TP1: getting to know a Unix-like 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`? SOL: `ls – al` (should be root)
    b. Who can override file `chmod`? SOL: `ls – all` (only the owner = ROOT)
    c. List the files whose name is composed of exactly 4 characters SOL: `ls ????` (total of 10 in Mac)
    d. How many files have names composed by exactly 3 characters in which character `a` is the second character? SOL: `ls ?a?` (cat and pax in Mac)
    e. What kind of file (e.g., executable, text) is `znew` (in some systems, e.g., OSX, this files is found at `/usr/bin/`)? SOL: `file znew` (it is a bourne shell script)
    f. Is file `znew` viewable as text? If yes: what does the first line of the file say? SOL: yes (`cat znew` or better `head znew`)
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. Navigate to your `Documents` directory
b. Create a local directory called UNIX-TD1 SOL: `mkdir UNIX-TD1`
c. Use the `vi` command 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) SOL: para salir Esc + :wq
d. 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

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

e. Make a copy of your file (call it `myAnimals.bkp`) and stock it in a new directory with the following path `./bacup` SOL: `mkdir bacpup` for creating the directory + `cp myAnimals.txt ./bacup/myAnimals.bkp`
f. 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) SOL: `mv bacup backup`

Exercise 3: redirecting I/O and basic piping

a) Navigate to your UNIX-TD1 directory
b) Read the manual page for command gedit (i.e., run the following command `man gedit`)
c) Now execute the following command: `gedit myAnimals2.txt`. What happens? Why?
d) 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

e) 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`
Solution: `echo Raven >> myAnimals2.txt`
f) If we combine the two files, how many lines are there? How many words? Use a pipeline of commands `cat` and `wc` to answer the questions.
Solution: `cat myAnimals.txt myAnimals2.txt | wc`
g) 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)
Solution: `cat myAnimals.txt myAnimals2.txt | sort | uniq > myCombinedAnimals.txt`
h) Now using a single pipeline write a file called `MYcOMBINEDaNIMALS.txt` where the names of the animals are written in capital case
Solution: `tr [:lower:] [:upper:] < myCombinedAnimals.txt > MYcOMBINEDaNIMALS.txt`
Solution: `cat myCombinedAnimals.txt | tr a-z A-Z > MYcOMBINEDaNIMALS.txt`

i) Using a single pipe answer the following question: how many words in your `MYcOMBINEDaNIMALS` file contain an `A` character? Hint: use the `grep` program
Solution: `grep "A" MYcOMBINEDaNIMALS | cw`

j) How many words do not contain an A character? Write a file called animalsWithNoA with the list.
Solution: `grep -v "A" MYcOMBINEDaNIMALS > animalsWithNoA`

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`

Solution

a) Displays the 10 newest files in the current directory (t sorts by modification time newest first)
b) Displays a list of directories and how much space they consume, sorted from the largest to the smallest (du computes the space, n sorts numerically, and r reverses the order)
c) Displays the total number of files in the current working directory and all of its subdirectories (find lists the files, -type f filers only to include files, print prints the list in the standard output)
d) Connects to host towel.blinkenlights.nl through telnet (ask them to check what telnet is) and watch an ASCI version of Star Wars
e) Reverses the strings in every line in input_file_name and writes the output to output_file_name
f) Prints a banner with your name

Exercise 5: creating users and changing permissions

a) Navigate to your TD1 directory
b) Change the permissions of file `myAnimals` so nobody can write the file
Solution: `chmod 555 myAnimals`
c) Execute the following command: echo `"lombriz" > myAnimals`. 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"
   Solution: `sudo useradd other_user`
   Solution: `sudo passwd`
f) Transfer the ownership of file `myAnimals` to user `other_user`
   Solution: `sudo chown other_user myAnimals`
g) Now try to change the permissions of file `myAnimals` so everybody can read, write, and execute the file. What happens? Why?
   Solution: `chmod 777 myAnimals`
h) Now delete user "`other_user`". What happended to the `myAnimals` file?
   Solution: `userdel other_user`

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.

Listing 1: Solution

```
#include <stdio.h>

int main(int argc, char **argv)
{
        int i;

        //argc[0] holds the name of the program

        //print to screen all the arguments
        for(i = 1; i < argc; i++)
                printf("%s\t", argv[i]);
        printf("\n");

        return 0;
}
```

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)

Listing 2: Solution

```
#include <stdio.h>


int main(int argc, char **argv)
{
    int i;


    // FOR EACH FILE IN THE PARAMETERS
    for(i = 1; i < argc; i++)
    {
        //print the name of the file and break the line
        printf("%s\n", argv[i]);
    }

    return 0;
}
```

NOTES FOR INSTRUCTORS:
To execute the program from the command file: ./a.out *

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.

Listing 3: Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/stat.h>

int main(int argc, char **argv)
{
    int i;
    struct stat fileStat;

    // FOR EACH FILE IN THE PARAMETERS
    for(i = 1; i < argc; i++)
    {
        // GET INFO
        if(lstat(argv[i], &fileStat)){
            //return value = -1 if error
            perror("stat");
            exit(errno);
        }

        // I-NODE NUMBER
        printf("%d␣", (int)fileStat.st_ino);


        // FILE NAME
        printf("%s\n", argv[i]);
    }

    return 0;
}
```

NOTES FOR INSTRUCTORS:
Study with them the the ”stat” structure man lstat.
Study with them the RETURN VALUES part of man lstat. Explain them what errno is.
Don't forget to add to the code #include ¡sys/stat.h¿ (the stat structure is defined there).
Don't forger to add #include ¡errno.h¿ (the errno variable is defined there).
In the code's first if: remember than in C there are no booleans. 0 Means false, any other value means true.
Tell them about the perror() function. Use the man perror to explain them what the function does.

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`

Listing 4: Solution

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/stat.h>

int main(int argc, char **argv)
{
    int i;
    struct stat fileStat;


    // FOR EACH FILE IN THE PARAMETERS
    for(i = 1; i < argc; i++)
    {
        // GET INFO
        if(lstat(argv[i], &fileStat)){
            //return value = -1 if error
            perror("stat");
            exit(errno);
        }

        // I-NODE NUMBER
        printf("%d␣", (int)fileStat.st_ino);

        // FILE TYPE
        switch(fileStat.st_mode & S_IFMT)
        {
            case S_IFSOCK : printf("s"); break;
            case S_IFLNK : printf("l"); break;
            case S_IFREG : printf("-"); break;
            case S_IFBLK : printf("b"); break;
            case S_IFDIR : printf("d"); break;
            case S_IFCHR : printf("c"); break;
            case S_IFIFO : printf("p"); break;
            default : printf("?");
        }

        // nom du fichier et saut de ligne
        printf("%s\n", argv[i]);
    }

    return 0;
}
```

NOTES FOR INSTRUCTORS:
Explain them the logic AND operator (read with them the wikipedia page).
Explain them how to extract the file time by bytewise and-ing the code in the stat struct and S_IFMT.

