EE5012 UNIT 4 – Tutorial

1. Multiple processes

As discussed earlier, you can run a script program by simply typing its name. When a shell script executes a command, or a script program, it runs another copy of the shell as a subprocess, referred to as a subshell. This subshell executes the command and terminates, handing control back to the calling shell, referred to as the parent shell.
For example, consider a script called **mainprog** that calls a script called **program1**. We will study some examples to learn the behaviour of a process and a **subprocess** that run concurrently.

1.1 **Example for a program and a subprogram executing serially**

The **mainprog** script, shown below, echoes a message to the screen and then calls the **program1** script, which echoes a message to the screen four times at one second intervals. The **mainprog** script then echoes two messages to the screen at one second intervals, and then exits. The **program1** script is also shown below.

Type in this **mainprog** script, as follows, and give it execute permission:

```
#! /bin/bash
# Demo program to call subprograms
echo 'Main program starting!'
./program1      # Here we start program1
for (( x=0 ; x < 2 ; x++ )) ;   do
   sleep 1
    echo 'I am the main program'
done
exit
```

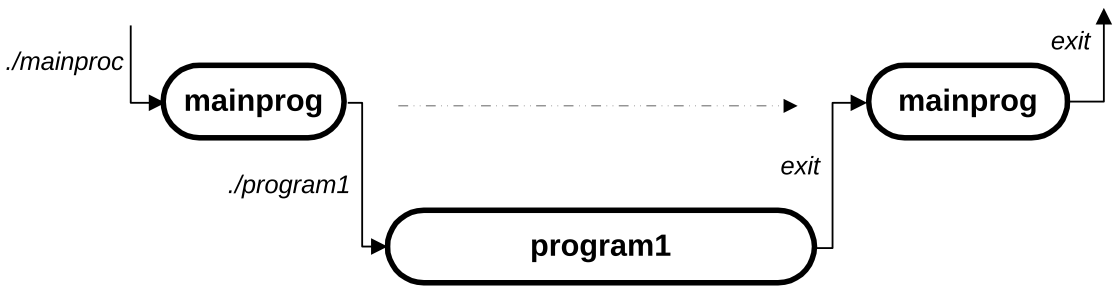
Type in the **program1** script, as follows, and give it execute permission:

```
#! /bin/bash
# announce name 4 times at one second intervals
for (( x=0 ; x < 4 ; x++ )) ; do
   sleep 1
   echo 'I am program 1'
done
exit
```

Run the **mainprog** script program on your system to make sure that it works! The output of your program will be as follows:

```
hristo@hristo-lubuntu18:~/EE5012/multi_proc$ ./mainprog
#########################
Main program starting!
#########################
Starting program 1!
I am program1
I am program1
I am program1
I am program1
#######################
Program1 exiting here!
#######################
I am the main program
I am the main program
```

The **mainprog** program execution behaviour can be plotted in a diagram as follows:



Since a UNIX/Linux operating system is multitasking, we should be able to run the subprocesses concurrently with the main process, as if we had a parallel processing system.

1.2 First attempt to execute the main program and a subprogram concurrently

We will modify the **mainprog** script, and call it **mainprog1**, so that it calls **program1** as a background process. This is achieved simply be adding the ampersand symbol '&' as shown in the bolded line below:

mainprog1

```
#! /bin/bash
# Demo program to call subprograms
echo 'Main program starting!'
./program1 &  # Here we start program1
for (( x=0 ; x < 2 ; x++ )) ;   do
   sleep 1
    echo 'I am the main program'
done
exit
```

