird' names like `sh -> dash` or `rbash -> bash`. These are called *symbolic links*. Google[1] what a symbolic link is and write it down in your own words (no more than 5 lines).

Exercise 2: manipulating files and directories

a. Open a Terminal
b. Navigate to your `Documents` directory
c. Create a local directory called UNIX-TD1
d. Use the `vi` editor to create a file called `myAnimals.txt` in the current working directory. Use the `man` command to get some information about the `vi` program. You can also Google a tutorial (or try this one https://kb.iu.edu/d/adxz)
e. Type the following list of animals in your file, save your file, and exit `vi`

Fox	Bear	Eagle	Deer
Dog	Zelot	Chicken	Elk
Giant Panda	Crab	Ant	Dove
Bee	Bison	Cat	Alligator
Camel	Bat	Falcon	Coyote

f. Make a copy of your file (call it `myAnimals.bkp`) and stock it in a new directory with the following path `./bacup`

[1] If you do not find a French site try this : https://kb.iu.edu/d/abbe

g. Now you realize you misspelled the name of the backup directory (you wrote bacup). Rename it to `backup` (hint: we didn't discussed the corresponding command in class)

Exercise 3: redirecting I/O and piping

a) Open a Terminal
b) Navigate to your UNIX-TD1 directory
c) Read the manual page for command gedit (i.e., run the following command `man gedit`)
d) Now execute the following command: `gedit myAnimals2.txt`. What happens? Why?
e) Using the `gedit` program, add the following list of animals to your file. Save and quit (you should be taken back to the shell)

Fox	Bear	Eagle	Pelican
Caribou	Zelot	Koala	Elk
Giant Panda	Manatee	Ant	Dove
Mouse	Polar Bear	Gorilla	Alligator
Camel	Bat	Falcon	Horse

f) Now, using command `echo` (check the manual page `man echo`) and I/O redirection add "Raven" to the animal list in your `myAnimals2.txt` file
g) If we combine the two files, how many lines are there? How many words? Use a pipe of commands `cat` and `wc` to answer the questions.
h) Using pipes, write a file called `myCombinedAnimals.txt` containing the names of all the animals in your `myAnimals.txt` and `myAnimals2.txt` files. The file should be sorted (alphabetically) and it should not contain repeated values (e.g., Fox appears in both input files, it should appear only once in the output file)
i) Now using a single shell command write a file called `MYcOMBINEDaNIMALS.txt` where the names of the animals are written in capital case. **Hint**: check the `tr` program
j) Using a single pipe answer the following question: how many words in your `MYcOMBINEDaNIMALS.txt` file contain an `A` character? Hint: check the `grep` program
k) How many words **do not** contain an A character? Write a file called a`nimalsWithNoA.txt` with the list.

Exercise 4: some new commands and a bit more piping

What do the following statements do?

a) `ls -lt | head`
b) `du | sort -nr`

c) `find . -type f -print | wc -l`
d) `telnet towel.blinkenlights.nl`
e) `rev < file_1 > file_1`
f) `banner your_first_name`

Exercise 5: creating users and changing permissions

a) Navigate to your TD1 directory
b) Change the permissions of file `myAnimals.txt` so nobody can write the file
c) Execute the following command: echo `'lombriz' > myAnimals.txt`. What happens?
d) Now, use google to find a command that allows you to create a new user
e) Create a user called "`other_user`" with password "o"
f) Transfer the ownership of file `myAnimals` to user `other_user`
g) Now try to change the permissions of file `myAnimals` so everybody can read, write, and execute the file. What happens? Why?
h) Now delete user "`other_user`". What happened to the `myAnimals.txt` file?

UNIX - TP2: programmatically manipulating files

1 Warm up exercise: playing with program parameters

What does command `"echo *"` do? have you ever asked yourself why? Write a C program that prints to screen the arguments passed to the program.

2 Manipulating file attributes

Write a C program that produces exactly same output than the shell command `"ls -il"`. We will implement our program step by step.

2.1 Printing the file names

For starters, implement the logic that goes through the files passed as parameters and prints on the screen only the names of the files (this looks pretty much as the code we did in the first exercise)

2.2 Printing the inode number (option -i)

This is a lot more trickier. Check the manual entry for system call `lstat` and the closely related struct `stat`. Using these two powerful build-ins you can easily obtain the information of the inode and print it on the screen.