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.

Listing 5: Solution

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/stat.h>



int main(int argc, char **argv)
{
    int i;
    struct stat fileStat;
    struct passwd *pwd;
    struct group *grp;
    char date[20];

    // FOR EACH FILE IN THE PARAMETERS
    for(i = 1; i < argc; i++)
    {
        // GET INFO
        if(lstat(argv[i], &fileStat)){
            //return value = -1 if error
            perror("stat");
            exit(errno);
        }

        // I-NODE NUMBER
        printf("%d␣", (int)fileStat.st_ino);

        // FILE TYPE
        switch(fileStat.st_mode & S_IFMT)
        {
            case S_IFSOCK : printf("s"); break;
            case S_IFLNK : printf("l"); break;
            case S_IFREG : printf("-"); break;
            case S_IFBLK : printf("b"); break;
            case S_IFDIR : printf("d"); break;
            case S_IFCHR : printf("c"); break;
            case S_IFIFO : printf("p"); break;
            default : printf("?");
        }

        // PERMISSIONS
        // permissions for the owner
        if((fileStat.st_mode & S_IRUSR) == S_IRUSR)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWUSR) == S_IWUSR)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXUSR) == S_IXUSR){
            if((fileStat.st_mode & S_ISUID) == S_ISUID)
                printf("s");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISUID) == S_ISUID)
                printf("S");
            else
                printf("-");
        }
        // permissions for the group
        if((fileStat.st_mode & S_IRGRP) == S_IRGRP)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWGRP) == S_IWGRP)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXGRP) == S_IXGRP){
            if((fileStat.st_mode & S_ISGID) == S_ISGID)
                printf("s");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISGID) == S_ISGID)
                printf("S");
            else
                printf("-");
        }

        // permissions for other users
        if((fileStat.st_mode & S_IROTH) == S_IROTH)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWOTH) == S_IWOTH)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXOTH) == S_IXOTH){
            if((fileStat.st_mode & S_ISVTX) == S_ISVTX)
                printf("t");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISVTX) == S_ISVTX)
                printf("T");
            else
                printf("-");
        }

        // FILE NAME
        printf("␣%s\n", argv[i]);
    }

    return 0;
}
```

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 ;)

Listing 6: Solution

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/stat.h>



int main(int argc, char **argv)
{
    int i;
    struct stat fileStat;
    struct passwd *pwd;
    struct group *grp;
    char date[20];

    // FOR EACH FILE IN THE PARAMETERS
    for(i = 1; i < argc; i++)
    {
        // GET INFO
        if(lstat(argv[i], &fileStat)){
            //return value = -1 if error
            perror("stat");
            exit(errno);
        }

        // I-NODE NUMBER
        printf("%d␣", (int)fileStat.st_ino);

        // FILE TYPE
        switch(fileStat.st_mode & S_IFMT)
        {
            case S_IFSOCK : printf("s"); break;
            case S_IFLNK : printf("l"); break;
            case S_IFREG : printf("-"); break;
            case S_IFBLK : printf("b"); break;
            case S_IFDIR : printf("d"); break;
            case S_IFCHR : printf("c"); break;
            case S_IFIFO : printf("p"); break;
            default : printf("?");
        }

        // PERMISSIONS
        // permissions for the owner
        if((fileStat.st_mode & S_IRUSR) == S_IRUSR)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWUSR) == S_IWUSR)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXUSR) == S_IXUSR){
            if((fileStat.st_mode & S_ISUID) == S_ISUID)
                printf("s");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISUID) == S_ISUID)
                printf("S");
            else
                printf("-");
        }
        // permissions for the group
        if((fileStat.st_mode & S_IRGRP) == S_IRGRP)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWGRP) == S_IWGRP)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXGRP) == S_IXGRP){
            if((fileStat.st_mode & S_ISGID) == S_ISGID)
                printf("s");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISGID) == S_ISGID)
                printf("S");
            else
                printf("-");
        }

        // permissions for other users
        if((fileStat.st_mode & S_IROTH) == S_IROTH)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWOTH) == S_IWOTH)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXOTH) == S_IXOTH){
            if((fileStat.st_mode & S_ISVTX) == S_ISVTX)
                printf("t");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISVTX) == S_ISVTX)
                printf("T");
            else
                printf("-");
        }

        // LINKS
        printf("␣%d␣", (int)fileStat.st_nlink);

        // nom du fichier et saut de ligne
        printf("␣%s\n", argv[i]);
    }

    return 0;
}
```

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.

Listing 7: Solution

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <pwd.h>


int main(int argc, char **argv)
{
    int i;
    struct stat fileStat;
    struct passwd *pwd;
    char date[20];

    // FOR EACH FILE IN THE PARAMETERS
    for(i = 1; i < argc; i++)
    {
        // GET INFO
        if(lstat(argv[i], &fileStat)){
            //return value = -1 if error
            perror("stat");
            exit(errno);
        }

        // I-NODE NUMBER
        printf("%d␣", (int)fileStat.st_ino);

        // FILE TYPE
        switch(fileStat.st_mode & S_IFMT)
        {
            case S_IFSOCK : printf("s"); break;
            case S_IFLNK : printf("l"); break;
            case S_IFREG : printf("-"); break;
            case S_IFBLK : printf("b"); break;
            case S_IFDIR : printf("d"); break;
            case S_IFCHR : printf("c"); break;
            case S_IFIFO : printf("p"); break;
            default : printf("?");
        }

        // PERMISSIONS
        // permissions for the owner
        if((fileStat.st_mode & S_IRUSR) == S_IRUSR)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWUSR) == S_IWUSR)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXUSR) == S_IXUSR){
            if((fileStat.st_mode & S_ISUID) == S_ISUID)
                printf("s");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISUID) == S_ISUID)
                printf("S");
            else
                printf("-");
        }
        // permissions for the group
        if((fileStat.st_mode & S_IRGRP) == S_IRGRP)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWGRP) == S_IWGRP)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXGRP) == S_IXGRP){
            if((fileStat.st_mode & S_ISGID) == S_ISGID)
                printf("s");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISGID) == S_ISGID)
                printf("S");
            else
                printf("-");
        }

        // permissions for other users
        if((fileStat.st_mode & S_IROTH) == S_IROTH)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWOTH) == S_IWOTH)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXOTH) == S_IXOTH){
            if((fileStat.st_mode & S_ISVTX) == S_ISVTX)
                printf("t");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISVTX) == S_ISVTX)
                printf("T");
            else
                printf("-");
        }

        // LINKS
        printf("␣%d␣", (int)fileStat.st_nlink);

        // OWNER
        if ((pwd = getpwuid(fileStat.st_uid)) != NULL)
            //fileStat.st_uid gives us the owner's ID (a number)
            //getpwuid returns a passwd structure with a field pw_name (pointer to char)
            printf("%s␣", pwd->pw_name);
        else
            printf("%d␣", fileStat.st_uid);
        //connect to TD1 when we deleted the user

