COMS3010A
Operating Systems

Lab 1 - Process API

August 8, 2022

Instructors

COMS3010A Lecturer:

Branden Ingram	branden.ingram@wits.ac.za

COMS3010A Tutors:
William Hill 2115261
Andrew Boyley 2090244
Oluwatimileyin Obagbuwa 2134111
Sheslin Naidoo 2094701
Derrin Naidoo 2127039
Phindulo Makhado 1832463
Mohammed Gathoo 2089236
Sedzani Ramathaga 2083519
Ryan Alexander 1827474
Talion Naidoo 1448771
Jonathan Nunes 2087190
Phoenix Krinsky 2233063
Christopher Walley 1846871
Alice Govender 1847313

Consultation Times

Questions should be firstly posted on the moodle question forum, for the Lecturer and the Tutors to answer. If further explanation is required consultation times can be organised.

1 Introduction

You need access to a Unix machine and know how to use the C compiler. If you are on Windows I suggest - https://docs.microsoft.com/en-us/windows/wsl/install-win10 the Windows Subsystem for Linux. The small examples that you will implement are simple but if you have done them once, it will hopefully help you understand the operations of the operating system and how it can control processes.

For this lab, you will be brushing up on your C skills and exploring how a compiled executable is structured. Part of this course requires you to solve problems independently. You are encouraged to consult the resources we link to and search the internet for any specific problems you encounter; Googling is a skill. Nevertheless, the tutors are available should you get stuck. You will require a “C” compiler for this lab namely “gcc”. The following link can be used to help you install the required packages: https://www.guru99.com/c-gcc-install.html. However, the MSL labs already have gcc installed.

2 MakeFiles

You may have used *gcc* to compile your programs from the command line, but this grows tedious and complicated as the number of files you need to compile increases. Makefiles simplify the compilation process into a simple call to "make". If this is new to you, consult the following tutorial: https://makefiletutorial.com/ I have provided some code to download from Moodle called "Lab1Files". First, create a file called "Makefile" in your Lab1Files directory and edit it. We have provided basic code below that compiles hello world.c:

```
all: hello_world

hello_world: hello_world.c
	gcc hello_world.c -Wall -Wextra -pedantic -o hello_world

clean:
	rm -f hello_world
```

Note that the indentation must be a tab, **not** spaces! After saving your Makefile, executing "make" from the terminal should produce the "hello world" executable. Calling "make clean" should remove the executable.

2.1 Hello World

There are several differences between C and C++, most notably in string handling and IO. For example:

```
int val = 2001;
char word[] = "student";
printf("I am a %s doing COMS%d\n",word,val);
```

The formatting options presented above are string (%s) and signed integer (%d). Also note that there are no built-in string objects in C, you use null-terminated char arrays. Open hello_world.c and look at how to manipulate strings using strcpy and strcat. Then, complete the program by making it print the length of the string.

2.2 Arguments

In arguments.c, you will notice that main takes in two parameters: "argv": the array of parameters passed to the program. For example: "arguments file1.txt file2.txt", *argv* is then ["arguments","file1.txt","file2.txt"]. "argc": the number of parameters passed to the program and will always be at least 1, since the program name is always the first parameter. Complete arguments.c to print the executable name and the arguments passed to it in a comma-separated list. Strictly adhere to the output format. Example terminal output1:

- ./arguments asdf qwerty
- Program name: ./arguments
- Arguments: asdf,qwerty

Example terminal output2:

- ./arguments
- Program name: ./arguments
- Arguments: NONE

3 CPU Virtualisation

Within the files provided you will find a file called “cpu.c” compile and run the executable produced. Then run the same executable in an additional terminal. You can also open up the “c” file to see what it is doing. This exercise is meant to demonstrate how the OS virtualises our CPU given both our processes the appearance they are the only thing running on the actual hardware. In reality we know that while both the identical copies of the same executable are running seemingly in parallel they are actually being swapped in and out of the CPU multiple times.