2.3 Printing the file type

Ok now things are getting more serious. As you may have learned while dealing with the inode numbers, the `stat` struct returned by the `lstat` call also holds information about the file's type. Unfortunately, extracting this information is far less trivial than extracting the inode number. You need to find out which is the element of the `stat` struct that holds the file type information and how to interpret it. You may find the following link useful `http://www.gnu.org/software/libc/manual/html_node/Testing-File-Type.html`

2.4 Printing the file permissions

There is a good and a bad news here. The good one is that our friend `stat` holds all the information we need to print file permissions (no need to learn new system calls or structs). The bad one is that interpreting the information is a little tricky, specially because there are a couple of permission types that we did not discuss in class. Check out the following tutorial to try to understand the concept of SUID, SGID, and Sticky bit: `https://www.ibm.com/developerworks/community/blogs/brian/entry/every_possible_unix_linux_file_permission_listed_and_explained_all_4_096_of_them?lang=en`. Now that you understood this "new" permissions, you're ready to implement this part of our program.

Hints:

- here is a list of `st_mode` bits were you can find the information about file permissions: `http://www.gnu.org/software/libc/manual/html_node/Permission-Bits.html#index-file-permission-bits`

- check out this tutorial to understand how the binary comparison between bit permissions and the contents of element `st_mode` is done: `http://stackoverflow.com/questions/15055634/understanding-and-decoding-the-file-mode-value-from-stat-function-output`. This may guide your on your implementation.

2.5 Adding the links

This is a pretty easy one; all the info is directly accessible at the `stat` struct. You do not need hints here ;)

2.6 Adding the owner's name

You probably already saw that struct `stat` contains an element called `st_uid` holding the ID of the owner of the file. Unfortunately, we are not looking for the owner's ID but for the owner's user name. Nonetheless, the ID is a very good starting point for our search! Check out the manual pages for system call `getpwuid` and the related `passwd` struct; those tools will help you accomplishing your mission.

2.7 Adding the group's name

Here you are facing a very similar problem. The `stat` struct contains an element holding the ID of the group, but we are looking for the name of the group. Check out system call `getgrgid` and the related `group` struct.

2.8 Adding the file size

This one is not difficult. You do not need hints.

2.9 Adding the modification date

Once again the `stat` struct has all the information you need. However, before outputting the modification date you need to format it. Check out the library functions `strftime` and `gmtime`; they can be handy for this job.

3 Accessing directories

To pass the name of the files as parameters to our current implementation we use meta-character `*`. Based on our current implementation now develop a C program that generates exactly the same output than `"ls -ila directory`, where `directory` is the name of the directory you want to explore.

Hints:

- Check what library function `opendir` can do for you

UNIX - TP3: the `fork()` system call

1 The "Hello `fork()`"

Write a C program that executes the following sequence of actions.

1. A process creates a child process using the `fork()` system call
2. The parent process should print to screen the following message:
   ``Hello I am the parent process, my process id is XX, my parent's id is YY, and I am the parent of process ZZ.''
3. The child process should print to screen the following message:
   ``Hello I am the child process, my process id is XX, and my parent's id is YY.''

Run your program a few times. Do you find something disturbing in the output? what is it? how can you explain it?

2 "Hello `fork()`" V.2.0

Write a new version of your "Hello `fork()`" program solving the problem.

3 Multitasking

In class we said that Unix is a time-sharing + multitasking operative system. Write a C program showing that a parent and a child process are executed concurrently.

4 Multiple child processes

Write a C program that uses multiple child processes to perform the following operation $\frac{(a+b)\times(c+d)}{(e+f)}$. Your program should be written so:

- $(a + b)$ is performed by child process 1
- $(c + d)$ is performed by child process 2
- $(e + f)$ is performed by child process 3
- the multiplication and the division are performed by the parent process
- the values for a, b, c, d, e, and f are read from the keyboard[1]

As we discussed in class, there are several mechanism that we can use for process communication. Can you enumerate some of them? Here we want to use what is known as *shared memory*. Check out this short, yet excellent, tutorial on how the mechanism works `http://www.ibm.com/developerworks/aix/library/au-spunix_sharedmemory/`. The example at the end of the tutorial is a good starting point to tackle the exercise.

[1] for simplicity assume that a, b, c, d, e, and f are always integers

UNIX - TP4: the `pipe()` system call

1 Redirecting outputs

Write a C program that:

- Creates a child process using the `fork()` system call
- Redirects the child process' standard output to a file. Assume the name of the file is `output.txt`
- Writes, using the `printf()` function, some characters (of your choice) to the file

2 Communication between process

Write a C program that:

- sets a pipe
- creates a child process using the `fork()` system call

Show that a pipe allows process to communicate (in this case send characters to each other) and synchronize execution of two processes (block a process if an instruction cannot be executed).

3 Pipe redirection

Now modify your code so the two processes communicate using function `printf()` and `fgets()`. Show the data structures (per process file table, file table, inode table) for every step of the program's execution.

