6G7Z1004 Advanced Computer Networks and Operating Systems
(Lab 2: Exploring OS concepts in Linux)

A part of this lab was adapted from a lab originally designed by Emma Norling, MMU, UK

The aim of this lab session is to familiarize yourself with the Linux version installed in the lab and learn some interesting basic and OS related Linux commands. In particular, you will investigate how *processes* are created and manipulated in C (i.e., the OS perspective).
If you have booted the machine in Windows you have to reboot and make the appropriate selection (i.e. Linux) when the boot menu appears.

Please note that some of the commands in this lab may only be executed by a SuperUser (i.e. a **root** user), so if they do not work in your lab machine you can run them in your own computer if you have Linux installed.

Task A: Basic Linux commands

(Skip this section if you are familiar with Linux commands)

Open a terminal and learn about the following commands by typing:
man command-name (manual or man page)
For example, to understand what the command **man** does you may type: **man man**

Command	Meaning
ls	List files and directories
ls -a	List all files and directories including the hidden ones
ls -al	List all files and directories including the hidden ones, and all the information about permissions.
mkdir OS	Create a new directory called "OS"
cd OS	Change directory to OS
cd	Change to home-directory
cd ~	Change to home-directory
cd ..	Change to parent directory
pwd	Display the path of the current directory
gedit file1	Create a new file called "file1" or display its content if it exists already
rm file1	Remove the file "file1"
rmdir OS	Remove the directory "OS"
cp file1 file2	Copy file1 and call it file2
mv file1 file2	Move or rename file1 to file2
cat file	Display a file
chmod file	Change file mode bits, permissions
cat > OS-concepts	Here, we use the symbol > to redirect the output of the command. E.g. to create a file named **OS-concepts** containing the list of the list of concepts learned in this unit so far, type **cat >** OS-concepts

	Then type in the names of some concepts. Press **[Return]** after each one. **Program** **Process** **Thread** **Multiprogramming** **Concurrency** Press **Ctrl-d** to stop What happens here is that the **cat** command reads the standard input (the keyboard) and the symbol > redirects the output, which normally goes to the screen, into a file called **OS-concepts** To read the contents of the file, type **cat** OS-concepts To append other concepts to this file type **cat >>** OS-concepts Then follow the same previous steps. After adding the new concepts, type **cat** OS-concepts to see the results of using the symbol >>.
cat file1 file2 **>** file3	Concatenate file1 and file2 to file3
grep keyword file	Print the lines containing the searched Keyword Example, type **grep** Thread OS-concepts (your current directory should be the same where the file OS-concepts is saved)
who	Print information about users who are currently logged in Type **who -H**

More commands can be found here: http://ss64.com/bash/

Task B: OS related Linux commands and their usage

Open a terminal and type the following command

man man

Notice that in **man** syntax the following numbers refer to:

1: executable programs or shell commands, e.g. ls, ps
2: system calls (functions provided by the Shell), e.g., kill
3: Library calls (functions within program libraries), e.g., printf

To get a dynamic real-time view of your running system and the list of processes or threads currently being managed by the Linux kernel type

top

Alternatively, you can also use the following command:

ps

To learn more about these commands examine their manual pages using **man top** (man ps).

Now, try the commands:

ps -ef

Check the PPID (Parent Process id) column, and then try

ps -axu

To find out the id of a specific process (e.g., sshd) owned by a given user (e.g. root user) type

pgrep -u root sshd

In order to kill a running process we can use the system call "kill" followed by the chosen process id.
Examine the manual page for the command kill (*man kill*). Notice that there are different types of kill commands.

Type **Kill -l** (this allows you to see the list of available signal choices)

Type **Kill –l 13** (this will display the corresponding signal name)

Now, type **Kill -9 -1**

What do you observe?

Try to kill a root user owned process, what do you observe?

Try to kill one of the processes running under your user id (UID should be your student number), what do you observe?

Try to kill a parent process, then check the status of its children processes, what do you observe?

Try to kill a child process, then check the status of its parent process, what do you observe?

