SOFE 3950U / CSCI 3020U:
Operating Systems

TUTORIAL #6: POSIX Threads Part II

Objectives

- Learn the fundamentals of multithreading
- Gain experience using POSIX threads

Important Notes

- Work in groups of **four** students
- All reports must be submitted as a PDF on blackboard, if source code is included submit everything as an archive (e.g. zip, tar.gz)
- Save the file as <tutorial_number>_<first student's id>.pdf (e.g. tutorial6_100123456.pdf)
- If you cannot submit the document on blackboard then please contact the TA with your submission at **jonathan.gillett@uoit.net**

Notice

It is recommended for this lab activity and others that you save/bookmark the following resources as they are very useful for C programming.

- http://en.cppreference.com/w/c
- http://www.cplusplus.com/reference/clibrary/
- http://users.ece.utexas.edu/~adnan/c-refcard.pdf
- http://gribblelab.org/CBootcamp

The following resources are helpful as you will need to use pthreads in order to make your program multithreaded.

- https://computing.llnl.gov/tutorials/pthreads/
- http://randu.org/tutorials/threads/
- http://pages.cs.wisc.edu/~travitch/pthreads_primer.html
- http://www.cs.rutgers.edu/~pxk/416/notes/c-tutorials/

Conceptual Questions

1. What is **fork()**, how does it differ from multi-threading (pthreads)?

2. What is inter-process communication (**IPC**)? Describe methods of performing IPC.

3. Provide an explanation of **semaphores**, how do they work, how do they differ from mutual exclusion?

4. Provide an explanation of **wait (P)** and **signal (V)**.

5. Research the main functions used for semaphores in **<semaphore.h>** and explain each function.

Application Questions

All of your programs for this activity can be completed using the template provided, where you fill in the remaining content. A makefile is not necessary, to compile your programs use the following command in the terminal. **If you do not have clang then**

replace clang with gcc, if you use gcc you must use -pthread instead of -lpthread. If you are still having issues please use -std=gnu99 instead of c99.

```
clang -Wall -Wextra -std=c99 -lpthread <program name>.c -o
<program name>
```

Example:

```
clang -Wall -Wextra -std=c99 -lpthread question1.c -o question1
```

You can then execute and test your program by running it with the following command.

```
./<program name>
```

Example:

```
./question1
```

Template

```
#define _XOPEN_SOURCE 700 // required for barriers to work
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

int main(void)
{

}
```

1. Create a program that does the following.
    - Creates a master process with **two** child processes using **fork()**
    - The master process writes two files **child1.txt** containing the line **child 1** and **child2.txt** containing the line **child 2**
    - Each of the two child processes waits **one second** then reads the contents of their text file and prints the contents.

- Reading and writing files between process is a simplified method of IPC.

2. Create a program that does the following.
    - Creates a parent and child process using **fork().**
    - The child process sleeps for 1 second and prints “Child process”.
    - The parent process instead of immediately executing, **waits** for the child process to terminate using the **wait()** function before printing “Parent process”.
    - The parent process must check the return status of the child process after it has finished waiting.
    - See the following for more information on forking and waiting: http://www.cs.rutgers.edu/~pxk/416/notes/c-tutorials/

3. Create a program that does the following.
    - Create a **global** array length five, **moving_sum**, and initialize it to zeros.
    - Prompts a user for fives numbers
    - For each number creates a thread
    - Each thread executes a function **factorial** which takes a struct containing the **number** and **index** of the number entered (0 - 4) and does the following:
        - Calculates the factorial (e.g. 5! = 5x4x3x2x1 = 120, 0! = 1).
        - **Using a semaphore**, gets the previous value in the **moving_sum[index-1]** if the value at that **index is > 0**. If the value is retrieved it is added to the factorial calculated and the sum is added to **moving_sum[index].**
        - Until the value in **moving_sum[index-1]** is > 0, performs an infinite loop, each time it must perform signal and wait to allow other threads access to the critical section.
    - After all threads finish (using **join()**) print the contents of **moving_sum**

4. The **producer/consumer problem** is a classic problem in synchronization, create a program that does the following.
    - Create a global array **buffer** of **length 5**, this is shared by producer and consumers and initialized to zero.
    - Prompts the user for **ten numbers** (store in an array use #define NUMBERS 10 for the size)
    - Creates two threads, one a producer, the other a consumer
    - The **producer thread** calls the function **producer** which takes the array of numbers from the users as an argument and does the following:

    - Loops until all ten items have been added to the buffer, each time with a **random delay** before proceeding
    - **Using semaphores** gets access to the critical section (buffer)
    - For each number added to **buffer** prints “Produced <number>”, to indicate the number that has been added to the buffer
    - If the **buffer is full,** it waits until a number has been consumed, so that another number can be added to the buffer
- The **consumer thread** calls the function **consumer** and does the following:
    - Loops until ten items have been consumed from the buffer, each time with a **random delay** before proceeding
    - **Using semaphores** gets access to the critical section (buffer)
    - For each number **consumed** from the **buffer**, sets the buffer at that index to **0,** indicating that the value has been consumed.
    - For each number consumed, also prints “Consumed <number>”, to indicate the number that has been consumed from the buffer
    - If the **buffer is empty,** it waits until a number has been added, so that another number can be consumed from the buffer
- The program waits for both threads to finish using **join()**, and then prints the contents of **buffer**, the contents of buffer should be all zeros.

5. Create a program that does the following.
    - A master process which prompts a user for five numbers and writes the five numbers to a file called **numbers.txt**
    - The master process forks and creates a child process and then **waits for the child process to terminate**
    - The child process reads the five numbers from **numbers.txt** and creates five threads
    - Each thread executes a function **factorial**, which takes the number as an argument and does the following:
        - Calculates the factorial (e.g. 5! = 5x4x3x2x1 = 120).
        - Adds the factorial calculated to a global variable **total_sum** using the **+=** operator
        - The **total_sum** must be incremented in a thread-safe manner using **semaphores**
        - Prints the current factorial value and the calculated factorial
    - The child process has a **join** on all threads and after all threads have completed writes the **total_sum** to a file called **sum.txt** and terminates

- After the child process has terminated the parent process reads the contents of **sum.txt** and prints the total sum.
- Reading and writing files between processes is one of the simplest methods of IPC.