4 Two redirections

Write a C program which opens two pipes (`tube1` and `tube2`) and creates a child process. The child process should redirect its standard output to `tube1` and its standard input to `tube2`. The parent process should redirect its standard input to `tube1` and its standard output to `tube2`. The child process should send the message "Message from the child" to its parent, and the parent process should send the message "Message from the parent" to its child. To check that everything works properly, each process should write the received message to the console.

5 Concurrent access to a pipe

Can a pipe be used by more than two processes? To answer the question write a C program that creates two processes. Each child process should send a message to the parent process using a single pipe.

6 Concurrent access to two pipes

Write a C program that creates two processes. Each process should send messages to the parent process using a dedicated pipe (i.e., each child process shares one pipe with the parent process). The messages sent by the children should be printed out to the console as soon as they are sent. To accomplish that, read access to the pipes should be non-blocking.

Hint: check how function `fcntl()` can help you configuring the file descriptors for I/O operations.

UNIX - TP5: the `kill()` and `signal()` system calls

1 Sending signals

Write a C program in which:

- a process creates a child process using the `fork()` system call
- the parent process sends a signal `SIGUSR1` to the child process using the `kill()` system call
- the child process does some computations (of your choice)
- the child process reacts to the `SIGUSR1` by outputting to the consolue its PID and the signal number (*hint*: check the `signal()` system call)

Remember that a process can react to a signal in three possible ways: ignore it, stop the execution, execute a predefined function. Use your program to illustrate how to implement these behaviors.
It is worth recalling also that a process can only receive signals if it is "alive" and that if it chooses to handle signals by executing a function it needs "some time" to install the signal handlers before the first signal can be handler.

2 Establishing non-blocking communication

Write a C program in which:

- a process creates a child process
- the parent process sends a message to the child process and then notifies the child process that a message has been sent
- the child process does some computations (of your choice)
- the child process outputs to the console the message sent by the parent process without stopping what it is doing.

Hint: you can use pipes and signals to implement the messaging + notification mechanism.

3 Better together: sharing the workload between processes

Write a C program for adding two values. The program should split the work between two process as follows: P1 reads the values (real numbers) from the keyboard and prints the result while P2 performs the sum.

Process P1 transfers the data to P2 in the format "$< value1 > < value2 >$" using a pipe. Process P2 performs the sum as soon as it gets a `SIGUSR1` signal from P1.

Process P2 sends the result ($value1 + value2$) to P1 in the format "$< PID > < result > \backslash n$", where PID is the id of P2, using a pipe. Process P1 then outputs the result to the console while P2 remains alive waiting for new data to compute.

The program should stop when P1 receives an interruption signal `SIGINT` (which is sent by the terminal when the user enters Ctrl + c). Process P2 should not be interrupted by that signal. On the other hand, P2 should stop its execution when it gets a `SIGUSR2` from P1. Both processes should output a message to the console announcing their interruptions.

UNIX - TP6: the `exec()` system call

1 Meet the `execs`

As we mentioned it in class the `exec` family of functions replaces the current process image with a new process image.

1.1 The hello world

Write a C program that uses one of the functions of the `exec` family to execute command `ls -l`. Which function did you choose? why?

1.2 `execs` and `pipes`

Now we want to re-write our program so the output is printed by the parent process (if you coded exercise 1.1 right, the output is currently printed by the child process). The child process should send its output to its parent using a pipe.

Hint: check library function `fgets()`, it may help you retrieving the text written in the pipe.

1.3 A more tricky one

Based on what you learned in exercises 1.1 and 1.2 write a C program that executes command `ls -l | tail -3 | wc -w`. Each of the three programs (i.e., `ls`, `tail`, and `wc`) should be executed by a child process and the output should be printed to the screen by the parent process.

2 The watchdog

Write a C program that creates a child process executing a long-lasting task. Now, implement in the parent process a watchdog system that kills the child process if its execution takes longer than, say, 5 seconds.

Hint: check out the `kill()` and `alarm()` functions.

3 `execs` and signals

3.1 Signal handlers

Can a signal handler set before a call to an `exec` function be used by the `exec`ed program? Why? Write a C program illustrating your answer.

3.2 Ignoring signals

From the answer to question 3.1 you should know that you can configure processes running `exec`ed programs to ignore signals. Is this true for any type of signal? Write a C program illustrating your answer.

TP noté: programmation d'un shell

1 Introduction

Dans ce TP nous allons implanter un "mini-shell". Vous avez 4 séances pour travailler sur votre implantation. Lors de la dernière séance vous devrez répondre à des questions posées binôme par binôme. Vous devez déposer votre code sur CELENE la veille de votre dernière séance de TP à 23h59 [1]. Pour rappel, la note de ce TP correspond à 75% de la note finale du cours.

