CS2106 Operating Systems

Lab 2 – Fork and Pipe

1. Introduction and Administrative Details

In this lab we will explore how to use fork() to create new processes, and pipes to communicate between processes.

This lab is a graded lab, and you should submit your report at the end of your lab session to the IVLE workbin created for your lab group.

Team Setup

You should work in groups of 2 (recommended) or 3 (maximum) to finish this lab. Single person submissions are not allowed.

Lab Platform

This lab is to be done on Ubuntu 16.04 either running directly on your computer, or on a virtual machine. Your lab is not guaranteed to work correctly on any other platform. See Lab 1 on how to install Ubuntu 16.04 on Oracle VM VirtualBox.

Submission Instructions

Decide who within the team is the team leader. Name your lab report <student-number>.docx, where <student-number> is the Student Number on the team leader's student card.

Use the template provided in cs2106lab2ans.docx for your report.

Upload your lab report to the IVLE workbin folder for your lab group by Sunday 25 February, 2359.

ENSURE THAT EVERY TEAM MEMBER'S STUDENT NUMBER AND NAME IS IN THE REPORT.

SUBMIT TO THE FOLDER FOR YOUR LAB GROUP!. **REPORTS SUBMITED TO THE WRONG FOLDER WILL NOT BE GRADED AND YOU WILL RECEIVE ZERO (0) FOR THIS LAB.**

2. Getting Familiar with Fork

The POSIX fork() call creates a new process. When successful the parent process will create a new child process. The fork() call returns the process ID (pid) of the child to the parent, and returns a 0 to the child.

Launch a terminal in Ubuntu, and using your favorite editor (e.g. vim), KEY IN the following code. Name your program "lab2p0.c".

```c
#include <stdio.h>
#include <unistd.h>

int main()
{
	int cpid;
	if((cpid=fork()) != 0)
	{
		// Tjis is the parent.

		printf("Hello, I am the PARENT. My PID is %d, my child's PID is %d, and
my parent's PID is %d\n",
			getpid(), cpid, getppid());
	}
	else
	{
		printf("Hello, I am the CHILD. My PID is %d, my parent's PID is %d, fork
returned %d.\n",
			getpid(), getppid(), cpid);
	}
}
```

Compile and run your program with:

gcc lab2p0.c –o lab2p0
./lap2p0

Question 1. List down the following:

Parent's process ID:
Child's process ID:
Child's parent's process ID (as reported by child):
Parent's parent's process ID (as reported by parent):

The "ps" command lists processes that are running and their PIDs. Type "ps –A" (without quotes, and note the space between ps and –A) to list all process IDs.

Which process is parent of the parent?

3. Building a Web Server

In this lab package (cs2106lab2.zip) you will find a file called lab2p1.c. The lab2p1.c file contains the code for a web server.

Launch a terminal in Ubuntu, and cd to the directory where you unzipped cs2106lab2.zip to.

Step 1.

Open the lab2p1.c file and look through the code to familiarize yourself with it. In particular you should pay attention to the following code inside the startServer function:

```
        while(1)
        {
                connfd = accept(listenfd, (struct sockaddr *) NULL, NULL);
                writeLog("Connection received.");

                deliverHTTP(connfd);
        }
```

This block of code accepts a new connection, then calls deliverHTTP to deliver the requested HTML file.

You will find the following code fragment in deliverHTTP. It reads the incoming connection and parses it to get the HTTP request type (GET, PUT, POST and HEAD) and the requested HTML file (the parsing isn't very robust, but this is an operating systems class, not a networks programming class).

```
void deliverHTTP(int connfd){

                … OTHER CODE …

        read(connfd, HTTPBuffer, MAX_BUFFER_LEN);

        int method;
        char filename[MAX_FILENAME_LEN];
        char fetchName[MAX_FILENAME_LEN];

        parseHTTP(HTTPBuffer, &method, filename);
        printf("Method = %d filename = %s\n", method,filename);

        … OTHER CODE …

}
```

Note that the read(.) call is a blocking call; i.e. execution does not proceed past the read(.) until data comes in (or an error occurs).

