Lab 1
Processes, Threads and Synchronization Basics

CS3210 – 2020/21 Semester 1

Learning Outcomes

1. Understand the differences between processes and threads
2. Use pthread library in shared-memory parallel programming
3. Implement critical sections in the code
4. Apply basic synchronization constructs in programs

You can obtain 2% of you the grade in CS3210 by submitting your work at the end of the lab. Full marks can be obtained for submissions that have minor bugs.

Logging in & Getting Started

For the lab and assignments, you are going to remotely connect and work on the machines in the Parallel Computing Distributed Lab located in COM1-B1-02. You will receive more details about these computers during the next class. For this lab, you are required to use your SoC id (Unix id) to connect to Sunfire. If you do not have an SoC id, you can create one at this link: `https://mysoc.nus.edu.sg/~newacct`. From Sunfire, you can remotely connect to the computers in the lab. These computers (machines) run Ubuntu 18.04 on x86_64 architecture. The hostnames are:

- `soctf-pdc-001` - `soctf-pdc-016`,
- `soctf-pdc-018` - `soctf-pdc-021`.

`Use the username & password provided to you using LumiNUS gradebook to connect to the lab machines.` To remotely connect to the lab machines (from Sunfire) use the following sample command:

```
> ssh 12345Z@soctf-pdc-018
```

You can change your password by running `passwd` in the console. Note that each account on each machine is independent (there is no automatic synchronization among machines). This means that changing password on on one machine will not change password on the other machines. Also, placing a file on the home directory of one machine will not be automatically copied on all machines (you need to manually copy).

Since all classes are done online, you need to get confortable with working in terminal (console). For this lab, connect to one of the machines, and start working on completing the tasks in the lab. Use scp, wget or git to transfer files to and from the lab machines.

The lab files can be found here: `https://www.comp.nus.edu.sg/~ccris/cs3210/L1_code.zip`

Part 1: Processes vs. Threads
(Optional for students who are familiar with processes & threads implementations)

Multi-process programming on Linux with C

Let us look at a simple code which demonstrates use of processes in Linux. Download and open `processes.c` file and study the use of `fork()` system call and its return values.

Note the `wait(NULL)` call by the parent process. The purpose of this call is to make sure the parent process waits until all its child processes are completed. In a situation where the child continues to run after the parent process is completed (died), the child is called an orphan process.

- Compile the code in a terminal (console):
  ```
  > gcc -o processes processes.c
  ```
- Run the program in a terminal (console):
  ```
  > ./processes
  ```

Exercise 1
Compile and run `processes.c`. Observe the output. Why is the line "We just cloned a process..!" printed twice? Fix the code such that the line only prints once.

Creating and terminating threads

Listing 1: Snippet of `threads.c`

```c
for(t=0;t<NUM_THREADS;t++)
{
  printf("main thread: creating thread %ld\n", t);

  //pthread_create spawns a new thread and return 0 on success
  rc = pthread_create(&threads[t], NULL, work, (void *)t);
}
```

`threads.c` contains a simple example on creating (spawning) threads with pthread library and terminating them. In threads.c, the loop runs `NUM_THREADS` number of times and calls pthread_create function to create/spawn new threads.

`pthread_create` takes in four arguments:

1. `thread` – Reference to a thread variable of type pthread_t (element in threads array in this example)
2. `attr` – Thread attributes
3. `start_routine` – The function to be executed by the newly spawned thread (function work in this example)
4. `arg` – Arguments to be passed on to the parallel function (t in this example). Please note that instead of passing a single variable, we could pass a structure when multiple arguments are required by the parallel function.

- To find out more about different C functions, you can use the manual in command line:
  ```
  > man 2 pthread_create
  ```
- Compile the code in a terminal (console):
  ```
  > gcc -pthread -o threads threads.c
  ```
- Run the program in a terminal (console):
  ```
  > ./threads
  ```

Exercise 2
Compile and run the threads.c program. Observe the output. Modify the `NUM_THREADS` value and observe the order of thread execution. Do threads execute in the same order they are spawned each time the program runs? Is the final value of the variable counter always the same? Explain.

Part 2: Process and Thread Synchronization

A critical section is a section of code that uses mutual exclusion to ensure that:

- Only one thread at a time can execute in the critical section
- All other threads have to wait on entry
- When a thread leaves a critical section, another can enter

A race condition happens when **two concurrent threads (or processes) accessed a shared resource without any synchronization**. Race conditions arise in software when an application depends on the sequence or timing of processes or threads for it to operate properly.