Task C: Processes in C (the OS perspective)

The following section requires you to compile and run some C code. Although you might not be able to write a program in C, you should definitely be able to compile and run it, and probably even understand the source code (of a simple program) when you look at it. Even if you cannot understand the details of the C code, it is important that you get to the section at the end on process manipulation.
Use **gedit** to create a file named newproc-posix.c

gedit newproc-posix.c &

(Remember, the "&" sign allows the Terminal to accept commands while gedit is still open, which is convenient.)

and add the following code

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

int main()
{
pid_t pid;

        /* fork a child process */
        pid = fork();

        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed\n");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                printf("I am the child %d\n",pid);
                 execlp("/bin/ls","ls",NULL);
                        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                printf("I am the parent %d\n",pid);
                wait(NULL);

                printf("Child Complete\n");
                exit(0);
        }
```

}

Compile the file newproc-posix.c using the command

```
gcc  newproc-posix.c
```

Now an executable file named a.out should appear in the current directory. Type **ls** to see the new file. You can now run it using

./a.out

When using gcc to compile your programs without any options, a.out is the default output file name. We can use option **-o** to indicate the output file name as shown below.

gcc -o newproc-posix newproc-posix.c

Alternatively, you can compile your file using

gcc -Wall -o newproc-posix newproc-posix.c

-Wall option enables all compiler's warning information, which helps improving your code quality.

Now a new executable file named newproc-posix appears. Run it using the command

./newproc-posix

Examine the source code and the output of this program. What does this program do? What are the purposes of the fork and execlp system calls? HINT: type

man fork

and

man excelp

Make the child process create another process that executes the equivalent of an **ls /tmp** in a command window. (Try it with **/bin** and the current directory (i.e., **./**)

HINT: you need to edit the source file and recompile!

What happens if you make the child process execute the program ./newproc-posix (that is, itself) – you might need to place a fully qualified (absolute) filename path in your modified program source, and in order to see what is happening you need to run the command

top -d 1

in another terminal window.

Task D: Manipulating Processes from the Command Line

Use **gedit** to create a file named procs.c and add the following code to it.

```
/*
 * code for exploring processes and threads, taken from the wait(2) man page
 * of the Linux Programmmer's Manual
 */

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

int
main(int argc, char *argv[])
{
  pid_t cpid, w;
  int status;

  cpid = fork();
  if (cpid == -1) {
     perror("fork");
     exit(EXIT_FAILURE);
  }

  if (cpid == 0) {          /* Code executed by child */
     printf("Child PID is %ld\n", (long) getpid());
    printf("Parent PID is %ld\n", (long) getppid());
     if (argc == 1)
              pause();                  /* Wait for signals */
     _exit(atoi(argv[1]));

  } else {                  /* Code executed by parent */
     do {
              w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);
              if (w == -1) {
                 perror("waitpid");
                 exit(EXIT_FAILURE);
              }

              if (WIFEXITED(status)) {
                 printf("exited, status=%d\n", WEXITSTATUS(status));
              } else if (WIFSIGNALED(status)) {
                 printf("killed by signal %d\n", WTERMSIG(status));
```

```
            } else if (WIFSTOPPED(status)) {
                printf("stopped by signal %d\n", WSTOPSIG(status));
            } else if (WIFCONTINUED(status)) {
                printf("continued\n");
            }
        } while (!WIFEXITED(status) && !WIFSIGNALED(status));
        exit(EXIT_SUCCESS);
    }
}
```

Now, run this program using the command: gcc -o procs procs.c

Open a second terminal window and experiment with the use of the **top** and **ps** commands. Find the **two** processes associated with the program you have just compiled and run. Enter the command

kill **PID**

but replace **PID** with the number of the child process. (Observe the output in the window where of the process is running)
Run the program again, and this time try killing the parent process. (Observe the output in the window where of the process is running).

Try the command

kill -STOP **PID**

(Again, using the child process's number instead of **PID**)

Exercise (optional):

We would like you to create an application in **c** that provides information relating to the processor installed in the machine. To achieve this we will access the cpuinfo file contained inside the /proc directory. Files inside the proc directory are not really files. They provide indirect access to the kernel space. For example if you type the command **cat** /proc/cpuinfo you see the kernels current view of the processor architecture it is executing upon.

cat /proc/cpuinfo

Your c application should access the file and extract the processor name and model, the frequency, the number of cores and the size of the cache. This information should be printed to the screen on separate lines as follows.

```
CPU : Intel(R) Core(TM)2 Duo CPU     E8400
Clock Speed : 3.00GHz
Cores : 2
Cache : 6mb
```