4 Fork

A C function that calls the fork syscall to create a new process by duplicating the calling process. The new process, referred to as the child, is an almost exact duplicate of the calling process, referred to as the parent: the parent and the child have identical data, stack, and heap segments and share the text segment of their address space. A fork() call returns twice: once to the child and once to the parent. If the call has been successful, the PID of the child process is returned to the parent, and 0 is returned to the child. On failure, -1 is returned to the parent, and no child process is created. We will use the library procedure fork() so first take a look at the manual pages. You will probably not understand everything they talk about but we get the important information that we need to start experimenting. In a terminal type the following command.

- man fork

We see that fork requires the library “unistd.h” so we need to include this in our program. We also read that fork will return a value of type “pid_t”. This type is defined in a header file included by “unistd.h” and is a way of making the code architecture independent. We will ignore this and assume that “pid_t” is a “int”. Further down the man pages we read that fork returns both the process identifier of the child process and zero. This is strange, how can a procedure return two different values? Let’s give it a try, create a file called fork_test.c and write the following:

```
#include <stdio.h>
#include <unistd.h>
int main(){
int pid = fork();
printf("pid = %d\n" , pid);
return 0;
}
```

The above program will call fork and then print the returned value. Compile and run this program.

5 The mother and the child

So the call to fork() somehow creates a duplicate of the executing process and the execution then continues in both copies. By looking at the returned value we can determine if we’re executing in the mother process or if we are in the child process. Try this extension to the program.

```
#include <stdio.h>
#include <unistd.h>
int main(){
        int pid = fork();
        if(pid == 0){
                printf(" I'm the child %d\n", getpid());
        }
        else{
                printf("My child is called %d\n", pid);
        }
        printf("That's it %d\n" , getpid());
        return 0;
}
```

This is not the only way this could have been implemented. One could for example have chosen to have a construction where we would provide a function that the child process would call. Different operating systems have chosen different strategies and Windows for example have chosen to provide a procedure that creates a new process that is independent of the mother process.

6 Wait

A class of C functions that call syscalls that are used to wait for state changes in a child of the calling process and obtain information about the child whose state has changed. The following state changes are recognised:

- the child terminated
- the child was stopped by a signal
- the child was resumed by a signal

To terminate the program in a more controlled way we can have the mother wait() for the child process to terminate. Try the following and save it in a fill called wait_test.c:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main( ){
        int pid = fork();
        if(pid == 0){
                printf( " I'm the child %d\n" , getpid());
                sleep(1);
        }
        else{
                printf("My child is called %d\n",pid);
                wait(NULL);
                printf( "My child has terminated\n");
        }
        printf("That's it %d\n" , getpid());
        return 0 ;
}
```

The mother waits for its child process to terminate (actually it waits for any child it has spawned). Only then will it proceed, print out the last row and terminate.

6.1 returning a value

A process can produce a value (an integer) when it terminates and this value can be picked up by the mother process. If we change the program so that the child process returns 42 as it exits, the value can be picked up using the wait() procedure. Modify your previous code with the following:

```
if(pid == 0){
        return 42;
}
else{
        int res = -5;
        wait(&res) ;
        printf("the result was %d\n" , WEXITSTATUS(res));
}
```

6.2 zombies

A zombie is a process that has terminated but whose parent process has not yet been informed. As long as the parent has not issued a call to wait() we need to keep part of the child process. When calling wait, the parent process should be able to pick up the exit status of the child process and possibly a return value. If the child process is completely removed from the system this information is lost. We can see this in action if we terminate the child process but wait for a while before calling wait(). Do the following changes to the program, call it zombie.c, compile and run it in the background.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main( ){
        int pid = fork();
        if(pid == 0){
                printf("check the status\n");
                sleep(10);
                printf("and again\n");
                return 42;
        }
        else{
                sleep(20);
                int res;
                wait(&res) ;
                printf("the result was %d\n" , WEXITSTATUS(res));
                printf("and again\n");
                sleep(10);
        }
        return 0 ;
}
```

Check the status of the processes using the ps command. Notice how the two processes are created, how the child becomes a zombie and is then removed from the system once we have received the return value.