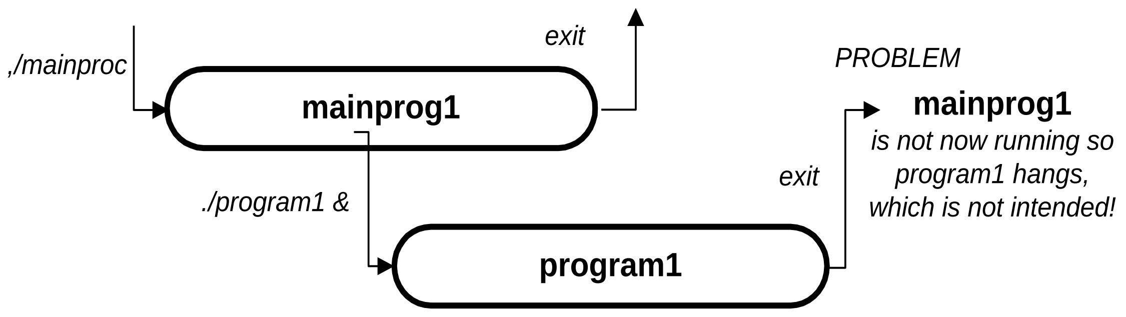
Now run this modified mainprog1 script by typing:
./mainprog1

```
hristo@hristo-lubuntu18:~/EE5012/multi_proc$ ./mainprog1
########################
Main program1 starting!
########################
Starting program1 in the background!

          I am the main program
I am program1
          I am the main program
I am program1
############################
Main program1 exiting here!
############################
hristo@hristo-lubuntu18:~/EE5012/multi_proc$ I am program1
I am program1
######################
Program1 exiting here!
######################
^C
```

A serious problem is now seen. The **mainprog1** seems to run and then exits back to the shell allowing **program1** to continue and then to hang, because its calling program, **mainprog1**, has already exited. The output is shown above, where the main program exits and leaves **program1** running and then **program1** hangs when it is finished because its calling program is no longer active.

The execution sequence for **mainprog1** is shown in the following diagram:



1.3 Fix the concurrency problem by use of the wait command

Now, we will use the **wait** command to cause the main program to wait until **program1** exits, before the main program itself exits. Modify **mainprog1** to include a **wait** command as shown below in bold font. Name the modified program **mainprog2**, as follows:

mainprog2

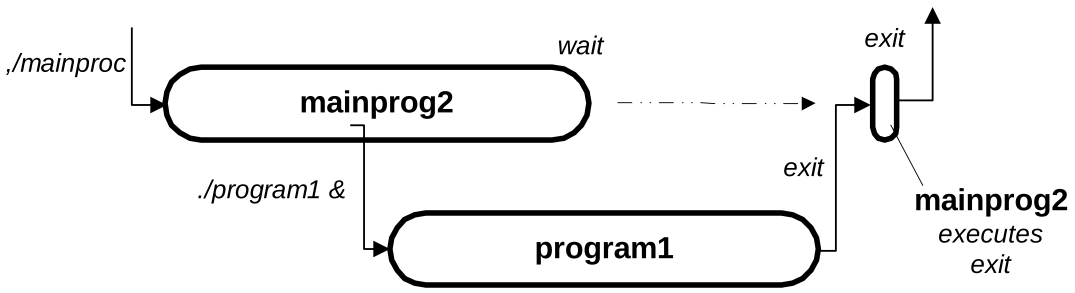
```
#! /bin/bash
# Demo program to call subprograms
echo 'Main program starting!'
./program1 &  # Here we start program1
for (( x=0 ; x < 2 ; x++ )) ;   do
   sleep 1
   echo 'I am the main program'
done
wait
exit
```

The **wait** command will wait for all subprocesses to complete before **mainprog2** completes. Run this program, by typing **./mainprog2**, and you will see a result like as follows:

```
hristo@hristo-lubuntu18:~/EE5012/multi_proc$ ./mainprog2
#########################
Main program2 starting!
#########################
Starting program1 in the background!

         I am the main program2
I am program1
I am program1
         I am the main program2
I am program1
I am program1
#######################
Program1 exiting here!
#######################
```

The execution sequence for **mainprog2** is shown in the following diagram:



1.4 Demonstrate a main program and two subprograms executing concurrently

We will now modify **mainprog2**, let’s call it **mainprog3**, to run two subprocesses in the background, as follows:

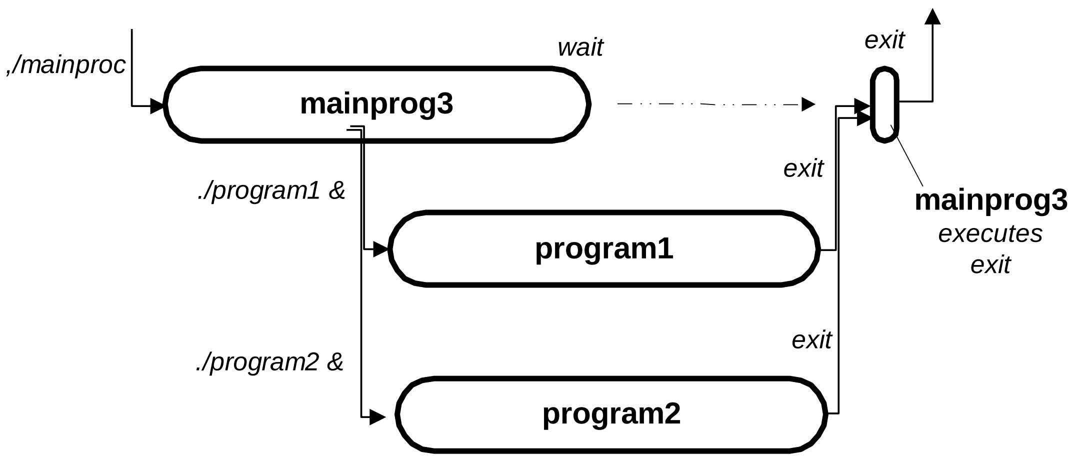
mainprog3

```
#! /bin/bash
# Demo program to call subprograms
echo 'Main program starting!'
./program1 &  # Here we start program1
./program2 & # Here we start program2
for (( x=0 ; x < 2 ; x++ )) ; do
sleep 1
echo 'I am the main program'
done
wait
exit
```

You must create the script file called **program2** which will be identical to **program1**, except it echos ‘I am program 2’ to the screen. Now run **mainprog3** and you will get a result as follows:

```
hristo@hristo-lubuntu18:~/EE5012/multi_proc$ ./mainprog3
########################
Main program3 starting!
########################
Starting program1 and program2!
                I am program2
        I am the main program3
I am program1
                I am program2
I am program1
        I am the main program3
############################
Main program3 exiting here!
############################
                I am program2
I am program1
                I am program2
I am program1
######################
Program2 exiting here!
######################
######################
Program1 exiting here!
######################
```

The execution sequence for **mainprog3** is shown in the following diagram:



The above program leads to the following questions:

1) If all three processes are writing to the same terminal (screen) then is this not confusing? The answer is that it is not normally desirable for three processes to be writing to the same terminal. Often the background processes redirect their standard outputs to a file.

2) How can you predict the sequence in which the processes will write to the terminal? The answer is that it is not easy to predict the sequence or timing of access to the terminal. Therefore we say that a ‘race condition’ problem exists here. Such race condition problems are classical problems in multitasking (concurrent) systems.

3) How do three processes run in parallel (concurrently) on a system with just a single processor (a CPU)? The answer is that we have the illusion of parallelism. In reality the same processor is being shared by switching very quickly among all of the processes.

1.5 Note on “wait” and “exit”

In the previous script examples you saw the “**wait**” and “**exit**” commands being used. The correct use of these commands is important for the multiple programs to work correctly.

When a script program calls a subprogram then there is a **parent/child** relationship between the program and its subprogram.

It is common for the parent to suspend, using a **wait** command, until the child process, or child processes, are terminated. If the parent did not wait then the child process would become an **orphan process**.

The **exit** command terminates a process. When a process calls **exit**, the kernel ensures that all file descriptors are closed, deallocates code, data and stack and the process terminates by sending a SIGCHILD signal to the parent. If the parent does not react properly, the child process can be left in a **zombie** state, i.e. it has terminated but the termination has not been recognised by the system.

2. More on UNIX/Linux pipes

We have already seen the use of a pipe to connect the standard output of one command to the standard input of another, as in the following example command:

```
ls  .  a*  |  wc  -l
```

2.1 Unnamed pipes

In this example, the **ls** command lists all the files in the current directory, which start with the letter ‘a’, and the list output is passed to the **wc** utility, where the number of lines is counted.

