EECS 3221

Assignment 1 **(Version 1.1)**

Processes, Signals and Inter-process Communication.

The purpose of this assignment is to become familiar with **processes, signals and interprocess communication**. Specifically, you will create a C program to create and destroy processes and to send signals among them. Therefore you must be familiar with process creation methods and **Linux Signals.**

The first part of the assignments includes 4 questions. The second part of the assignment requires developing a program written in C language, that performs process management.

Note that you do not have to submit the answers to questions in part 1, but you should try them before you start part 2. What you learn from part 1 can help you develop and debug your C program easier.

Exploring Linux Use the command **`strace`** to trace the system calls of the execution of `wc <file>`
and answer the following questions:

(1) Why does your process read files like `ld.so*` when it starts up?

(2) Why does it read files like `*locale*` immediately afterwards?

(3) Program `wc` reads the file one chunk at a time. How big is this chunk?

(4) Which system call does not return anything?

As mentioned, you do not need to submit the answers to these questions, but do them before you start the programming part. Use this experience when you debug your C program.

1. A Program for Performance Evaluation of Signals

You will write a simple program called **`ring`** that creates a ring of **`N`** processes that send a signal round and round **`cnt`** times.

The program should be invoked from terminal with:

```
./ring N leader cnt
```

where **`leader`** is 0 when you first invoke it.

The program works as follows:

If the ***leader*** parameter is **0** then the process knows it is the **leader**. It then creates (with `fork-exec`) a child process that has parameters:

```
N-1 leader cnt
```

where now `leader` is the **PID** of the true **leader** process. The child then will create another child process with parameters:

```
N-2 leader cnt
```

where `leader` is **again** the same **PID** of the true **leader** process.

The process that eventually is created with `N=1` is the last one and so it does not create another child. Instead it sends a **`SIGUSR1`** to the **leader** to start signaling phase. Signaling will repeat for **`cnt`** rounds.

All the processes **suspend** themselves waiting for a **`SIGUSR1.`** Once they receive the signal, they send a **`SIGUSR1`** to the next. At the end of the execution the leader (only) prints the `cnt` and the true **time elapsed** since the beginning of execution as a **floating point number.**

In addition, to automate your program execution with different number of **cnt**, write a **shell script** called **`ring_time.sh`** using a **`for`** loop that runs the **`ring`** program for `N=20` processes and for **`cnt`** 10, 100, 1000 and 10000 signals. The script should write the results to the file **`timing.out`**. The format of the file **`timing.out`** is the results of one run per line. Each line contains the **`cnt`** and the time it took.

2. Skeleton Code

Please download the skeleton code provided and use it as starting point for the assignment. It includes a `Makefile` with several useful targets to help you develop and debug your code. It also includes an empty C program (just prints its PID) and an empty shell program.

3. Style

Your code should be properly **commented, indented and readable**. There should be a comment near the top of the file that includes a **short description (4-5 lines), your name and date**. Compilation should produce **no warnings on the EECS Linux systems**. All system calls that may return an error condition **should be checked**. Please keep in mind that style is very important. In addition to the comments added to the top of the file, you should add comments to lines of code that perform an important step such as making a system call. Your TA should be able to easily read and follow your code.

4. Reading

Besides the code in the notes, you should read several man pages. Pages like **kill**(2), **sigaction**(2), **fork**(2), **signal**(7), **errno**(3), **sigsuspend**(2), **clock_gettime**(2) are good starting points. Also study how to pass **SIGUSR1,** and how to use a Makefile.

5. Submission

Make sure you include all the files necessary (C code, shell code, makefile, …) for your TA to compile and run your program. Submit your program as a single **.tar** file to eClass, under assignment 1 submission. The **Makefile** included in assignment files provides a quick way to create your .tar file with the necessary files included.

Your TA will use the **EECS Linux** environment to compile and run your program.

6. Common Questions and Answers

What should the final output of the leader be? Something like `"%d %7.3e\n"`, where %d displays the **cnt** and %7.3e shows the elapsed time.

What does the `timing.out` look like? Like this:

```
10 2.402e-02
100 5.610e-02
1000 3.490e-01
10000 2.773e+00
```

Do I put anything in the signal handler? While you may put a `printf` during debugging, you should remove it before submitting your final code.

How many comments are required? A header for sure for the program. A short header for any function you define or system calls.

Is style important? Yes! Your code should be professional and easy to read.

Do I need to use `exec` after fork? Yes. It is very hard otherwise. You can use any of the functions/syscalls in the exec family.

How do I debug this code? Code with system calls and multiple child-processes is very hard to debug. Make sure that ALL system calls that may return an error condition (usually a negative number but not always) are checked and a meaningful error message is printed in the standard output. Code your program one section/feature at a time. That may make it easier to debug.

How to send a signal? There are many ways to send a signal in Linux. Start by checking the **Kill** command.

How do I pass parameters to a child process? Read the exec family of syscalls/functions manual. You need to convert the numbers to ASCII strings. You can use **sprintf** for the conversion.

Check eClass for other Q/A regarding assignment 1.