Operating Systems

Practical 6

Workshop on Monitoring Processes in Linux

Dr. Kevin Farrell

March 2025

Table of Contents

Legal

1 *http://creativecommons.org/*
2 *http://creativecommons.org/licenses/by-nc-nd/3.0/*

1. Introduction and Objectives

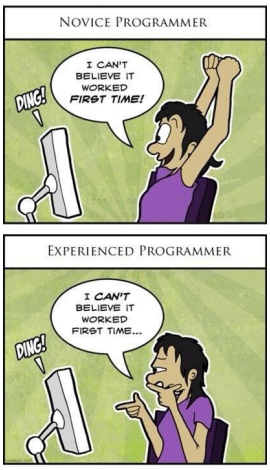
In this practical, your objective is to:

1. Revise lecture material on process creation, process hierarchy, process termination.
2. Learn some fundamentals of running programs on the command-line and how to obtain information about the processes that result when you run those programs.
3. Carry out some research to learn about process monitoring tools available in the Linux operating system, and subsequently to write a short report on your findings.

To aid you with the above, I provide some code which creates a child process from a parent process. However, a large part of this practical is a “workshop”, where *you* drive your learning.

All work **must** be carried during your practical session. You should take no more than **four** hours to complete it.

Before moving on to the next section, please create a new LibreOffice Writer document called: **StudentName-Practical06.odt**. Please use this document to take notes during the two sessions for this practical.



2. Installing the *xeyes* program

2.1 To aid you in your task of studying how to monitor processes in Linux (as detailed in the next section), in this section, you are going install a little “toy” program on your Linux system, called **xeyes**. You will then use this toy program as part of a small C program to create parent and child processes.

2.2 Open a **konsole**, log in as **root**, and install the **xeyes** program as follows:

```
# urpmi xeyes
```

2.3 Exit as root, by typing:

```
# exit
```

2.4 Check that the **xeyes** program works by running it (as an ordinary user) in the foreground:

```
$ xeyes
```

2.5 A pair of eyes should appear on-screen, and when you move your mouse cursor, the pupils of the eyes should follow it. Now quit **xeyes**, by typing:

```
CTRL-c
```

3. A Program to create a Parent and Child process

3.1 Before proceeding, open **Lecture 03 – Processes**, and review slides 23 – 30 inclusive. These describe process-creation, the process hierarchy (the process tree), the **fork()** and **exec()** system calls and process-termination.

3.2 Compare the program listing below to the one in slide 26 of Lecture 03. Write in your document the principal difference(s) between the listing below and the listing in slide 26 of the lecture.

3.3 As an ordinary user, create a directory called **practical6** in your programs directory.

3.4 Now, run Emacs, and type the program below, saving it as **parentchild.c** in your **practical6** directory.

3.5 **Note**: Do not copy and paste this code from the PDF handout, since this will result in hidden characters in the file, which will cause the compiler to give errors. Instead, you **must** type it manually.

3.6 **Note**: the comments are for explanation – you don't need to type these if you don't want to.

```
//
// This is the parentchild.c program: when run, the resulting process
// duplicates itself so that we have both a parent and a child version
// in RAM. The child version then replaces itself in RAM with a new
// process, by loading and running the xeyes program into its memory space.
//
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>

int main(){
  // Declare a variable pid, of data type pid_t. Essentially, this
  // is an integer that can store a Process ID number.
  pid_t pid;

  // Fork a child process: this duplicates this entire parentchild
  // program, so that we then have two processes in RAM:
  //     a parent "parentchild" process, and
  //     a new child "parentchild" process, which is a duplicate
  //     of the parent
  pid = fork();


  if(pid < 0){
    // ...if pid < 0 => An error has occurred with fork(): pid should
    // be greater than or equal to 0. Print an error message to
    // UNIX/Linux Standard Error device (which, by default is the console).
    fprintf(stderr,"Fork Failed\n");
    return 1;
  }
  else if(pid == 0){
    // ...the fork() system call returns a ZERO value to the CHILD process,
    // which is assigned to the variable, pid. So, if pid = 0
    // => this version of the process is the child
```

```
        // Get the child to print a message to the console
        printf("Child: I am running...\n");

        // The execlp() system call replaces the duplicated "parentchild"
        // process code in RAM (the child process) with the code from
        // the xeyes program, found in the /usr/bin directory
        execlp("/usr/bin/xeyes","Xeyes",NULL);
    }
    else {
        // ...the fork() system call returns the child's Process ID (a positive
        // integer) to the PARENT process

        // Put the parent to sleep for 5 seconds. We do this just to give
        // time for the child's print statement appear on the console first
        // It's not actually needed - Kevin thought this was a good idea! :-)
        sleep(5);

        // After sleeping for 5 seconds, get the parent to print some messages
        printf("Parent: I am running...");
        printf("Parent: I am waiting until Child is terminated by the user\n");

        // The wait() system call forces the parent to wait for the child
        // to terminate. The parent process does not proceed beyond this
        // statement until the parent receives a signal from the child
        // that it (the child) has terminated
        wait(NULL);

        // This printf() statement is executed after the parent finishes waiting
        // That is, after the child has been terminated, and has sent a signal
        // to the parent
        printf("Parent: finished waiting because child terminated by user\n");
    }
    return 0;
}
```

3.7 Now, compile the program from within Emacs, creating an executable called **parentchild** – check Practical 3 for how to do this!