        // nom du fichier et saut de ligne
        printf("␣%s\n", argv[i]);
    }

    return 0;
}
```

NOTES FOR INSTRUCTORS:
Remember to add #include < *sys/types.h* >, #include < *pwd.h* >, #include < *uuid/uuid.h* > to the code.
Check the RETURN VALUES (to explain the comparison to NULL).

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.

Listing 8: Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>

int main(int argc, char **argv)
{
    int i;
    struct stat fileStat;
    struct passwd *pwd;
    struct group *grp;
    char date[20];

    // FOR EACH FILE IN THE PARAMETERS
    for(i = 1; i < argc; i++)
    {
        // GET INFO
        if(lstat(argv[i], &fileStat)){
            //return value = -1 if error
            perror("stat");
            exit(errno);
        }

        // I-NODE NUMBER
        printf("%d␣", (int)fileStat.st_ino);

        // FILE TYPE
        switch(fileStat.st_mode & S_IFMT)
        {
            case S_IFSOCK : printf("s"); break;
            case S_IFLNK : printf("l"); break;
            case S_IFREG : printf("-"); break;
            case S_IFBLK : printf("b"); break;
            case S_IFDIR : printf("d"); break;
            case S_IFCHR : printf("c"); break;
            case S_IFIFO : printf("p"); break;
            default : printf("?");
        }

        // PERMISSIONS
        // permissions for the owner
        if((fileStat.st_mode & S_IRUSR) == S_IRUSR)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWUSR) == S_IWUSR)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXUSR) == S_IXUSR){
            if((fileStat.st_mode & S_ISUID) == S_ISUID)
                printf("s");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISUID) == S_ISUID)
                printf("S");
            else
                printf("-");
        }
        // permissions for the group
        if((fileStat.st_mode & S_IRGRP) == S_IRGRP)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWGRP) == S_IWGRP)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXGRP) == S_IXGRP){
            if((fileStat.st_mode & S_ISGID) == S_ISGID)
                printf("s");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISGID) == S_ISGID)
                printf("S");
            else
                printf("-");
        }

        // permissions for other users
        if((fileStat.st_mode & S_IROTH) == S_IROTH)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWOTH) == S_IWOTH)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXOTH) == S_IXOTH){
            if((fileStat.st_mode & S_ISVTX) == S_ISVTX)
                printf("t");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISVTX) == S_ISVTX)
                printf("T");
            else
                printf("-");
        }

        // LINKS
        printf("␣%d␣", (int)fileStat.st_nlink);

        // OWNER
        if ((pwd = getpwuid(fileStat.st_uid)) != NULL)
            //fileStat.st_uid gives us the owner's ID (a number)
            //getpwuid returns a passwd structure is a fiel pw_name (pointer to char)
            printf("%s␣", pwd->pw_name);
        else
            printf("%d␣", fileStat.st_uid);
        //connect to TD1 when we deleted the user

        // GROUP
        if ((grp = getgrgid(fileStat.st_gid)) != NULL)
            //follow the same principle
            printf("%s␣", grp->gr_name);
        else
            printf("%d␣", fileStat.st_gid);

        // nom du fichier et saut de ligne
        printf("␣%s\n", argv[i]);
    }

    return 0;
}
```

2.8 Adding the file size

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

Listing 9: Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>


int main(int argc, char **argv)
{
    int i;
    struct stat fileStat;
    struct passwd *pwd;
    struct group *grp;
    char date[20];

    // FOR EACH FILE IN THE PARAMETERS
    for(i = 1; i < argc; i++)
    {
        // GET INFO
        if(lstat(argv[i], &fileStat)){
            //return value = -1 if error
            perror("stat");
            exit(errno);
        }

        // I-NODE NUMBER
        printf("%d␣", (int)fileStat.st_ino);

        // FILE TYPE
        switch(fileStat.st_mode & S_IFMT)
        {
            case S_IFSOCK : printf("s"); break;
            case S_IFLNK : printf("l"); break;
            case S_IFREG : printf("-"); break;
            case S_IFBLK : printf("b"); break;
            case S_IFDIR : printf("d"); break;
            case S_IFCHR : printf("c"); break;
            case S_IFIFO : printf("p"); break;
            default : printf("?");
        }

        // PERMISSIONS
        // permissions for the owner
        if((fileStat.st_mode & S_IRUSR) == S_IRUSR)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWUSR) == S_IWUSR)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXUSR) == S_IXUSR){
            if((fileStat.st_mode & S_ISUID) == S_ISUID)
                printf("s");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISUID) == S_ISUID)
                printf("S");
            else
                printf("-");
        }
        // permissions for the group
        if((fileStat.st_mode & S_IRGRP) == S_IRGRP)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWGRP) == S_IWGRP)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXGRP) == S_IXGRP){
            if((fileStat.st_mode & S_ISGID) == S_ISGID)
                printf("s");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISGID) == S_ISGID)
                printf("S");
            else
                printf("-");
        }
        // permissions for other users
        if((fileStat.st_mode & S_IROTH) == S_IROTH)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWOTH) == S_IWOTH)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXOTH) == S_IXOTH){
            if((fileStat.st_mode & S_ISVTX) == S_ISVTX)
                printf("t");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISVTX) == S_ISVTX)
                printf("T");
            else
                printf("-");
        }

        // LINKS
        printf("␣%d␣", (int)fileStat.st_nlink);

        // OWNER
        if ((pwd = getpwuid(fileStat.st_uid)) != NULL)
            //fileStat.st_uid gives us the owner's ID (a number)
            //getpwuid returns a passwd structure is a fiel pw_name (pointer to char)
            printf("%s␣", pwd->pw_name);
        else
            printf("%d␣", fileStat.st_uid);
        //connect to TD1 when we deleted the user

        // GROUP
        if ((grp = getgrgid(fileStat.st_gid)) != NULL)
            //follow the same principle
            printf("%s␣", grp->gr_name);
        else
            printf("%d␣", fileStat.st_gid);

        // FILE SIZE
        printf("%d␣", (int)fileStat.st_size);