A race condition can also happen when the **result of the program depends on the sequence of which the threads access the critical section**. These inconsistencies of the result occur in critical sections due to sharing of resources by multiple processes/threads. (eg. sharing arrays, variables, files, etc.)

Process Synchronization with Semaphores

Download and study semaph.c program which illustrates the usage of semaphores for synchronizing Linux processes. To manage inter process communication, we need to create a shared memory space. This shared memory need to be destroyed at the completion of the program. Observe the use of Semaphore related function calls in the code. You may refer to man pages for each function call to learn more information.

- Compile the code in a terminal (console):
  ```
  > gcc -pthread -o semaph semaph.c
  ```
- Run the program in a terminal (console):
  ```
  > ./semaph
  ```

Thread Synchronization with Mutexes and Condition Variables

Download and study `race-condition.c` program which illustrates a multithreaded program with a race condition. `race-condition.c` demonstrates manipulation of a shared `global_counter` by multiple threads. There are 4 ADD threads that increment the global_counter by 1 each, and 4 SUB threads that decrement the global_counter by 1 each. At the end, the program should print the final value of the global_counter. Please note that `sleep(rand() % 2)` is called inside add and sub functions to delay the completion for few seconds.

- Compile the code in a terminal (console)

  ```
  > gcc -pthread -o race race-condition.c
  ```

- Run the program in a terminal (console)

  ```
  > ./race
  ```

Exercise 3
Compile and run the race-condition.c program. Observe the output.

You should notice that the final result of the global counter is printed before completion of all threads. This is due to non-completion of threads before printing the final result.

`pthread_join` is a pthread library function which guarantees the caller thread that the target thread is terminated. For example, in race-condition.c program, if the main thread calls pthread_join for all the ADD threads and SUB before printing the final result of the global variable, we should see the real final value after all SUB and ADD threads are completed.

```
int pthread_join(pthread_t thread, void **retval);

// example
pthread_join(thread, NULL);
```

Exercise 4
Modify `race-condition.c` (new name `ex4-race-condition.c`) to ensure that all ADD threads and SUB threads complete before printing the final result. Compile, run, and observe the output. (run multiple times to see if the output is consistent)

Although the threads are synchronized, you may still see a wrong final result. Each ADD thread increments the counter by 1 and each SUB thread decrements the counter by 1. Thus, the final value of the `global_counter` should remain at its initial value of 10. This behavior is caused due to the existence of a **race condition**.

Mutexes

Mutex is a synchronization construct which is used to control access to a critical section in the code. A mutex variable acts like a lock and the thread that acquires the thread gets to access the critical section. Once a thread has acquired a mutex lock to a critical section, no other thread can acquire it until the first thread releases the mutex.

pthread mutex example

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&lock);