The **ls** utility is running concurrently with the **wc** utility. We say that there are two processes running and that the **pipe** is the *intercommunication* mechanism between the processes.
This simple pipe, as frequently used in shell commands, is referred to as an *unnamed pipe*, or an *anonymous pipe*.

The unnamed pipe is a unidirectional communication link between processes. The pipe is a simple buffer, where the writer process will block if the pipe is full and the reader process will block while waiting for an output from an empty pipe.

2.2 Named pipes

UNIX/Linux also supports a **named pipe** mechanism, which has advantages over the simple *unnamed pipe*. The named pipe name exists in the file system and thus it is accessible to all processes that have the appropriate access permissions. In contrast, the unnamed pipe is accessible only by parent/children related processes. The named pipe will continue to exist, just like a file, until it is explicitly deleted. The named pipe is designed as a unidirectional communications link, just like the unnamed pipe.

The named pipe has a FIFO (first in first out) buffer structure and is created using the **mkfifo** command, as in the following example:

```
        mkfifo  mypipe
```

If you now type the **ls -l** command you will see that the pipe exists as an entry in the file system. The **‘p’** character at the start of the line signifies that **mypipe** is a named pipe as follows:

```
prw-rw-r--  1 hristo hristo     0 Dec  6 10:51 mypipe
```

Since a named pipe is constructed to look like a file, we can use the normal UNIX read and write commands and system calls to communicate on the pipe.

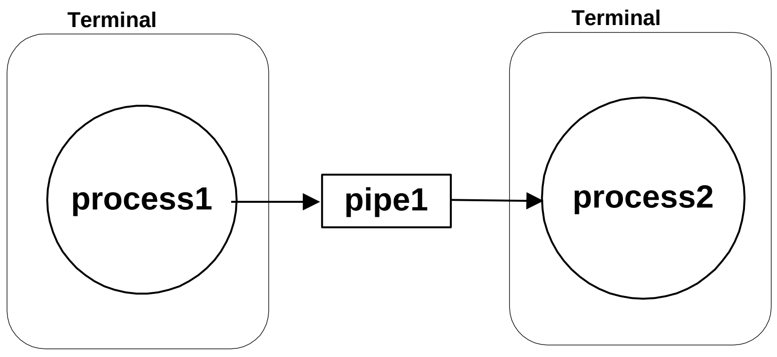
The following is a simple example where two processes are created and one communicates with the other using a named pipe. Try this example to see that it works. Note, this is a simple example of a multitasking program with a communications mechanism. It is easy to run into problems, even with simple examples, where a process might block indefinitely, waiting for an input from another process.

Instructions:

NB: this example asks you to use two terminals, running one program in each terminal.

1) Using an editor write the programs **process1** and **process2** as below.
2) Make these programs executable using **chmod +x**
3) Run these two programs (e.g. ./process1), running one program in each terminal.
4) The result should be as follows, as an output from **process2**:

```
Hello from ./process1
Hello from ./process1
Hello from ./process1
Last message!
I am finished!
```



Below is the **process1** program, it creates **pipe1**, if that pipe does not already exist, and sends a message 3 times on the pipe and then a final message.
The **$0** shell variable always represents the name of the shell or shell script.
Note that **process1** writes to the pipe as if the pipe were a simple file as follows: **echo "Hello from $0" > pipe1**.
There is nothing new to learn in terms of writing, but the program might block on writing to the pipe if the pipe has not been read. This can be tricky to debug.

```
#! /bin/bash
# Creates the pipe, if it does not already exist.
if [ !  -p pipe1 ] ; then
mkfifo pipe1
else echo “pipe already exists”
fi

# Send a message 3 times
for (( x=1; x<4; x++ )) ; do
echo "Hello from $0"  >  pipe1
```

```
sleep 1
done

# Send the last message and remove the pipe.
echo "Last message!"  >  pipe1
sleep 1
rm pipe1

exit 0
```

