National University of Singapore
School of Computing

CS2106 **Laboratory 3** Semester 1 11/12

Deadline

9 September, 2011, Friday, 11:59pm.

Submission

Download the template for your solution:

```
wget http://www.comp.nus.edu.sg/~cs2106/lab03-A000000X.txt
```

Rename the file by replacing the string A000000X with your matriculation number.

Enter your answer into the text file and submit the file into IVLE Workbin (a folder named Lab 3) before the deadline.

Reading

Reading Sections 1.11, 10.3.1, and 10.3.2 from the textbook would be helpful before attempting this lab exercise.

Platform

This exercise requires the Linux platform. You *must* use the Linux machines in the OS Lab (since different versions of Linux will give different answers).

Marking Scheme

The marks for each question are indicated besides each question.

Marks are deducted for the following:

- 3 marks for not naming your file properly.
- 3 marks for submitting your answer in incorrect format (not in ASCII text, not using the template).
- 0.1 marks for every minute late after deadline.

1. (4 points) **System Call Overhead.** In this exercise, you will run a little experiment to measure the overhead of system calls. We will focus on two particular calls: `fork()`, which is suppose to be expensive, and `getpid()`, which is one of the simplest system call available.

   Download the measurement code with the command:

   ```
   wget http://www.comp.nus.edu.sg/~cs2106/lab03-time.c
   ```

   The code above measures the CPU time spent in the process executing the code marked between `/* start timing */` and `/* end timing */`.

   To compile the code, use the following command:

   ```
   gcc -O0 lab03-time.c -lrt
   ```

   The flag `-lrt` links the executable binary to the library `librt.a`, which provides the implementation of function `clock_gettime`. The flag `-O0` (that's an "oh" character, followed by a "zero") tells the compiler not to optimize our code.

   (a) (1 point) Let's begin by measuring the time it takes to execute the function `getpid()`, which returns the value of the current process ID.

   Modify the program above to measure the *total* CPU time spent calling `getpid()` 1,000,000 times. Note down the time in milliseconds (ms).

   (b) (1 point) On Linux, the implementation of `getpid()` does not result in a system call. The function caches the process ID, so that repeated calls to `getpid()` does not result in expensive system call. To bypass this optimization and invoke the system call directly, we can use the following code:

   ```
   syscall(SYS_getpid);
   ```

   Modify the program above to measure the *total* CPU time spent calling `syscall(SYS_getpid)` 1,000,000 times. Take note of the time (in ms) printed by the program. Note that your program will give you different answers everytime your run your program, but they should be in the same order of magnitude.

   (c) (1 point) Based on the experiment above, what is the average overhead (in ns) of making a system call compared to a function call?

   (d) (1 point) Now, modify the program above to measure the *total* CPU time spent executing the following line of code 1,000 times.

   ```
   if (fork() == 0) return;
   ```

   Does the code above measure the CPU time of `fork()` at the parent process or the child process? What is the average time taken to `fork()`, in ns?

   WARNING: DO NOT (NEVER EVER) DO THE FOLLOWING

   ```
   for (i = 0; i < 1000; i++) {
           fork();
   }
   ```

   which will create 2^{1000} processes. See Tutorial 2 Question 1.

2. (4 points) **Orphan and Zombie Process.**

(a) (2 points) Enter and compile the following program into a executable called `zombie`.

```
/* zombie.c */
#include <sys/types.h>
#include <unistd.h>

int main()
{
        if (fork() > 0) {       /* Line A */
                while (1);
        }
        return 0;
}
```

In your shell, execute `zombie` in the background (append your command with `&`):

```
./zombie &
```

Now run `ps` to examine your processes:

```
ps -f
```

You should see a process labeled as `defunct`. What does this mean? Explain how the program above would lead to a defunct process (also known as zombie process).

(b) (0 points) To see why such a process is called zombie process, issue the command `kill` to terminate the process, by passing in the process ID of the defunct process. For example, if the defunct process has process ID 1234, issue command:

```
kill 1234
```

Run `ps -f` again to see if the defunct process is still there.

To return to norm, kill the parent of the defunct process.

(c) (2 points) Make a copy of `zombie.c` and name it `orphan.c`. In `orphan.c`, change the line labelled `Line A` to:

```
if (fork() == 0) {
```

Compile and name your executable `orphan`. In your shell, execute `orphan` in the background, and examine your processes just like Part (a).

What is the name of the parent process of the process `orphan`?

Explain why the parent process is different from the parent process of the non-defunct process `zombie` in Part (a).

3. (2 points) **Introducing Process Group.**

In Linux, a process group is a collection of one or more processes, identified by a process group ID (PGID). Each process group has a leader. The process group ID of a process group is the process ID of its leader.

The command `ps -ef` does not print PGID of the processes by default. To ask `ps` to print out the PGIDs along with other information, use the following command, which specifies what information about a process to print on the screen.

```
ps o user,pid,ppid,pgid,comm
```

Enter and compile the following program into a executable called `eight`.

```
/* eight.c */
#include <unistd.h>

int main()
{
        int i;
        for (i = 0; i < 3; i++)
                fork();
        while (1);
}
```

As you learnt from your Tutorial 1, this program will create a total of eight processes. Now observe the PGID of the processes created by running the executable `eight`.

(a) (1 point) What processes consitute the process group created? Which process is the leader?

(b) (1 point) You have learnt that you can use the `kill` command (or similarly, the `kill()` system call) to send a signal to a process. The same `kill` can be used to send a signal to a process group. Read the man page of the `kill` command to find out how to send a signal to a process group.

Note down the command.

THE END