        // nom du fichier et saut de ligne
        printf("%s\n", argv[i]);
    }

    return 0;
}
```

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.

Listing 10: Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
```

```c
#include <sys/stat.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>

int main(int argc, char **argv)
{
    int i;
    struct stat fileStat;
    struct passwd *pwd;
    struct group *grp;
    char date[20];

    // FOR EACH FILE IN THE PARAMETERS
    for(i = 1; i < argc; i++)
    {
        // GET INFO
        if(lstat(argv[i], &fileStat)){
            //return value = -1 if error
            perror("stat");
            exit(errno);
        }

        // I-NODE NUMBER
        printf("%d␣", (int)fileStat.st_ino);

        // FILE TYPE
        switch(fileStat.st_mode & S_IFMT)
        {
            case S_IFSOCK : printf("s"); break;
            case S_IFLNK : printf("l"); break;
            case S_IFREG : printf("-"); break;
            case S_IFBLK : printf("b"); break;
            case S_IFDIR : printf("d"); break;
            case S_IFCHR : printf("c"); break;
            case S_IFIFO : printf("p"); break;
            default : printf("?");
        }

        // PERMISSIONS
        // permissions for the owner
        if((fileStat.st_mode & S_IRUSR) == S_IRUSR)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWUSR) == S_IWUSR)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXUSR) == S_IXUSR){
            if((fileStat.st_mode & S_ISUID) == S_ISUID)
                printf("s");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISUID) == S_ISUID)
                printf("S");
            else
                printf("-");
        }
        // permissions for the group
        if((fileStat.st_mode & S_IRGRP) == S_IRGRP)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWGRP) == S_IWGRP)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXGRP) == S_IXGRP){
            if((fileStat.st_mode & S_ISGID) == S_ISGID)
                printf("s");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISGID) == S_ISGID)
                printf("S");
            else
                printf("-");
        }

        // permissions for other users
        if((fileStat.st_mode & S_IROTH) == S_IROTH)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWOTH) == S_IWOTH)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXOTH) == S_IXOTH){
            if((fileStat.st_mode & S_ISVTX) == S_ISVTX)
                printf("t");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISVTX) == S_ISVTX)
                printf("T");
            else
                printf("-");
        }

        // LINKS
        printf("␣%d␣", (int)fileStat.st_nlink);

        // OWNER
        if ((pwd = getpwuid(fileStat.st_uid)) != NULL)
            //fileStat.st_uid gives us the owner's ID (a number)
            //getpwuid returns a passwd structure is a fiel pw_name (pointer to char)
            printf("%s␣", pwd->pw_name);
        else
            printf("%d␣", fileStat.st_uid);
        //connect to TD1 when we deleted the user

        // GROUP
        if ((grp = getgrgid(fileStat.st_gid)) != NULL)
            //follow the same principle
            printf("%s␣", grp->gr_name);
        else
            printf("%d␣", fileStat.st_gid);

        // FILE SIZE
        printf("%d␣", (int)fileStat.st_size);

        // DATE
        strftime(date, 20, "%b␣%d␣%H:%M", gmtime(&fileStat.st_mtime));
        printf("%s␣", date);

        // nom du fichier et saut de ligne
        printf("%s\n", argv[i]);
    }

    return 0;
}
```

NOTES FOR INSTRUCTORS:
Don't forget to add #include $< time.h >$.
Explain them the day formats. Use man strftime as support.

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

Listing 11: Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <dirent.h>

int main(int argc, char **argv)
{
    struct dirent *file;
    struct stat fileStat;
    struct passwd *pwd;
    struct group *grp;
    char date[20];
    DIR *directory;
    char fileName[256];

    if(argc < 2) exit(-1);

    if((directory = opendir(argv[1])) == NULL) {
        perror("opendir");
        exit(errno);
    }

    while((file = readdir(directory)) != NULL)
    {
        snprintf(fileName, sizeof fileName, "%s%s%s", argv[1], "/", file->d_name);


        if(lstat(fileName, &fileStat)) {
            perror("stat"); exit(errno);
        }


        printf("%d␣", (int)fileStat.st_ino);


        switch(fileStat.st_mode & S_IFMT)
        {
            case S_IFSOCK : printf("s"); break;
            case S_IFLNK : printf("l"); break;
            case S_IFREG : printf("-"); break;
            case S_IFBLK : printf("b"); break;
            case S_IFDIR : printf("d"); break;
            case S_IFCHR : printf("c"); break;
            case S_IFIFO : printf("p"); break;
            default : printf("?");
        }

        // PERMISSIONS
        if((fileStat.st_mode & S_IRUSR) == S_IRUSR)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWUSR) == S_IWUSR)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXUSR) == S_IXUSR){
            if((fileStat.st_mode & S_ISUID) == S_ISUID)
                printf("s");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISUID) == S_ISUID)
                printf("S");
            else
                printf("-");
        }


        if((fileStat.st_mode & S_IRGRP) == S_IRGRP)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWGRP) == S_IWGRP)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXGRP) == S_IXGRP){
            if((fileStat.st_mode & S_ISGID) == S_ISGID)
                printf("s");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISGID) == S_ISGID)
                printf("S");
            else
                printf("-");
        }


        if((fileStat.st_mode & S_IROTH) == S_IROTH)
            printf("r");
        else
            printf("-");
        if((fileStat.st_mode & S_IWOTH) == S_IWOTH)
            printf("w");
        else
            printf("-");
        if((fileStat.st_mode & S_IXOTH) == S_IXOTH){
            if((fileStat.st_mode & S_ISVTX) == S_ISVTX)
                printf("t");
            else
                printf("x");
        }
        else{
            if((fileStat.st_mode & S_ISVTX) == S_ISVTX)
                printf("T");
            else
                printf("-");
        }


        printf("␣%d␣", (int)fileStat.st_nlink);

        if ((pwd = getpwuid(fileStat.st_uid)) != NULL)
            printf("%s␣", pwd->pw_name);
        else
            printf("%d␣", fileStat.st_uid);

        if ((grp = getgrgid(fileStat.st_gid)) != NULL)
            printf("%s␣", grp->gr_name);
        else
            printf("%d␣", fileStat.st_gid);

        printf("%d␣", (int)fileStat.st_size);

        // DATE
        strftime(date, 20, "%b␣%d␣%H:%M", gmtime(&fileStat.st_mtime));
        printf("%s␣", date);

        printf("%s\n", file->d_name);
    }

    if(errno != 0) {
        perror("stat"); exit(errno);
    }

    return 0;
}
```

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?

Listing 1: Solution

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    //Variables
    pid_t pid;

    //Create chid process
    pid = fork();

    if(pid == 0)
    {
        //Code for the child
        printf("Hello, I am the child process, my process id is %d"
               ", and my parent's id is %d\n", (int)getpid(), (int)getppid());
    }else{
        //Code for the parent
        printf("Hello, I am the parent process, my process id is %d"
               ", my parent's id is %d, and I am the parent of process %d\n",
               (int)getpid(), (int)getppid(),pid);
    }

    return 0;
}

/*
Answer to the question: Sometimes the child process prints that its parent
 id is 1. The reason is that the parent process has completed its execution
 and the root process (init) takes control of orphan processes. Process init
 has PID=1.
*/
```

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

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

Listing 2: Solution

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    //Variables
    pid_t pid;
    int status;

    //Create chid process
    pid = fork();

    if(pid == 0)
    {
        //Code for the child
        printf("Hello,␣I␣am␣the␣child␣process,␣my␣process␣id␣is␣%d"
               ",␣and␣my␣parent's␣id␣is␣%d\n", (int)getpid(), (int)getppid());
        exit(0);

    }else{
        //Code for the parent
        printf("Hello,␣I␣am␣the␣parent␣process,␣my␣process␣id␣is␣%d"
               ",␣my␣parent's␣id␣is␣%d,␣and␣I␣am␣the␣parent␣of␣process␣%d\n",
               (int)getpid(), (int)getppid(),pid);
    }

    pid=wait(&status);

    printf("Child␣with␣id␣%d␣completed␣and␣returned␣%d\n",pid,status);

    return 0;
}

//The parent process may then issue a wait system call, which suspends the execution of th
```

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.

Listing 3: Solution

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    //Variables
    pid_t pid;
    int status;

    //Create chid process
    pid = fork();
    if(pid == 0)
    {
        printf("CHILD:␣execution␣starts\n");
        sleep(2);
        printf("CHILD:␣nap␣ends\n");
        exit(0);

    }else{
        //Code for the parent
        sleep(1);
        printf("PARENT:␣else␣starts\n");
    }

    printf("PARENT:␣waiting␣for␣child\n");

    pid=wait(&status);

    printf("PARENT:␣execution␣ends\n");

    return 0;
}
```

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.

