ENCE360 Lab 2: Processes and Threads

1. Objectives

The overall goal of this lab is to introduce you to creating processes and threads by having to read, compile and run some small programs written in C on the Linux operating system.

This lab should:

1. Enhance your understanding of processes and threads covered in the lecture
2. Create threads and processes using standard Linux system calls
3. Understand using threads, processes, semaphores and a mutex.

2. Preparation

Download the "Lab 2 threads and processes" zip archive from Learn.

Note that all examples can be compiled using the provided makefile. (Navigate to the Lab 2 directory in your shell and type 'make')

3. Fundamental Concepts

As explained in lectures, a thread of control or thread[1] is a sequence of instructions executed by a program. Every process contains at least one thread. A multi-threaded process is associated with two or more threads. Although these threads share most of the process resources (code, data, open files, etc.) and a common address space, they execute independently of each other, each having a program counter and a stack to keep track of local variables and return addresses.

4. Threads – mutex.c

The program mutex.c is computing the sum of a large array. Your job is to convert it into a multi-threaded program.
Compile using: gcc mutex.c -o mutex --std=c99 -lpthread

Start out by turning the call run_summation into a call to spawn a thread (pthread_create) using the same method. Check the **man pages** by typing man pthread_create this will be a common source of documentation for system calls throughout this course.

After converting it to a multi-threaded program. Run the program several times (using ./mutex) and observe the output. Is it the same as before? What are the reason(s) for this happening?

Fix the issues by making sure all threads finish before we print the final total, and by protecting the global variables with a **mutex.** Check that the fixed program returns the same result as the original single-threaded program.

1

5. Creating a new process

The system call to create a new process is called `fork()`. When executed by the parent process, `fork()` will create a child process that will have its own address (data) space, that will look identical to that of the parent process. If the call fails, -1 is returned, the reason for failure is placed in `errno`.

Using the file `fork.c`:

Read and understand this source file and in particular observe

1. the header files that are included. They are important when creating process

```
       #include <sys/types.h>
       #include <unistd.h>
```

2. Compile and run fork.c (`gcc -o fork fork.c && ./fork`)
3. Observe the output generated and fill in the missing values below (next page):

In the process of completing the above exercise be sure you understand the answers to the following questions:

1. Can you explain their values?

2. How do they relate to each process's data space?

3. Uncomment the line:

   ```
   /* waitpid(childPid, NULL, 0); */
   ```
   recompile, and execute `fork.c`. Be sure you understand how and why the behaviour of `fork.c` has changed.

```
[Start with] global: (___________) local: (__________)

[Parent] childPid:    _______       parent: _______

[Child]  childPid:    _______

[Child] global:       _______       local:  ________

[Parent] global:        (___________) local: (________)

[At end(499)] global: (___________)  local: (________)

[At end(500)] global: (___________)  local: (________)
```

6. Creating a new thread

The call to create a new thread is called `pthread_create`. When executed, `pthread_create` will create a new thread of execution. The procedure `main()`, itself, has a single thread of execution. Each thread executes simultaneously with all the other threads contained within the one calling process. For more details on threads enter the command `man pthread_create`

Using the file `thread.c`:

1. Read and understand the source file.

2. Compile this using the following command:

```
gcc -o thread thread.c -lpthread
```

3. Run thread several times and observe the output generated. Be sure you can explain the different behaviour that `thread.c` exhibits when executed.

Concurrency using semaphores – semaphore.c

semaphore.c - The threads communicate with the main thread using a semaphore based channel I have called **Chan**. Your task is to implement the operations.

First, observe the **producer**() function, and the **main**() function and understand how the **Chan** is being used to communicate between threads. Then try to implement the operations.

At any one point in time the channel is either empty, or full. (queue of length 1)

If the channel is empty:

- The read semaphore has a value of 0 (must wait to read)
- The write semaphore has a value of 1 (allowed to write immediately)

If the channel is full:

- The read semaphore has a value of 1 (allowed to read immediately)
- The write semaphore has a value of 0 (must wait to read)

The read and write operations are therefore symmetrical and proceed like this:

1. Acquire access to read or write via **sem_wait**
2. Perform read or write activity
3. Signal to opposite read or write that they can now proceed with **sem_post**

The channel should be initialized to an empty state – think about how the semaphores should be initialized and use **sem_init**