To compile:

gcc lab2p1.c –o lab2p1

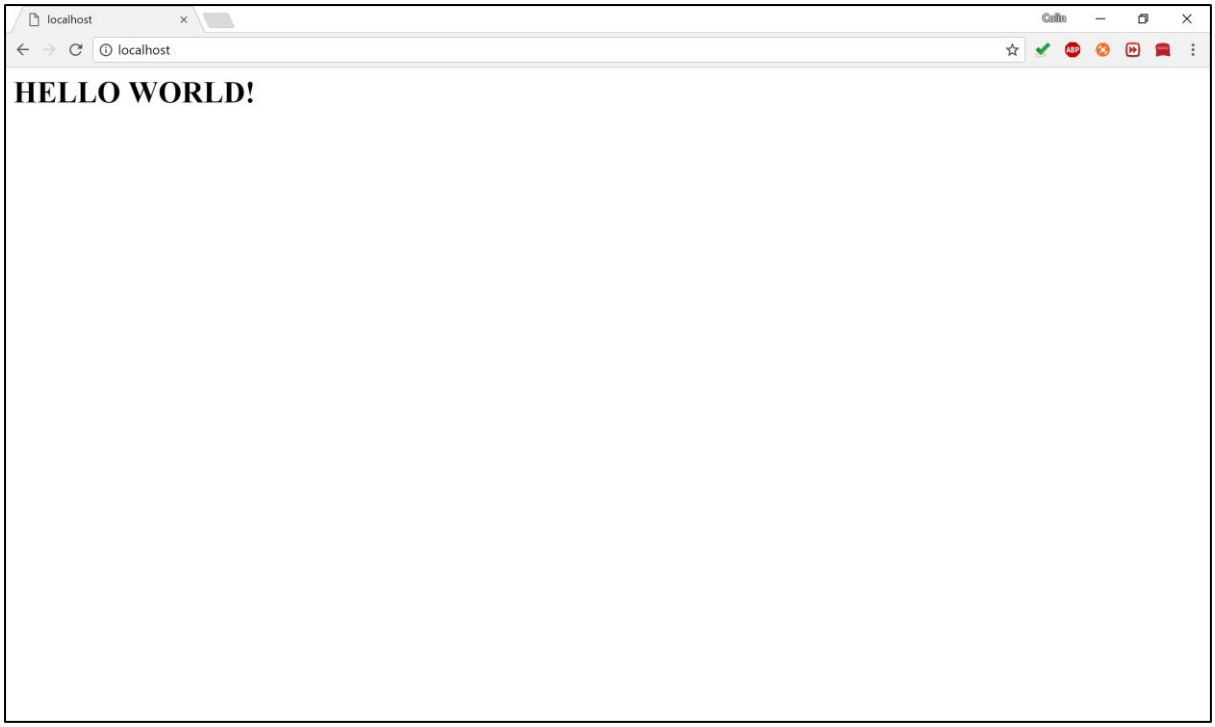
Now type "make" to compile the web server. To run it, type:

sudo ./lab2p1

Note that you must sudo to run this server as it binds to port 80 (the default HTTP port). Binding to any port with a number of less than 1024 requires superuser (root) privileges.

Now start Firefox or Chrome within your Ubuntu installation, and in the address bar type in http://127.0.0.1.

You will see the following appear:



This verifies that your web server is running. Now press CTRL-C in the terminal running lab2p1 to break the web server's execution.

Step 2.

Open two terminals and change to the directory with lab2p1 in it.. In the first one, type:

sudo ./lab2p1

This, as before, will start the web browser.

In the second terminal, type:

telnet localhost 80

(If your Ubuntu system says that telnet is not found, install telnet by typing:

sudo apt-get install telnet

And try the step above again.)

Now from within Chrome or Firefox, once again type "127.0.0.1" in the address bar.

Question 2.

You will find that when telnet has an existing open connection the web server, Chrome (or Firefox) cannot connect to it. Based on what you understand of the web server's code (particularly the code for deliverHTTP), explain why.

Step 3.

You will now use what you learnt about fork() to allow our web server to accept multiple simultaneous connections.