Listing 4: Solution

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/file.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <fcntl.h>
//Create a struct to share the results
struct shared{
    int ab;
    int cd;
    int ef;
```

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

```
};

void error_and_die(const char *msg){
    perror(msg);
    shm_unlink("MYSHM");
    exit(EXIT_FAILURE);
}

int main(){

    //Variables
    pid_t pid1, pid2, pid3;
    int a,b,c,d,e,f;
    int r;

    //Get input from the user
    printf("input␣a:");
    scanf("%d",&a);
    printf("input␣b:");
    scanf("%d",&b);
    printf("input␣c:");
    scanf("%d",&c);
    printf("input␣d:");
    scanf("%d",&d);
    printf("input␣e:");
    scanf("%d",&e);
    printf("input␣f:");
    scanf("%d",&f);

    //Parent process creates the shared memory
    int fd = shm_open("MYSHM", O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
    if(fd==-1)
        error_and_die("shm_open");

    //Parent defines the size of the shared memory
    r=ftruncate(fd, sizeof(struct shared));
    if(r!=0)
        error_and_die("ftruncate");

    //Parent process maps the shared memory
    struct shared* shared_data = mmap(NULL, sizeof(struct shared), PROT_READ | PROT_WRITE,
    if(shared_data == MAP_FAILED)
        error_and_die("mmap");

    //Clean the house
    close(fd);

    //Create child process and do computations
    pid1 = fork();
    if(pid1 == 0){
        shared_data->ab =a+b;
        printf("a+b=%d\n",shared_data->ab);
        exit(EXIT_SUCCESS);
    }else { //Back to parent process
        pid2 = fork();
        if(pid2 == 0){
            //Inside process (c+d)
            shared_data->cd = c+d;
            printf("c+d=%d\n",shared_data->cd);
            exit(EXIT_SUCCESS);
        }else{//Back to parent process
            pid3 = fork();
            if(pid3 == 0){
                //Inside process (e+f)
                shared_data->ef = e+f;
                printf("e+f=%d\n",shared_data->ef);
                exit(EXIT_SUCCESS);
            }
        }
    }

    //Wait child process termination
    int status_ab,status_cd,status_ef;
    waitpid(pid1,&status_ab,0);
    waitpid(pid2,&status_cd,0);
    waitpid(pid3,&status_ef,0);

    //Check if all child exited normally
    if(!WIFEXITED(status_ab) || !WIFEXITED(status_cd)||!WIFEXITED(status_ef)){
        shm_unlink("MYSHM");
        exit(EXIT_FAILURE);
    }

    //Calculate result
    int result = ((shared_data->ab)*(shared_data->cd))/(shared_data->ef);
    printf("Result␣is␣%d\n", result);

    //Clean the house
    //Parent de-attaches the memory
    r=munmap(shared_data,sizeof(struct shared));
    if(r != 0)
    {
        error_and_die("nunmap");
    }
    //Parent removes share memory
    r=shm_unlink("MYSHM");
    if(r != 0){
        perror("shm_unlink");
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;

}
```

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

Listing 1: Solution

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

int main(void){
    //Auxiliary variables
    pid_t pid;

    //Create child process
    pid=fork();

    if(pid==0){
        //Child's code
        printf("CHILD: still printing to the console\n");
        //Open file
        int fd=open("output.txt", O_RDWR|O_CREAT, S_IRWXU);
        //Redirect the std output
        if(dup2(fd,1)==-1){
            perror("dup2");
            exit(errno);
        }
        //Close unused fds
        close(fd);
        //Print to file
        printf("CHILD: now printing to file\n");
        exit(EXIT_SUCCESS);

    }

    //Parent's code
    exit(EXIT_SUCCESS);
}
```

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).

Listing 2: Solution

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

int main(void){
    //Auxiliary variables
    pid_t pid;
    int tube[2];
    char buf[50];

    //Set up pipe
    pipe(tube);

    //Create child process
    pid=fork();

    if(pid==0){
        //Child's code

        //Close unused fds
        close(tube[0]);

        //Short pause (simulates a long computation)
        sleep(3);

        //Send a message to the parent
        write(tube[1], "This␣is␣a␣message␣from␣the␣child",34);
        printf("CHILD:␣message␣sent\n");

        //Close fds
        close(tube[1]);

        exit(EXIT_SUCCESS);

    }

    //Parent's code

    //Close unused fds
    close(tube[1]);

    //Read message from the child
    read(tube[0],buf,34);
    printf("PARENT:␣message␣read␣--␣%s␣--\n",buf);

    //Close fds
    close(tube[0]);

    exit(EXIT_SUCCESS);
}
```

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.

Listing 3: Solution

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

int main(void){
    //Auxiliary variables
    pid_t pid;
    int tube[2];
    char buf[50];

    //Set up pipe
    pipe(tube);

    //Create child process
    pid=fork();

    if(pid==0){
        //Child's code

        //Redirect standard output
        close(tube[0]);
        dup2(tube[1],1);
        close(tube[1]);


        //Short pause (simulates a long computation)
        sleep(3);

        //Send a message to the parent
        printf("This␣is␣a␣message␣from␣the␣child\n");

        exit(EXIT_SUCCESS);

    }

    //Parent's code

    //Redirect standard input
    close(tube[1]);
    dup2(tube[0],0);
    close(tube[0]);


    //Read message from the child
    fgets(buf,34,stdin);
    printf("PARENT:␣message␣read␣%s",buf);

    exit(EXIT_SUCCESS);
}
```

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.