// critical section here

pthread_mutex_unlock(&lock);
```

Exercise 5

Modify `race-condition.c` (new name: `ex5-race-condition.c`) by adding a mutex variable to control access to the global_counter. Compile, run, and observe the output. (run multiple times to see if the output is consistent)

Condition variables

Mutexes provide mechanism for controlling access to a critical section and prevent races. However, they do cannot be used for threads to wait until another thread completes some arbitrary task. Condition variables provide a mechanism for threads to be signaled by other threads rather than continuously polling to check if a certain condition has been met. Condition variables are used in association with mutex variables.
Related pthreads functions:

- Create and destroy

  `pthread_cond_init(condition,attr)`

  `pthread_cond_destroy(condition)`

- Waiting and signaling:

  `pthread_cond_wait(condition,mutex)`

  `pthread_cond_signal(condition)`

  `pthread_cond_broadcast(condition)`

Download and study `cond.c` which demonstrates use of condition variables. The main thread creates three threads. Two of those threads increment a “count” variable, while the third thread watches the value of “count”. When “count” reaches a predefined limit, the waiting thread is signaled by one of the incrementing threads. The waiting thread “awakens” and then modifies count. The program continues until the incrementing threads reach TCOUNT. The main program prints the final value of count.

Exercise 6

Modify `race-condition.c` (new name: ex6-race-condition.c) using condition variables to prevent SUB threads from executing until all ADD threads are completed.

Further reading and examples: `https://computing.llnl.gov/tutorials/pthreads/`

Part 3: Producer-Consumer Problem

In this part we solve the producer consumer problem using (i) processes and semaphores, and, (ii) threads, mutexes and condition variables.

Recall the producer consumer problem from the lecture. Let us limit the scope of the problem as follows: There are two producers and one consumer. Each producer generates a random number between 1 and 10 (inclusive) and inserts (writes) to `producer_buffer`. Consumer reads (consumes) these numbers one at a time and updates the sum of all numbers consumed in `consumer_sum` variable. The `producer_buffer` can store only 10 numbers and the producers are not allowed to overwrite any unconsumed number.

Exercise 7

`prod-con-threads.c` is a basic code skeleton for producer consumer problem without synchronization constructs. Implement the above-mentioned producer-consumer scenario with synchronization using pthreads, mutexes and/or condition variables starting from the code skeleton (`prod-con-threads.c`). You may add any number of synchronization constructs and other functions as required. Name your new program `ex7-prod-con-threads.c`

Implementing the same producer consumer logic with processes involves allocating memory from the kernel space as a means of maintaining a global variable (for inter-process communication). Refer to the example which uses shared memory with processes in `semaph.c`.

Exercise 8

Implement the above-mentioned producer-consumer scenario with synchronization using processes and semaphores. Name your new program `ex8-prod-con-processes.c` The very basic approach should be as follows:

```
// allocate shared memory
// allocate semaphores
if (fork()) producer(); // producer 1
if (fork()) producer(); // producer 2
consumer();
// cleanup shared memory
```

Exercise 9

Limit the number of total items (numbers) produced by the consumers to 100 and measure the time taken to complete the program for both cases (processes and pthreads). Comment in a paragraph on the observations (maximum a paragraph).

Lab sheet (2% of your final grade):

You are required to submit your solutions for exercise 7, 8, and 9. Submit before Friday, 28 August, 2pm via LumiNUS an archive named using your students number (for example, A0123456X.zip) containing:

- ex7-prod-con-threads.c
- ex8-prod-con-processes.c
- README file explaining how to compile your code and one paragraph to answer exercise 9. Don't forget to add your name and student number in the files.

Attempting the lab and having a working solution is more important than having a perfect solution. Full marks can be obtained for submissions that have minor bugs.

Appendix: Debugging

Viewing Processes and Threads

To view the running processes and threads in a linux console, we can use ps and top commands. These commands should be invoked separately in a different terminal window.

To see a list of processes running on your system details, run any of the following commands in a terminal:

- `> ps -ef`
- `> ps -A`
- `> top`

If too many information is printed and impossible to read at one time, you can pipe the output through the less command to scroll through them at your own pace:

```
> ps -A | less
```

More information on `ps`: `http://man7.org/linux/man-pages/man1/ps.1.html` or type in `man ps` in the console.

To list individual threads under each process:

```
> top -H
```

More information on `top`: `http://man7.org/linux/man-pages/man1/top.1.html` or type in `man top` in the console.

You may also try `htop`, an improved version of top which supports advanced visualization features.

To kill a running process use either one of these commands:

- `> kill -p <pid>`
- `> pkill`
- `> killall`

Debugging C Programs

There are multiple debugging tools available for debugging C programs. The gdb debugger is a command line debugger for C (and many other languages). To use gdb debugger, we need to compile the source code with -g compiler flag. (When you compile with -g the compiler includes debugging information in the binary, making it easier for gdb to find bugs.) gdb provides debugging features such as adding breakpoints, step execution, and, examining the call stack.

- Compiling the code in a terminal (console)

  ```
  > gcc -g -o prog prog.c
  ```

- Invoke gdb

  ```
  > gdb prog
  ```

- Run the program inside gdb

  ```
  > run <prog argument1> <prog argument 2>
  ```

Resources on gdb

- Gdb tutorial from UChicago

  https://www.classes.cs.uchicago.edu/archive/2017/winter/51081-1/LabFAQ/lab2/gdb.html

- Official gdb documentation

  https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_toc.html

Valgrind is a more advanced profiler which helps us debug applications as well as detect performance issues. It includes advanced features such as detecting race conditions and false sharing.

You may read more on Valgrind at: http://valgrind.org/docs/manual/manual.html