Here is the **process2** program, it creates **pipe1**, if the pipe does not already exist, and receives messages on the pipe, checking for the final message. Note that **process2** reads the pipe as if the pipe were a simple file as follows: **read input < pipe1**. The program will block on reading the pipe if the pipe is empty

```
#! /bin/bash
# Create the pipe, if it does not already exist.
if [ ! -p pipe1 ] ; then
 mkfifo pipe1
 else echo “pipe already exists”
fi

# Wait for last message
while [ "$input" != "Last message!" ]

# Read the pipe
do
 read  input  <  pipe1
 echo  $input
 sleep 1
done

echo I am finished!
exit 0
```

3. Job Control

Each process is assigned a unique number, referred to as a PID (**P**rocess **I**dentifier). The PID is assigned when the process is created. The shell supports the concept of foreground jobs and background jobs. The term job simply means a command that has been started from the shell command line. Normally a foreground job has control of the terminal. A program can be run as a background job by appending the **&** symbol. For example, a program called **testrun** could be run as a background job as follows:

```
testrun  &
```

The shell will respond with a message as in the following example, where 2 is the job number and 4365 is the PID:

```
$ testrun  &[2]  4365
```

When a background job completes a message such as the following is outputted:

```
[1]+   Done        testrun
```

The **jobs** command will list all of the background jobs. The **jobs -l** option will list the PIDs also.

As an example, the **jobs** command might give the following result:

```
[2]    Running          testrun  &
[3]-   Running          backup  &
[4]+  Running          account  &
```

The **fg** (foreground) command brings a background job to the foreground. By default **fg** will bring the most recent job to the foreground. The following command will put job number 2 into the foreground:

```
fg  %2
```

Alternatively the **fg %testrun** command could have been used. The most recent background job can be referred to by **%+** , while **%-** refers to the second-most-recent background job.

A foreground job can be put in the background by suspending the foreground job. This is achieved by typing **CRTL-Z** for the running job.
The job will now be suspended and put in the background.
You can use **CRTL-Z** followed by the **bg** (background) command to put a foreground job running in the background, instead of being suspended in the background.

Note - The **kill** command can also use job numbers, as follows:

```
kill  -QUIT  %3
```

4. Linux Process Priority Adjustment

The kernel stores a great deal of information about processes including process priority which is simply the scheduling priority attached to a process.
Processes with a higher priority will be executed before those with a lower priority, while processes with the same priority are scheduled one after the next, repeatedly.

There are a total of 140 priorities and two distinct priority ranges implemented in Linux.
The first one is nice value (niceness) which ranges from **-20** (highest priority value) to **+19** (lowest priority value) and the default is **0**.

The other aspect is the real-time priority, which ranges from **1** to **99** by default, then **100** to **139** are meant for user-space.
One important characteristic of Linux is dynamic priority-based scheduling, which allows nice value of a processes to be changed (increased or decreased) depending on your needs, as we'll see later on.

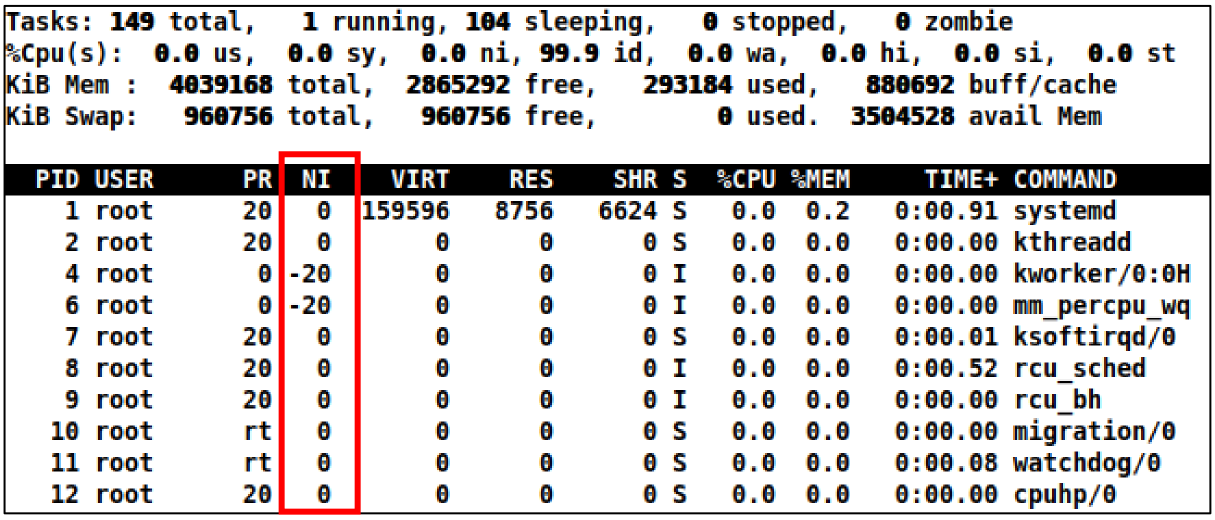
4.1 Be "nice" or how to check the *nice* value of a Linux process