Listing 4: Solution

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

int main(void){
    //Auxiliary variables
    pid_t pid;
    int tube1[2], tube2[2];
    char buf[50];

    //Set up pipe
    pipe(tube1);
    pipe(tube2);

    //Create child process
    pid=fork();

    if(pid==0){
        //Child's code

        //Set up standard output
        close(tube1[0]);
        dup2(1,3); //back up standard output
        dup2(tube1[1],1);
        close(tube1[1]);

        //Set up standad input
        close(tube2[1]);
        dup2(tube2[0],0);
        close(tube2[0]);

        //Send a message to the parent
        printf("Message␣from␣the␣child\n");
        fflush(stdout);

        //Read the message from the parent
        fgets(buf,25,stdin);

        //Print the received message to the console
        write(3,"CHILD␣-␣message␣read:␣",22);
        write(3,buf,strlen(buf));

        //Close fds
        close(3);

        exit(EXIT_SUCCESS);
    }

    //Parent's code

    //Redirect standard input
    close(tube1[1]);
    dup2(tube1[0],0);
    close(tube1[0]);

    //Redirect standard output
    close(tube2[0]);
    dup2(1,3);
    dup2(tube2[1],1);
    close(tube2[1]);

    //Send a message to the child
    printf("Message␣from␣the␣parent\n");
    fflush(stdout);

    //Read the message from the child
    fgets(buf,25,stdin);

    //Print the received message to screen
    write(3,"PARENT␣-␣message␣read:␣",23);
    write(3,buf,strlen(buf));

    //Close fds
    close(3);

    exit(EXIT_SUCCESS);
}
```

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.

Listing 5: Solution

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

int main(void)
{
        //Auxiliary variables
    pid_t pid1, pid2;
        int tube[2], ret;
        char buff[20];

        //Create tube
    pipe(tube);

        pid1 = fork();
    //Create child 1
        if(pid1 == 0)
        {
                close(tube[0]);
                write(tube[1], "Message␣from␣child␣1\n", 21);
                close(tube[1]);
                exit(EXIT_SUCCESS);
        }
    //Create child 2
        pid2 = fork();
        if(pid2 == 0)
        {
                close(tube[0]);
                write(tube[1], "Message␣from␣child␣2\n", 21);
                close(tube[1]);
                exit(EXIT_SUCCESS);
        }

        close(tube[1]);

        //Read messages sent by the children
        while((ret = read(tube[0], buff, 20)) != 0)
                write(1, buff, ret);

        close(tube[0]);

        printf("%d\n",PIPE_BUF);

        return 0;
}
```

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.

Listing 6: Solution

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>

int main(void)
{
    //Auxiliary variables
        pid_t pid1, pid2;
        int tube1[2], tube2[2], i, ret1 = 1, ret2 = 1;
        char buff[20];

    //Create pipe 1
        pipe(tube1);

    //Create child 1
        pid1 = fork();
        if(pid1 == 0)
        {
                //Child 1
                close(tube1[0]);

                for(i = 0; i < 5; i++)
                {
                        sleep(1);
                        write(tube1[1], "Message␣from␣child␣1\n", 21);
                }

                close(tube1[1]);
        exit(EXIT_SUCCESS);
        }

    //Create pipe 2
        pipe(tube2);
    //Create child 2
        pid2 = fork();
        if(pid2 == 0)
        {
                //Child 2
                close(tube2[0]);

                for(i = 0; i < 5; i++)
                {
                        sleep(1);
                        write(tube2[1], "Message␣from␣child␣2\n", 23);
                }

                close(tube2[1]);
        exit(EXIT_SUCCESS);
        }

    //Parent's code

    //Close unused fds
        close(tube1[1]);
        close(tube2[1]);

        //Setting non blocking reading for the tubes
        fcntl(tube1[0], F_SETFL, O_NONBLOCK);
        fcntl(tube2[0], F_SETFL, O_NONBLOCK);

        while(ret1 != 0 && ret2 != 0)
        {
                //Read messages from child 1
                while((ret1 = read(tube1[0], buff, 20)) != 0 && ret1 != -1)
                        write(1, buff, ret1);

                //Read messages from child 2
                while((ret2 = read(tube2[0], buff, 20)) != 0 && ret2 != -1)
                        write(1, buff, ret2);

        }

    //Clean the house
        close(tube1[0]);
        close(tube2[0]);

        exit(EXIT_SUCCESS);
}
```

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.

Listing 1: Solution

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

//Auxiliary variable
int i;

//Signal handler
void sigHandler(int sigNumber)
{
        printf("Signal␣N.␣:␣%d,␣i=%d\n", sigNumber,i);
}

//Main function
int main(void)
{
        //Auxiliary variables
    pid_t pid;
        //Create the child process
        pid = fork();
        if(pid == 0)
        {
        //Child's code

        //Case 1: stop execution
        //We do not have anything to do. This is the default behavior.

        //Case 2: ignore the signal
        //signal(SIGUSR1,SIG_IGN);

                //Case 3: respond to the signal by executing a singnal handler
                //signal(SIGUSR1, sigHandler);

                //Child's computations
                for(i = 0; i < 6; i++)
                {
                        printf("I'm␣working␣hard...\n");
                        sleep(1);
                }
        exit(EXIT_SUCCESS);
        }

    /*Short pause so the child has the time to configure
      the signal handler */
        sleep(2);

        //Case 1: kill the child process
    //kill(pid, SIGKILL);


        //Case 2: send signal SIGUSR1 to the child
    kill(pid, SIGUSR1);

    //Wait for the completion of the child process
    int status;
    wait(&status);

        exit(EXIT_SUCCESS);
}
```

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.

Listing 2: Solution

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

//gobal auxiliary variables
int tube[2];
//the pipe should be declare dhere so the signal handler has access to it

//signal handler
void sigHandler(int sigNumber){
    //local auxiliary variables
    char buf[23];
    //read the message from the pipe
    read(tube[0],buf,23);
    //print the message
    printf("Message:␣%s\n", buf);
}

//main function
int main(void){

    //local auxiliary variables
    pid_t pid;

    //open the communication pipe
    pipe(tube);

    //create child process
    pid=fork();

    if(pid==0){ //child's code

        //close unused fds
        close(tube[1]);

        //install the signal handling mechanism
        signal(SIGUSR1, sigHandler);

        //Perform computation
        int i;
        for(i=0; i<6;i++){
            printf("I'm␣working␣hard...\n");
            sleep(1);
        }

        exit(EXIT_SUCCESS);
    }

    //parent's code

    //close unused fds
    close(tube[0]);

    //make a short pause so the child has time to configure the signal handler
    sleep(2);

    //send the message to the childe
    write(tube[1],"Message␣from␣the␣parent",23);

    //Send the signal SIGUSR1 to the child
    kill(pid,SIGUSR1);

    //close unused fds
    close(tube[1]);

    //wait for the completion of the child process
    int status;
    wait(&status);

    exit(EXIT_SUCCESS);

}
```

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 "$< PID > < result >$ \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.

Listing 3: Solution

```c
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

//Auxiliary variables

// stores the process id of the child process
int pid;
// stores the FDs of the pipe use to transfer messages from the parent to the child
int tubeP[2];
// stores the FDs of the pipe use to transfer messages from the child to the parent
int tubeC[2];


//Error handler routine: call this function every time an error is produced
void error_and_die(const char *msg){
    perror(msg);
    exit(errno);
}

//Parent process closing routine. This handler should be called when the parent process re
void arretPere(int signum){

    printf("\nClosing␣parent␣process\n");
    fflush(stdout);