- gcc -o zombie zombie.c

- ./zombie&

- ps -ao pid,stat,command

6.3 a clone of the process

So we have created a child process that is a clone of the mother process. The child is a copy of the mother with an identical memory. We can exemplify this by showing that the child has access to the same data structures but that the structures are obviously just copies of the original data structures. Extend fork_test.c and try the following:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main(){
        int pid;
        int x = 123;
        pid = fork();
        if(pid == 0){
                printf("child: x is %d\n", x);
                x = 42;
                sleep(1);
                printf("child: x is %d\n", x);
        }
        else{
                printf("mother: x is %d\n", x);
                x = 13;
                sleep(1);
                printf("mother: x is %d\n", x);
                wait(NULL);
        }
return 0;
}
```

As you see both the mother and the child sees variable x as 123 but the changes made are only visible by themselves. If you want to see something very strange you can change the printout to also print the memory address of the variable x. Do this for both the mother and the child and you will see that they are actually referring to the same memory locations.

```
printf("child : x is %d and the address is 0x%p\n" , x , &x);
```

The explanation is that processes use virtual addresses and they are identical, they are however mapped to different real memory addresses. We will go deeper in virtual addresses in the upcoming weeks.

6.4 what we do share

Since the child process is a clone of the mother process we do actually share some parts. On thing that we do share are references to open files. When a process opens a file a "file table entry" is created by the operating system. The process is given a reference to this entry and this reference is stored in a "file descriptor table" that is owned by the process. Now when the process is cloned, this table is copied and all the references are of course pointing to the same entries in the "file table". The standard output is of course nothing more than a entry in the “file descriptor table” so this is why both processes can write to the standard output. We also read from the same standard input. If you look at man pages you will see a whole range of structures that the processes share or not share but most of those are not very interesting to us in this set of experiments.

7 Signals

Signals are primarily used by the operating system to signal to processes that something has happened that probably needs some attention. It could also be used in between processes or even inside a process to raise an exception. If you open a shell (terminal or command prompt) you can list all possible signals by using the command “kill”. As you can see there are quite a few, but don’t be afraid, you do not have to learn them by heart.

- kill -l

You might have used the “kill” command when you wanted to kill a process but the command will be able to send any signal to a process. When no signal is given the “SIGTERM” signal is sent to the process. You might have learned to write “kill -9” when you really wanted to kill something; what is signal number 9 called? When you hit “ctrl-c” in a terminal window, the shell will send a “INT”(interrupt) signal to the foreground processes. There is some magic taking place when you’re running a program in a shell and the “ctrl-c” will terminate the program but not the shell; all will be crystal clear in a couple of weeks when we look into threads. To learn a bit more about how signals work we will write small examples and see how they behave.

8 catching a signal

Normally when you run a program the process inherits the default signal handlers of its creator. The process can then set its own signal handlers to change the behavior of the process. One example is when a process wants to do some final things before terminating when it receives the “SIGTERM” signal.

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

int volatile count;
void handler(int sig){
        printf("signal %d ouch that hurt \n" , sig);
        count++;
}
int main(){
        struct sigaction sa;
        int pid = getpid();
        printf("ok, let's go ,kill me (%d) if you can!\n" , pid);
        sa.sa_handler = handler;
        sa.sa_flags = 0;
        sigemptyset(&sa.sa_mask);
        if(sigaction(SIGINT, &sa, NULL) != 0){
                return(1);
        }
        while(count != 4){
        }
        printf(" I've had enough! \n");
        return(0);
}
```

Copy this program, call it “test1.c”, compile and execute it in a terminal. In this terminal try to kill it by hitting ctrl-c. The program does not do very much besides trying to stay alive. If we take a look at how it achieves this we see that it uses a “sigaction” structure and a call to “sigaction”. The structure holds some parameters to control what will happen. We will set the “sa_handler” property to point to our own handler procedure. We will then clear the “sa_mask” entry because we do not want to block any other signals when we're in the handler. The interesting thing is the call to “sigaction”, here we pass three arguments: the signal we want to handle, a pointer to the sigaction structure and a null pointer (that we don't have to bother about now). This is the library call that will change our signal table and when a “INT” signal is sent we will be able to do what we want. The signal handler itself will in this example will not do anything useful, only print a message so that we know that it was invoked. Note that one should not use a system library call such as printf inside a handler since this might be in conflict with an ongoing library call. We do so in this exercise to keep things as simple as possible.