3.8 Assuming your program compiled successfully, you are now ready to go on to the next section.

4. Monitoring Processes: Some fundamentals

4.1 Use your new **`parentchild`** program to aid your study of monitoring processes in Linux. In particular, as some initial learning, I would like you to investigate the following aspects of ***running***, ***stopping*** and ***killing*** (terminating) a console-based/terminal-based[3] process in Linux. Please read through **all** of these points **before** starting because these points are all inter-related:

4.2 Run your **`parentchild`** program in the ***foreground*** in the console (terminal). **Note**: a foreground process is one which occupies the console. This means that you can't type any other commands in that console until it terminates. To do so, simply type:

```
./parentchild
```

4.3 As you will probably realise, you have been running foreground processes all along this semester! We'll look at running a process in the ***background*** later in the handout.

4.4 Write in your document what you see when you run the program.

4.5 Stop your **`parentchild`** program running on the console using **CTRL-Z**. **Note**: A **stopped** process is **not terminated**. It simply means the process is moved from Memory (RAM), and put into Virtual Memory (Swap partition on Linux) – See slides 10 and 11 of **Lecture 03 - Processes**. It then waits to be resumed at some future time – usually, it is the user who resumes the process: see **`fg`** and **`bg`** below.

4.6 Write in your document, what has changed about the behaviour of the program. **Hint**: do the pupils of the **`xeyes`** program move? If not, why do you think that is?

4.7 List stopped and running jobs using the **`jobs`** command:

```
$ jobs
```

4.8 Write in your document what you see from the above listing.

3 *A terminal-based process is a program/process run from a terminal/console/konsole. The console is the original and proper name for the terminal or what is called* ***konsole*** *in the Plasma Desktop (formerly KDE)*

4.9 Once your **`parentchild`** and associated **`xeyes`** process is stopped, you can then run commands on the same console (i.e. where you started them) to list the current jobs. **Note**: you can only run commands *after* you stop the (foreground) process on the console.

4.10 Resume your process with either **`fg`** or **`bg`**: first, try using **`fg`** to bring the stopped process back into the ***foreground***. Stop it again using **CTRL-Z**, and then use **`bg`** to put the process in the ***background***. After you have put it into the background, use the **`jobs`** command again.

4.11 Make a note in your document what you see from the above listing, and compare it to the earlier **`jobs`** listing when the program was stopped; making some notes of the differences.

4.12 Now use the **`ps`** command (on the same console) to see more detailed information about the processes on that console, by typing the command below. Note the absence of a "**-**" preceding the letter "a":

```
$ ps a
```

4.13 Write into your document the output that you see from the above command. Do some searching on the Internet to find out what the column-headings in the out mean.

4.14 Kill a foreground program with **CTRL-C**: run **`parentchild`** in the foreground again, and then kill it with **CTRL-C** to see what happens.

4.15 In your document, make a note of what happened.

4.16 Run your **`parentchild`** in the foreground again. Open a second **konsole** window, and list the processes in your system to see even more detailed information, using:

```
$ ps au
```

4.17 Describe in your document what you see; in particular, what is different. See if you can find on the Internet what the various additional column headings signify.

4.18 Run your **`parentchild`** program in the ***background*** using a trailing "**&**" symbol:

```
./parentchild &
```

4.19 Notice how you now have access to the same **konsole** window, unlike in the case of when you ran **parentchild** in the foreground.

4.20 Use the **`jobs`** command again, and the **`ps au`** command (on the same console) to see information about the processes on that console.

```
$ jobs
$ ps au
```

4.21 What do you notice that is different, if anything, compared to before?

4.22 Repeat the above step, running **`parentchild`** in the background ***multiple*** times on one konsole, and then running the above **`ps`** command on a second konsole:

```
$ ./parentchild &
$ ./parentchild &
$ ./parentchild &
```

4.23 Since the parentchild program prints some information to the console, use a second **konsole** application, to open up a new **console**, and view your running processes by typing:

```
$ ps au
```

4.24 Once more, describe in your document what you see, and compare it to previous outputs.

4.25 Kill a program (from the second **konsole**) using:

```
$ kill PID
```

- where **`PID`** is the process' Process ID obtained using the "**`ps au`**" command

4.26 Kill a program, or multiple copies of it, (from the second **konsole**) using:

```
$ killall programname
```

4.27 Examples:

```
$ killall xeyes
$ killall parentchild
```

4.28 In each case, describe in your document, what happens.

5. Researching Process Monitoring Tools

5.1 Once you are comfortable with the various ways to run, stop and kill programs, as mentioned above, start carrying out the tasks described below:

1. Research and study at least ***three*** process monitoring tools available for your Linux system. **Note**: these must be ***process monitoring tools***, not network monitoring tools or other unrelated tools. One of these tools **must** be a command-line process monitoring tool, and one **must** be a process monitoring tool that runs as a graphical application from your GUI. For the third process monitoring tool, the choice is yours – either graphical or console-based.
2. You may need to install some of the tools. You can do this by using the Mageia Software Installer. Alternatively, you could research how to install the software from source code – this is a nice challenge to set yourself! If you run into problems and need help, ask me.
3. For each tool, run it on your system, and investigate and learn how it works. Use your **parentchild** program to help you understand the monitoring tools. That is, use your process monitoring tools to identify your running **parentchild** process and its child/children **xeyes** process(es), and information about these
4. In your document, write a report[4], ***in your own words***, which summarises each tool according to:
    - How to execute/run it.
    - What kind of information it provides.
    - What options (if any) are available with it, to obtain more/enhanced information about the processes on your system. You do not have to cover every single option. Approximately, three to four options will do.

4 *Using **LibreOffice** on your Mageia Linux system, **NOT** on Windows!*

5. In your report, write a ***detailed*** description (with screen and/or window snapshots) of ***one*** of the tools you research. **Note**: you can take screen and window snapshots using the KDE **spectacle** tool.

5.2 Paste your document (including your report) at the end of your Reflective Journal, and submit it through the link on BrightSpace before the deadline. The text of the report part of your document must be **no more** than 600 words long.