    //i) kill child process
        kill(pid, SIGUSR2);

    //ii) close all remining fds
        if(close(tubeP[0]) == -1)
        error_and_die("close");

        if(close(tubeC[1]) == -1)
        error_and_die("close");

    exit(EXIT_SUCCESS);
```

```
}

//Child process closing routine. This handler should be called when the child process rece
void arretFils(int sigNum){

        printf("Closing␣child␣process␣with␣id=%d\n", getpid());
    fflush(stdout);

    //i) close fds
        if(close(tubeP[1]) == -1)
        error_and_die("close");
    if(close(tubeC[0]) == -1)
        error_and_die("close");

    exit(EXIT_SUCCESS);
}

//Main function
int main(void)
{
        //Auxiliary variables
    float a, b, result; //variables for the computations
        int pidResult;      //auxiliary variable for storing the pid of the child process
    char buffer[128];   //buffer to store text
        int lenBuf;         //lenght of the buffer

    //i) open pipes
        if(pipe(tubeP) == -1)
        error_and_die("pipe");
        if(pipe(tubeC) == -1)
        error_and_die("pipe");

        //ii) create child process
    if((pid = fork()) == -1)
        error_and_die("fork");

    if(pid == 0){

        //Child's code

                //Auxiliary variables
        float var1, var2;

        //i) set up singal handlers
                signal(SIGINT, SIG_IGN); //Ignore Ctr + C
                signal(SIGUSR2, arretFils); //Close on SIGUSR2

        //ii) close unused fds
                if(close(tubeP[0]) == -1)
            error_and_die("close");
                if(close(tubeC[1]) == -1)
            error_and_die("close");

        //iii) wait for data + perform calculations + send response
                while(1)
                {
                        //i) read data from the pipe
            if((lenBuf = read(tubeC[0], buffer, 127)) == -1)
                error_and_die("read");

            //ii) parce data into working variables (see function sscanf)
                        buffer[lenBuf] = '\0';
                        if(sscanf(buffer, "%f␣%f", &var1, &var2) == EOF)
                error_and_die("sscanf");

            //iii) compute the result
                        result = var1 + var2;

            //iv) build a string with the response (see function snprintf)
                        if(snprintf(buffer, 128, "%d␣%f", getpid(), result) < 1)
                error_and_die("snprintf");

            //v) write the response to the pipe
                        if(write(tubeP[1], buffer, lenBuf) == -1)
                error_and_die("write");
                }

        exit(EXIT_SUCCESS);
        }

    //Parent's code

    //i) close unused fds
        if(close(tubeP[1]) == -1)
        error_and_die("close");
        if(close(tubeC[0]) == -1)
        error_and_die("close");

    //ii) set up signal handlers
        signal(SIGINT, arretPere); //Close at Ctr + c

    //iii) wait for input, send data to child process, wait for response, print response
        while(1)
        {
                //i) read imput from the user (see function scanf)
        if(scanf("%f␣%f", &a, &b) == EOF)
            error_and_die("scanf");

        //ii) build a string with the data to transfer
                if((lenBuf = snprintf(buffer, 128, "%f␣%f", a, b)) < 0)
            error_and_die("snprintf");

        //iii) write data
                if((lenBuf = write(tubeC[1], buffer, lenBuf)) == -1)
            error_and_die("write");

        //iv) read response from the pipe
                if((lenBuf = read(tubeP[0], buffer, 127)) == -1)
            error_and_die("read");

        //v) parce response (see function sscanf)
        buffer[lenBuf] = '\0';
                if(sscanf(buffer, "%d␣%f", &pidResult, &result) == EOF)
            error_and_die("sscanf");

        //vi) print result to screen
                printf("Result:␣%f\n", result);
        }

    exit(EXIT_SUCCESS);
}
```

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

1 Meet the `exec`s

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?

Listing 1: Solution

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <errno.h>
#include <sys/wait.h>

int main(void)
{
    //Variables
        pid_t pid;
        int status;

    //Create child process
        pid = fork();
        if(pid == 0)
        {
        //Inside the child process
                //Execute ls - we need
                if(execlp("ls", "ls", "-l", NULL) == -1) {
            perror("execlp");
            exit(errno);
        }

    }else{

        //Synchronize parent and child processes
        wait(&status);

        printf("PARENT - Execution ended. Child returned : %d\n",
           WEXITSTATUS(status));
    }

        exit(EXIT_SUCCESS);
}
```

Listing 2: Notes about the exec family

```
The differences are combinations of:
```

```
1. L vs V: whether you want to pass the parameters to the exec'ed program as

- L: individual parameters in the call (variable argument list):
  execl(), execle(), execlp(), and execlpe()

- V: as an array of char* excev(), execve(), execvp(), execvpe()

2. The array format is useful when the number of parameters that are to be
sent to the exec'ed process are variable -- as in not known in advance, so
you can't put in a fixed number of parameters in a function call.

3. E: The versions with an 'e' at the end let you additionally pass an array
of char* that are a set of strings added to the spawned processes environment
before the exec'ed program launches. Yet another way of passing parameters.

4. P: The versions with 'p' in there use the environment path variable to search
for the executable file named to execute. The versions without the 'p' require
an absolute or relative file path to be prepended to the filename of the
executable if it is not in the current working directory.

WEXITSTATUS(status): returns the exit status of the child. This consists of the least signi

WIFEXITED(status): returns true if the child terminated normally, that is, by calling exit
```

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.

Listing 3: Solution

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/wait.h>

int main(void)
{
    //Variables
    pid_t pid;
    int status;
    int tube[2];
    char buffer[20];

    //Open pipe
    pipe(tube);

    pid = fork();
    if(pid == 0)
    {
        //Redirect child process input and ouputs
        close(tube[0]);  //Close std input
        dup2(tube[1], 1);//Dup pipe output to std output
        close(tube[1]);  //Close pipe output

        //Execute ls
        if(execlp("ls", "ls", "-l", NULL) == -1) {
            perror("execlp");
            exit(errno);
        }

    }else{

        //Reconfigure parent's input
        close(tube[1]);  //Close pipe output
        dup2(tube[0], 0);//Dup pipe input to std input
        close(tube[0]);  //Close pipe input

        //Synchronize parent and child processes
        wait(&status);

        //Generate output (si EOF -> retourne NULL)
        while(fgets(buffer, 20, stdin) != NULL)
            printf("%s", buffer);

        printf("PARENT␣-␣Execution␣ended.␣Child␣returned␣:␣%d\n",
               WEXITSTATUS(status));

