Problem Set 3: Synchronization and Scheduling1

- See course webpage for due date and time.
- Submit deliverables to CourSys: https://courses.cs.sfu.ca/
- Late penalty is 10% per calendar day (each 0 to 24 hour period past due).
  - Maximum 2 days late (20%)
- Do not show another student your code, do not copy work found online, and do not post questions about the assignment online. Please direct all questions to the instructor or TA: cmpt-300-d2-help@sfu.ca;
  You may use general ideas you find online and from others, but your solution must be your own.
- See the marking guide for details on how each part will be marked.

When asked to explain something, don't copy and paste text found online or in the man pages. Explain the idea in your own words.

Short Answer Questions

- Write brief answers to the following questions in a text file name `short_answer.txt` or write your answers in another program and generate a PDF named short_answers.pdf.

1. Describe the circumstances under which one should use each of the following synchronization tools. You may want to consider how long a lock will be held, if it is a multiprocessor system, and any other factors which seem relevant.
   a. spinlocks
   b. mutex locks
   c. semaphores
   d. condition variables
   e. reader-writer locks
2. On a single processor system, answer each of the following questions about disabling interrupts:
   a. Describe how disabling interrupts can be used to ensure mutual exclusion.
   b. Discuss the problems of allowing programs to disable interrupts in user mode without executing a sys-call.
3. Consider the code shown below which can be used for allocating resource numbers. For example, it could be use in the kernel to allocate process numbers.

```
#define MAX_RESOURCES 5

int resources_allocated = 0;

// Allocate a resource to the calling code.
// Return true if successful, false if unable to allocate.
_Bool allocate_resource(void)
{
```

```
10.         if (resources_allocated >= MAX_RESOURCES) {
11.             return false;
12.         } else {
13.             resources_allocated++;
14.             return true;
15.         }
16. }
17.
18. // Release the resource
19. void release_resource(void)
20. {
21.         resources_allocated--;
22. }
```

- Identify the race condition(s) by describing which piece(s) of code, when running concurrently, can trigger the race condition.
- Assume you have a mutex lock named `mutex` with operations acquire() and release(). Indicate where you would need to lock and unlock the mutex. In your answer, you may indicate "between lines 3 and 4" by saying "at line 3.5 add ..."
- Atomic variables are special variable types that ensure that increment, decrement and addition operations are performed atomically. Could the variable at line 2

  ```
  int resources_allocated = 0;
  ```

  be replaced with an atomic integer variable

  ```
  atomic_t resources_allocated = 0;
  ```

  to correctly prevent all race conditions in the code?

1. Consider the following set of processes, with the length of the CPU burst given in miliseconds:

Process	Burst Time	Priority
P1	2	2
P2	1	1
P3	8	4
P4	4	2
P5	5	3

1. The process are assumed to have arrived in the order P1 through P5 all at time 0.
   a. Draw four GANTT charts that illustrate the execution of these processes using the following scheduling algorithms:
      i. FCFS
      ii. SJF
      iii. Nonpreemptive priority (a larger priority number implies a higher priority)
      iv. RR (quantum = 2)
   b. What is the turnaround time for each process for each of the scheduling algorithms in part a?
   c. What is the waiting time for each process for each of the scheduling

algorithms?

d. Which of the algorithms results in the minimum average waiting time (over all processes)?

2. Consider a system running ten I/O bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for each millisecond of CPU computing (1ms CPU burst), that each I/O operation takes 10 milliseconds to complete, and all I/O operations are blocking (release CPU) and independent (for example, using different devices). Assume the CPU task does no I/O and uses all CPU time it is given. Also assume that the context-switching overhead is 0.1 milliseconds and that all process are long-running tasks. Describe the CPU utilization for a RR scheduler when:

   a. The time quantum is 1 milliseconds.

   b. The time quantum is 10 milliseconds.

3. For each of the following scheduling algorithms, explain how it favours short processes (if at all):

   a. FCFS

   b. RR

   c. Multilevel feedback queues

Coding Questions

- Write a multithreaded Linux C application in a file named `swapper.c` which will:
  - Declare three constants:
    - DATA_SIZE: The size of the data array. Set to 100.
    - NUM_THREADS: The number of swapper threads to create. Set to 40.
    - NUM_SWAPS: The number of swaps per threads. Set to 2,000,000.
  - Create an `int` array of size DATA_SIZE and initialize its elements to 0 through DATA_SIZE-`1`.
    - Hint: make this array a global because almost functions need access to it and it is shared.
  - Spawn NUM_THREADS swapper threads. Each swapper thread loops NUM_SWAPS times, each time it randomly picks two elements of the data array and swap their content. (May randomly pick the same index for each element, that's OK to swap and element with itself).
    - Hint: Make an array of type `pthread_t` of size `NUM_THREADS` to store all the thread IDs.
  - Spawn a single data checking thread which constantly loops through the following actions:
    - Sleep for 1 second (use `sleep()` function).
    - Prints a "`*\n`"

    - Count the number of expected values that are missing from the data array. (i.e., for each value between 0 and DATA_SIZE-1, check if that value is in the data array.) If the count is >0 print "`Count Missing = __`" (fill in blank).
- Once all swapper threads have exited, cancel the data-checking thread.
    - Hint: You can know all threads are done by running `pthread_join()` in the main thread and joining on each swapper thread's thread ID. This will not only wait for each thread to exit, but also cleanup memory allocated to the thread.
    - Hint: Use the code below to cancel the data checking thread. Canceling a thread causes it to exit. Once canceled, you must still join to the thread.

      ```
      pthread_cancel(thread_id);
      pthread_join(thread_id, NULL);
      ```

        - See man pthread_cancel for more.
- **Make your code thread-safe so there are no race conditions.**
    - Once all race cases are solved, your application should print out no error messages.
    - Hint: Initially it's OK for it to be not thread safe. This will help test out your checking thread's code. Then make it thread safe. You may need to identify multiple critical sections.
- Your code will likely be executed to prove it works. Always take pride in your code; make it work and be well structured.

Deliverables

Submit the following deliverabels in a ZIP or tar.gz file to CourSys:

1. `short_answer.txt` (or .pdf)
2. swapper.c

- You may include a Makefile if you like. Please remember that all submissions will automatically be compared for unexplainable similarities.