Question 3a. Modify the web server using fork() to allow it to accept multiple connections. Cut and paste the modifications you made in your report and explain them.

Question 3b. By opening up multiple terminals, find out what is the largest number of connections you can make to your web server. Explain why your web server can only accept this number of connections.

4. Getting Familiar with Pipes

We are now going to work with Unix pipes. Pipes are special, "unnamed" files that the OS uses to pass data from one process to another.

A pipe has an "input" end and an "output" end. A process reads from the input end, while another process writes to the output end.

To begin with pipes, declare an integer array of 2 elements to store the input and output file descriptors for the pipe:

```
int fd[0];
```

To create the pipe, use the POSIX pipe function:

```
pipe(fd);
```

The reading process should close the output end of the pipe, which is fd[1]:

```
close(fd[1]);
```

The writing process should close the input end of the pipe, which is fd[0]:

```
close(fd[0]);
```

In your Ubuntu terminal, create a new file called lab2p2.c and KEY IN the following code

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

// Maximum size of our comms buffer
#define MAX_BUFFER_LEN            1024
```

```c
// We will use a pipe to send data from one process to another
int main()
{
	int fd[2];

	pipe(fd);

	// Our buffer
	char buffer[1024];

	// # of characters written/read
	int n;

	// We are sending from parent to child
	if(fork() == 0)
	{
		// Child should close the output side of the pipe
		close(fd[1]);

		// Read from the pipe
		n = read(fd[0], buffer, MAX_BUFFER_LEN);
		printf("Child read %d bytes from parent: %s\n", n, buffer);
		close(fd[0]);
	}
	else
	{
		int status;
		// Parent should close input side of the pipe
		close(fd[0]);

		// Write to the pipe
		sprintf(buffer, "Hello child! This is your parent!");
		n = write(fd[1], buffer, strlen(buffer)+1);
		printf("Parent wrote %d bytes to the child: %s\n", n, buffer);
		close(fd[1]);

		// Wait for the child to end
		wait(&status);
	}
}
```

Answer the following questions:

Question 4a.

Why does the sending process have to close the input side of the pipe, while the receiving process has to close the output side of the pipe? (Hint: It has nothing to do with corrupting data in the pipe).

Question 4b.

Since both child and parent process have access to the "buffer" array, there is no need for them to use pipes to communicate with each other. The parent can just write to the array and the child can just read from it.

Explain why this statement is false.

5. Using Pipes in the Web Server

We will now modify the web server from Step 3 in Section 3 of this lab, which has been modified to allow multiple connections.

Opening up your modified lab2p1.c, you can find a function called writeLog that takes information from the server to be logged, and writes it to stdout. This function uses variable argument lists, but we will not explain that here. You are welcome to google about va_list to find out more.

The writeLog function is shown below:

```
void writeLog(const char *format, ...)
{
        char logBuffer[LOG_BUFFER_LEN];
        va_list args;

        va_start(args, format);
        vsprintf(logBuffer, format, args);
        va_end(args);

        printf("%s: %s\n", getCurrentTime(), logBuffer);

}
```

We will now modify lab2p1.c so that:

i) All logs are now written to log.txt instead of to stdout.
ii) Writing to log.txt is done by a single child process, listening to a pipe.
iii) The writeLog function now writes to the pipe instead of stdout.

Question 5.

Modify your lab2p1.c program so that it achieves the changes listed above. Cut and paste the modifications to your report and describe what you have done.

Some hints on how to do this:

It would help a lot to declare the pipe file descriptors as a global variable.

Fork the child process responsible for logging inside main().

Remember to close the ends of the pipe you are not using.

SUBMISSION REMINDER:

1. Deadline is Sunday 25 February, 2359.
2. Please name your report <student-number>.docx, where <student-number> is the Student Number on the team leader's Student Card.
3. Ensure that every team member's Student Number and name is in the report.
4. SUBMIT TO THE FOLDER FOR YOUR LAB GROUP! **REPORTS SUBMITTED TO THE WRONG FOLDER (OR WRONG LAB GROUP'S FOLDER) WILL NOT BE GRADED AND YOU WILL RECEIVE ZERO (0) FOR THIS LAB!**