    }

    exit(EXIT_SUCCESS);
}
```

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.

Listing 4: Solution

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/wait.h>

int main(void)
{
    //Variables
    pid_t pidC1,pidC2,pidC3;
    int tubeC1C2[2], tubeC2C3[2], tubeC3P[2];
    char buffer[20];

    //Open pipe child 1 -> child 2
    pipe(tubeC1C2);

    //Execute ls -l
    pidC1 = fork();
    if(pidC1 == 0)
    {
        //Redirect output
        close(tubeC1C2[0]);
        dup2(tubeC1C2[1], 1);
        close(tubeC1C2[1]);
        //Execute ls
        if(execlp("ls", "ls", "-l", NULL) == -1) {
            perror("execlp");
            exit(errno);
        }
    }else{
        //Back to parent's code

        //Open pipe child 2 -> child 3
        pipe(tubeC2C3);

        //Execute tail -3
        pidC2 = fork();
        if(pidC2 == 0)
        {
            //Redirect input
            close(tubeC1C2[1]);
            dup2(tubeC1C2[0], 0);
            close(tubeC1C2[0]);

            //Redirect output
            close(tubeC2C3[0]);
            dup2(tubeC2C3[1], 1);
            close(tubeC2C3[1]);

            // execute "tail -3"
            if(execlp("tail", "tail", "-3", NULL) == -1){
                perror("execlp");
                exit(errno);
            }

        }else{
            //Back to parent's code

            /*Pipe child 1 -> child 2 is not used
             by the parent process or by child 3
             we should then close it
             */
            close(tubeC1C2[0]);
            close(tubeC1C2[1]);

            //Open pipe child 3 -> parent
            pipe(tubeC3P);

            //Execute "wc -w"
            pidC3 = fork();
            if(pidC3 == 0)
            {
                //Redirect input (from tubeC2C3)
                close(tubeC2C3[1]);
                dup2(tubeC2C3[0], 0);
                close(tubeC2C3[0]);

                //Redirect output (to tubeC3P)
                close(tubeC3P[0]);
                dup2(tubeC3P[1], 1);
                close(tubeC3P[1]);

                // execution of "wc -w"
                if(execlp("wc", "wc", "-w", NULL) == -1) {
                    perror("execlp");
                    exit(errno);
                }
                }else{
                    //Back to parent's code
                    /*Pipe child 2 -> child 3 is not used
                     by the parent process.
                     We should then close it
                     */
                    close(tubeC2C3[0]);
                    close(tubeC2C3[1]);

                    //Redirect input (from pipeC3P)
                    close(tubeC3P[1]);
                    dup2(tubeC3P[0], 0);
                    close(tubeC3P[0]);

                    //Synchronize
                    int statusC1, statusC2, statusC3;
                    waitpid(pidC1,&statusC1,0);
                    waitpid(pidC2,&statusC2,0);
                    waitpid(pidC3,&statusC3,0);

                    /* Output to screen what is written on
                       pipeC3P
                    */
                    while(fgets(buffer, 20, stdin) != NULL)
                        printf("%s", buffer);
                }
            }
        }
    exit(EXIT_SUCCESS);
}
```

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.

Listing 5: Solution

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

//Variables
pid_t pid;

//Alarm handler
void alarmHandler(int sigNum)
{
        printf("Alarm␣handled:␣killing␣child␣process\n");
        kill(pid, SIGKILL);
}

//Main
int main(void)
{
    //Create child process
        pid = fork();
        if(pid == 0)
        {
                int i;

        //Starting a long lasting task
                for(i = 0; i < 10; i++)
                {
                        printf("Child␣process␣at␣work....␣(%ds)\n", i);
                        sleep(1);
                }

                printf("Fin␣du␣travail␣!\n");

    }else{
        //Back to parent process

        //Setting up the signal handler
        signal(SIGALRM, alarmHandler);

        //Setting up the timer
        alarm(5);
        printf("Alarm␣set\n");

        // attente de la fin du processus fils
        int status;
        wait(&status);

        if(WIFEXITED(status))
            printf("Child␣process␣successfully␣completed\n");
        else
            printf("Child␣process␣unsuccessfully␣completed\n");

    }

        exit(EXIT_SUCCESS);
}
```

Listing 6: Solution

```
Notes:
- The alarm call sets a timer on the process
- When the timer stops the kernel sends a SIGALRM to the process
- Since we set up a handler, the SIGALRM signal is watched
- The handler sends a SIGKILL to the child process
- Since the child process is killed, the parent
  process is waken up and exiting wait
- WIFEXITED(status) returns True if the process terminated
  normally by a call to _exit(2) or exit(3).

Remarks:
- There is another technique so set up timers
  the setitimer call: it is more generic
  you can set timers for the other 2 types
  of time (CPU and combined) but is less portable
  it is not POXI
```

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.

Listing 7: Solution

```
Notes:
- The answer is no! from the execve manual:
  Signals set to be ignored in the calling process are set to be ignored in
  the new process. Signals which are set to be caught in the calling
  process image are set to default action in the new process image.
  Blocked signals remain blocked regardless of changes to the signal
  action.  The signal stack is reset to be undefined (see sigaction(2) for
  more information).
- If the answer were yes, then the child process should catch the SIGUSR1
  signal and print the message. Since the signals are set to default actions
  (and the default action for SIGUSR1 is to terminate the process), the
  child process just stops when the parent sends the signal
```

Listing 8: Solution

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

//Variables
pid_t pid;

//Signal handler
void sigHandler(int sigNum)
{
    printf("Signal␣handled:␣%d\n", sigNum);
}

int main(void)
{
    pid = fork();
    if(pid == 0)
    {
        //Setting up signal handler
        //signal(SIGUSR1, sigHandler);
        signal(SIGUSR1, SIG_IGN);

        //Simulate execution for 10 seconds
        execlp("sleep", "sleep", "10", NULL);

    }else{

        //Give enough time to child process to set up handler
        sleep(2);

        //"SIGUSR1" stops the process
        kill(pid, SIGUSR1);

        int status;
        wait(&status);

        if(WIFEXITED(status))
            printf("Child␣process␣successfully␣completed\n");
        else
            printf("Child␣process␣unsuccessfully␣completed\n");

    }

    exit(EXIT_SUCCESS);
}
```

3.2 Ignoring signals

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

Listing 9: Solution

```
Notes:
- The answer is no! from the execve manual:
  Signals set to be ignored in the calling process are set to be ignored in
  the new process.
- Show that this is exactly the behaviour you get with SIGUSR1
- Show that on the other hand signals SIGKILL et SIGSTOP cannot be ignored
```

Listing 10: Solution

```c
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

//Variables
pid_t pid;

int main(void)
{
    pid = fork();
    if(pid == 0)
    {
        //Set up ignoring signal SIGUSR1
        signal(SIGUSR1, SIG_IGN);

        //Simulate 10 seconds of execution
        execlp("sleep", "sleep", "10", NULL);

    }else{

        //Give time to child to set up handler
        sleep(2);

        //Send the signal
        kill(pid, SIGKILL);
        printf("Signal␣sent...\n");

        int status;
        wait(&status);

        if(WIFEXITED(status))
            printf("Child␣process␣successfully␣completed\n");
        else
            printf("Child␣process␣unsuccessfully␣completed\n");

    }
    exit(EXIT_SUCCESS);
}
```