2 Fonctionnalités du shell

Fonctionnalités souhaitées :

- Exécution de commandes avec des pipes
- Exécution de commandes avec des redirection d'entrée / sortie
- Un chien de garde (à chaque exécution d'une commande, le shell met fin aux processus générés au bout de 5 secondes si celle-ci est sans réponse).

Fonctionnalités bonus :

- Auto completion
- Historique des commandes

Typiquement votre shell doit être capable d'exécuter correctement une commande de type :

```
cat < /var/log/messages | grep ACPI | wc -l >> truc.txt
```

3 Décomposition d'une commande

Une commande est séparée en un certain nombre de membres, délimités par des pipes (et le début et la fin de la commande).
Chaque membre comporte une ou plusieurs redirections d'entrées sorties :

- `cmd < f` est équivalent à un `cat f | cmd`. Note : un membre comportant un `<` ne peut pas être précédé d'un autre membre.
- `cmd > f` redirige la sortie standard de `cmd` vers un fichier `f`, qui est écrasé. Note : un membre comportant un `>` ne peut pas être suivi d'un autre membre.
- `cmd >> f` est identique au symbole précédent, mais l'écriture dans le fichier sur fait à la fin de celui-ci. Il n'est pas écrasé.

1. G1 et G2 : le 18/01, G3 : le 16/01, Mundus : le 15/01

— `cmd 2>f` et `cmd 2>>f` sont identiques aux deux précédents, à cela près qu'ils portent sur la sortie d'erreur.

L'exécution d'une commande se décompose ainsi :

1. Séparer les différents membres
2. Regarder si les membres comportent des redirections d'entrée sortie
3. Effectuer le bon nombre de *fork*
4. Rediriger correctement STDIN, STDOUT et STDERR
5. Effectuer les *exec*
6. Attendre la fin des fils

Il est nécessaire et important de définir une structure adaptée pour stocker une commande. La structure décrite ci-dessous devra donc être utilisée.

Listing 1 – command struct

```
typedef struct {
    //the command originally inputed by the user
    char *initCmd;

    //number of members
    unsigned int nbCmdMembers;

    //each position holds a command member
    char **cmdMembers;

    //cmd_members_args[i][j] holds the jth argument of the ith member
    char ***cmdMembersArgs;

    //number of arguments per member
    unsigned int *nbMembersArgs;

    //the path to the redirection file
    char ***redirection;

    //the redirection type (append vs. override)
    int **redirectionType;
} cmd;
```

en gardant l'exemple donné précédemment, cette structure doit être initialisée ainsi :

Listing 2 – exemple

```
init_cmd="cat␣<␣/var/log/messages␣|␣grep␣ACPI␣|␣wc␣-l␣>>␣truc.txt"
nb_cmd_members=3
cmd_members[0]="cat␣<␣/var/log/messages"
cmd_members[1]="grep␣ACPI"
cmd_members[2]="wc␣-l␣>>␣truc.txt""
cmd_members_args[0][0]="cat"
cmd_members_args[0][1]=NULL
cmd_members_args[1][0]="grep"
cmd_members_args[1][1]="ACPI"
cmd_members_args[1][2]=NULL
```

```
cmd_members_args[2][0]="wc"
cmd_members_args[2][1]="-l"
cmd_members_args[2][2]=NULL
nb_members_args[0]=1
nb_members_args[1]=2
nb_members_args[2]=2
redirection[0][STDIN]="/var/log/messages"
redirection[0][STDOUT]=NULL
redirection[0][STDERR]=NULL
redirection[1][STDIN]=NULL
redirection[1][STDOUT]=NULL
redirection[1][STDERR]=NULL
redirection[2][STDIN]=NULL
redirection[2][STDOUT]="truc.txt"
redirection[2][STDERR]=NULL
redirection_type[2][STDOUT]=APPEND
```

Vous mettrez la définition de cette structure ainsi que les prototypes des fonctions utiles à sa manipulation dans un fichier `cmd.h`. Ces fonctions peuvent être :

Listing 3 – Exemple de fonctions en `cmd.h`

```
//Prints the command
void printCmd(cmd *cmd);
//Frees memory associated to a cmd
void freeCmd(cmd  * cmd);
//Initializes the initial_cmd, membres_cmd et nb_membres fields
void parseMembers(char *s,cmd *c);
```

Une première version de `cmd.h` est disponible sur CELENE.

Créez une fonction `void exec_command(cmd c)`, dans un fichier `shell_fct.c`, qui prend une commande duement initialisée et qui effectue la création des `pipes`, les `fork` et les `execs` correspondants. Étant donné le formattage des arguments, il est judicieux d'utiliser l'appel `execvp`.

Un *makefile* vous ait également proposé pour compiler votre projet.

Pour la gestion des entrées de l'utilisateur, je vous conseille de regarder du côté de la librairie GNU/Readline, bien que cela ne soit pas une obligation.