To see the *nice* values of different processes, we can use **ps**, **top** or **htop**.

```
hristo@hristo-lubuntu18:~$ ps -eo pid,ppid,ni,comm
  PID  PPID  NI COMMAND
    1     0   0 systemd
    2     0   0 kthreadd
    4     2 -20 kworker/0:0H
    6     2 -20 mm_percpu_wq
    7     2   0 ksoftirqd/0
    8     2   0 rcu_sched
```

To view processes *nice* value with **ps** command in user-defined format (here the **NI** column shows *niceness* of processes).

Alternatively, you can use **top** or **htop** utilities to view the *nice* values of a process.



```
Tasks: 149 total,   1 running, 104 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni, 99.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  4039168 total,  2865292 free,   293184 used,   880692 buff/cache
KiB Swap:   960756 total,   960756 free,        0 used.  3504528 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    1 root      20   0  159596   8756   6624 S   0.0  0.2   0:00.91 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd
    4 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/0:0H
    6 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 mm_percpu_wq
    7 root      20   0       0      0      0 S   0.0  0.0   0:00.01 ksoftirqd/0
    8 root      20   0       0      0      0 I   0.0  0.0   0:00.52 rcu_sched
    9 root      20   0       0      0      0 I   0.0  0.0   0:00.00 rcu_bh
   10 root      rt   0       0      0      0 S   0.0  0.0   0:00.00 migration/0
   11 root      rt   0       0      0      0 S   0.0  0.0   0:00.08 watchdog/0
   12 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/0
```

Difference between **PR** and **NI**.

- **NI** is the *nice* value
- **PR** is the actual priority as seen by the Linux kernel

How to calculate **PR** values.

Total number of priorities = 140
Real time priority range (PR): 0 to 99
User space priority range: 100 to 139

The *nice* (NI) value ranges between: -20 to +19

PR = 20 + NI
PR = 20 + (-20 to +19)
PR = 20 + -20 to 20 + 19
PR = 0 to 39 which is the same as 100 to 139

4.2 Running a command with a predefined *nice* value in Linux

Here, we will look at how to prioritize the CPU usage of a program or command. If you have a very CPU-intensive program or task, but you also understand that it might take a long time to complete, you can set it a high or favourable priority using the *nice* command.

The syntax is as follows:

```
hristo@hristo-lubuntu18:~$ nice -n 10 ./busy_wait &
[1] 3616
```

Important:

- If no value is provided for (n), *nice* will set a priority of 10 by default
- If we run a command or a program without *nice*, its priority defaults to 0
- Only superuser or root can increase priority of a process/task
- Normal user can only lower the priority of a process

4.3 Changing the scheduling priority of a running process in Linux

As it was mentioned before, Linux allows dynamic priority-based scheduling.
Therefore, if a program or a task is already running, we can change its priority with the *renice* command.

The syntax is as follows:

```
hristo@hristo-lubuntu18:~$ renice -n 15 -p 3616
3616 (process ID) old priority 10, new priority 15
```

Any changes that you make with the *renice* command to a user'e processes nice values (**NI**) are only applicable until you reboot the machine.

For more information on *nice* and *renice,* please read the manual pages.