Lab 1
Processes, Threads and Synchronization Basics

CS3210 – 2024/25 Semester 1

Learning Outcomes

1. Understand the differences between processes and threads
2. Use the POSIX thread (`pthread`) library for shared-memory parallel programming
3. Implement critical sections in the code
4. Apply basic synchronization constructs in programs
5. Start to become familiar with our lab machines

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

Why Learn fork(), pthreads, etc?

`fork()` / pthreads are relatively lower-level ways to create and synchronize processes and threads. However, it's important to understand these intricacies before we explore the more abstracted and powerful libraries such as OpenMP / MPI.

Programming Language: C vs C++

We will be using the **C++** programming language for all labs, tutorials, and it is the recommended language for assignments. You may have learned some **C** from prior courses, but we find that students (even those that only know C) ultimately prefer C++ during assignments due to its expressive power. However, if necessary, we will still accept assignments in C if you strongly prefer it.

Please note that we don't expect you to become proficient with C++, and this module treats it more like "C with benefits". We will often not use idiomatic C++ for simplicity.

If you know C++, please do not use C++'s own `std::thread`, condvar/semaphore/mutex/unique_lock, etc. in CS3210 unless specifically allowed. Please use pthreads.

Logging in & Getting Started

For the lab and assignments, you are going to be running your code on the machines in the Parallel and Distributed Computing Lab located in COM1-B1-02. **Use the following instructions to connect to the lab machines remotely over** `ssh`.

Please follow `https://nus-cs3210.github.io/student-guide/accessing/`.

For this lab, **connect to one of the machines using the guide above, and start working on completing the tasks in the lab**. The lab files can be found here: `https://www.comp.nus.edu.sg/~srirams/cs3210/L1_code.zip`. You can use the command "`wget`" to download the code to the lab machine, and "`unzip`" to unzip the file.

Part 1: Processes vs. Threads

Multi-process programming on Linux with C++

Let us look at a simple program which demonstrates the use of processes in Linux. Open the `ex1-processes.cpp` file and study the use of the `fork()` system call and its return values.

Note the `wait(nullptr)` 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):

  `> g++ -o processes ex1-processes.cpp`

- Run the program in a terminal:

  `> ./processes`

Exercise 1

Compile and run `ex1-processes.cpp`. 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

```
for(size_t i = 0; i < NUM_THREADS; i++)
{
  printf("main thread: creating thread %zu\n", i);

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

Listing 1: Snippet of `ex2-threads.cpp`

`ex2-threads.cpp` contains a simple example on creating (spawning) threads with the `pthread` library and terminating them. In `ex2-threads.cpp`, the loop runs `NUM_THREADS` number of times and calls the `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 `man` (manual) command in the terminal (console):

  `> man pthread_create`

- Compile the code in a terminal:

  `> g++ -pthread -o threads ex2-threads.cpp`

- Run the program in a terminal:

  `> ./threads`

Exercise 2

Compile `ex2-threads.cpp` and run the 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) access 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 correctly.

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, e.g. sharing arrays, variables, files, etc.

Process Synchronization with Semaphores

Download and study the program `semaph_named.cpp` 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 needs to be destroyed at the completion of the program. Observe the use of semaphore-related function calls in the code. You may refer to the `man` pages for each function call to learn more information.

- Compile the code in a terminal:

  > g++ -pthread -o semaph semaph_named.cpp

- Run the program in a terminl:

  > ./semaph

Pitfalls: Named vs Unnamed Semaphores
Notice that we did not explicitly share our semaphore (`sem`) between parent and child processes. `sem` is shared correctly across all our processes because we used *named semaphores* through the POSIX `sem_open` library call. This automatically causes `sem` to be in a shared memory region. If we used *unnamed semaphores* through the POSIX `sem_init` library call, we would have to allocate the semaphore within shared memory ourselves. See `man sem_overview`.
Read `semaph_shm.cpp` to see the changes required for unnamed semaphores.

Thread Synchronization with Mutexes and Condition Variables

Download and study the program `ex3456-race-condition.cpp` which illustrates a multi-threaded program with a race condition. It 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 it by 1 each. At the end, the program should print the final value of `global_counter`. Please note that `sleep(rand() % 2)` is called from within the add and sub functions to delay completion for few seconds.

- Compile the code in a terminal:

  > g++ -pthread -o race ex3456-race-condition.cpp

- Run the program in a terminal:

  > ./race

Exercise 3
Compile `ex3456-race-condition.cpp` and run the 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. In the program `ex3456-race-condition.cpp`, if the main thread calls `pthread_join` for all the ADD and SUB threads before printing the final result of the global variable, we should see the real final value after all ADD and SUB threads are completed.

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

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

Exercise 4

Modify `ex3456-race-condition.cpp` (new name `ex4-race-condition.cpp`) 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)

Since each ADD thread increments the counter by 1 and each SUB thread decrements the counter by 1, the final value of the `global_counter` should remain at its initial value of 10, However, you may still see the wrong final value despite joining on all the threads before printing the result, as the threads are not synchronized. This behavior is caused due to the existence of a **race condition**.

Mutexes

A 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 `ex3456-race-condition.cpp` (new name: `ex5-race-condition.cpp`) by adding a mutex variable to control access to the `global_counter`. Compile, run, and observe the output. (Run multiple times to observe if the output is consistent!)

What do you think are the differences between a *pthread mutex* and a *binary semaphore*?

Condition variables

Mutexes provide a mechanism for controlling access to a critical section to prevent races. However, they 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 `pthread` functions are:

- 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 `ex6-cond-example.cpp` which demonstrates the 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 `ex3456-race-condition.cpp` (new name: ex6-race-condition.cpp) 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 (to be submitted)

In this part, we combine the first two parts to solve the producer-consumer problem using both (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 the `producer_buffer` variable (an array of integers). The consumer reads (consumes) these numbers one at a time and updates the sum of all numbers consumed into a `consumer_sum` variable. At the end of the program, **print out the `consumer_sum` value for verification**. The `producer_buffer` can store only 10 numbers and the producers are not allowed to overwrite any unconsumed number. The numbers should be consumed in the order they are produced, that is, **FIFO** order. You may also **limit the total number of items produced/consumed** to avoid catching Control-C (SIGINT) signals from the user.

Exercise 7

`ex789-prod-con-threads.cpp` is a basic skeleton for the producer-consumer problem without any synchronization constructs. Implement the above-mentioned producer-consumer scenario (two producers and one consumer) with synchronization, **efficiently, using pthreads, mutexes and condition variables** by modifying the skeleton code in `ex789-prod-con-threads.cpp`. You may add any number of synchronization constructs and other functions as required. Make sure that producers and consumers are allowed to process the items simultaneously (i.e. **not with the use of a single critical section**), but they are never accessing the `producer_buffer` at the same time. The producers and consumers process multiple items (until stopped). The challenge is to synchronize them to avoid buffer overflow. Name your new program `ex7-prod-con-threads.cpp`

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_named.cpp`.

Exercise 8

Implement the exercise above **but using processes and semaphores only** (i.e.,, no pthreads, condition variables, etc). Name your program `ex8-prod-con-processes.cpp`. The very basic approach of your program should be as follows:

```
// allocate shared memory
// allocate semaphores
if (fork() == 0) producer(); // producer 1
if (fork() == 0) producer(); // producer 2
consumer();
// cleanup shared memory
```

The producers and consumers process multiple items (until stopped). You might notice that some processes are left running if you forcefully (Ctrl+C) stop the execution. You may use `killall -9 <name_of_executable>` in terminal to clean up.

Exercise 9

Limit the total number of items produced/consumed to a **sufficiently-large fixed value** (to observe the performance of the programs accurately) and measure the time taken to complete the program for both cases (processes and pthreads). Then, vary this limit on the total number of items produced. Comment on the observations for your threads and processes implementations in exercises 7 and 8 **(maximum length: 1 paragraph)**.

Pitfalls: Correctly exiting multi-threaded / multi-process programs

It can be challenging to correctly detect a user-initiated termination of your program (a Control-C key combination which sends SIGINT to your processes) and clean up all of your program's resources. You do not need catch signals for any of the exercises in this submission (i.e., just produce and consume a fixed amount). If you are interested, you may want to consider these things:

- The `signal` function (`man 2 signal`), and what code can run safely in a signal handler.
- The `pthread_sigmask` function (`man 3 pthread_sigmask`).
- How to indicate to running processes that they should exit.
- How to ensure processes do not deadlock when trying to exit.

Lab sheet (2% of your final grade):

You are required to submit your solutions for **exercise 7, 8, and 9.** Submit before Monday, 2nd Sept, 2pm via Canvas a PDF report named using your student number (for example, A0123456X.pdf) containing:

- Your code for the `producer` and `consumer` functions in `ex7-prod-con-threads.cpp` and `ex8-prod-con-processes.cpp`.
- Your answer for exercise 9.

Please use a legible monospace font (e.g. 11-point Consolas) with single line spacing for your code. Your answer for exercise 9 should also be in a legible font (no smaller than 11-point Arial).

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/htop 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
- > htop

If too much 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

If you are looking for a specific process, e.g., bash, you can do
> ps -A | grep bash

> 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 (table of processes) which supports advanced visualization features.

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

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

Debugging C / 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 the 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 breakpoints, step execution, and, examining the call stack.

- Compiling the code in a terminal (console)

```
>g++ -g -o prog prog.cpp
```

- 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