Run the program and do some experiments; can you kill it by sending a “kill -SIGINT” from another terminal? What happens if you send it a “SIGTERM” signal? Look up “kill” and “sigaction” using the “man” command. There is probably a lot more here than you would ever want to know but you should get the habit of reading the “man’ pages. This is the simple way to catch a signal; using a slightly different technique we can get some more information on what is going on.

8.1 catch and throw

In Java you're quite used to declaring what exception methods could cause and you could always trap them in an exception handler to possibly do something else. When you're programming in C there is no such support in the language, but it turns out that you can use signals to handle exceptions. Let's catch any exception caused by division by zero. The handler will simply print an error message and then exit but we could of course have done something to save the situation. Create another c file and call it “test2.c”, within it you will be adding the following segments of code.

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int sig){
        printf("signal %d was caught \n", sig);
        exit(1);
        return;
}
```

We need to install the handler and do so by registering it under the “SIGFPE” signal. Then we call a not so good procedure that will divide by zero (note, it's integer division so it will generate a fault).

```
int not_so_good(){
        int x = 0;
        return 1 % x;
}
int main(){
        struct sigaction sa;
        printf("Ok, let's go I'll catch my own error. \n");
        sa.sa_handler = handler;
        sa.sa_flags = 0;
        sigemptyset(&sa.sa_mask);
        /* and now we catch... FPE signals */
        sigaction(SIGFPE, &sa , NULL);
        not_so_good();
        printf(" Will probably not write this. \n");
        return(0);
}
```

You might ask yourself how a division by zero is detected as a fault and how it is turned into a signal to the user process. The fault is first detected by the hardware; when the instruction is executed an exception is raised. The OS will then look in an "Interrupt Descriptor Table" (IDT) and jump to a location in memory that hopefully contains some code that will take care of the problem. This code is part of the kernel so the kernel decides what to do. In the case of a division by zero the user process that generated the fault must of course be interrupted. If the process has registered its own "SIGFPE" handler, as in the case above, control is passed to this function; the default procedure is to kill the process.

9 Who do you think you are

If we use the simple version of signal handler then there is not much information to go on; a signal has been sent but we don't know much more. We can however use a slightly more elaborate call that gives us some more information. To use this version we set a flag in the sigaction structure. Create a new file called "test3.c" and complete the following code segments.

```
int main(){
        struct sigaction sa;
        int pid = getpid();
        printf("Ok, let's go kill me (%d).\n" ,pid);
        /* we're using the more elaborated sigaction handler */
        sa.sa_flags = SA_SIGINFO;
        sa.sa_sigaction = /* INSERT HERE */
        sigemptyset(&sa.sa_mask);
        if(sigaction(/* INSERT HERE */, &sa ,NULL) != 0){
                return(1);
        }
        while(!done){
        }
        printf("Told you so! \n");
        return(0);
}
```

Nothing strange there, but now the handler will be passing three arguments: the signal number, a pointer to a "siginfo_t" structure and a pointer to a context that we can ignore.

```
int volatile done;
void anotherhandler(int sig, siginfo_t *siginfo, void context){
        printf("signal %d was caught \n", sig);
        printf("your UID is %d\n", siginfo->si_uid);
        printf("your PID is %d\n", siginfo->si_pid);
        done = 1 ;
}
```

The “siginfo_t” structure contains information about the process that sent the signal. If you start this program in one shell and then try to kill it from another shell (using “kill -SIGINT”) you should hopefully be able to tell which process that tried to kill you.

10 Submission

Please see the moodle quiz with questions relating to the work you have done in this lab.

Academic Integrity

There is a zero-tolerance policy regarding plagiarism in the School. Refer to the General Undergraduate Course Outline for Computer Science for more information. Failure to adhere to this policy will have severe repercussions.

During assessments:

- You may not use any materials that aren’t explicitly allowed, including the Internet and your own/other people’s source code.
- You may not access anyone else’s Sakai, Moodle or MSL account.

Offenders will receive 0 for that component, may receive FCM for the course, and/or may be taken to the legal office.