CS2106 Operating Systems

Tutorial 4 **Suggested Solutions**
More on Process Scheduling and IPC

1. [Adapted from AY1920S1 Midterm – Evaluating scheduling algorithms] **(Note: This question is similar but not identical to the last question of T03)** Briefly answer each of the following questions regarding process scheduling, stating your assumptions, if any.

   a. Under what conditions does FCFS (FIFO) scheduling result in the shortest possible average response time?

   b. Under what conditions does round-robin (RR) scheduling behave identically to FIFO?

   c. Under what conditions does RR scheduling perform poorly compared to FIFO?

   d. Does reducing the time quantum for RR scheduling help or hurt its performance relative to FIFO, and why?

   e. Do you think a CPU-bound (CPU-intensive) process should be given a higher priority for I/O than an I/O-bound process? Why?

   ANS:

   a. FIFO minimises the average response time if the jobs arrive in the ready queue in order of increasing job lengths. This avoids short jobs arriving later from waiting substantially for an earlier longer job. A special case also exists: when all jobs have the same completion time.

   b. RR behaves identically to FIFO if the job lengths are shorter than the time quantum, since it is essentially a pre-emptive variant of FIFO.

   c. There are multiple accepted responses, depending on the criteria of evaluation for RR scheduling. Some possible responses include:
      - When the job lengths are all the same and much greater than the time quantum, RR performs poorly in average turnaround time
      - When there are many jobs and the job lengths exceed the time quantum, RR results in reduced throughput due to greater overhead from the OS incurred due to context-switches when jobs are pre-empted

   d. Both *yes* and *no* responses are accepted, if the justification provided references the criteria by which RR is being evaluated by.

- *Yes*: reducing the time quantum increases the amount of CPU time spent by the OS performing context-switching, reducing throughput and increasing the average turnaround time
- *No*: reducing the time quantum reduces the waiting time for a task to first receive CPU time, improving the intial response time (*responsiveness*)

e. Two main lines of reasoning were accepted.
- *Yes*: CPU processes can be given higher priority for I/O so they may return to waiting for the CPU, decreasing overall turnaround time at the expense of response time of I/O-bound processes
- *No*: maximise responsiveness of I/O-bound processes

2. [Shared Memory] In this question, we are going to analyze the pitfalls of having multiple processes accessing and modifying data at the same time. Compile and run the code in `shm.c`. The executable accepts two command line arguments, **n** and **nChild**; if no command line arguments are given, the default values will be used: **n** is initialized to 100 and **nChild** is initialized to 1.

The code does a simple job: the parent creates **nChild** processes and then each process, the parent included, increases a shared memory location by 1 for **n** number of times. After all processes have finished incrementing the shared value, the parent prints the shared result and exits.

a. The value of the shared memory is initialized to 0, what is the expected final printed value given `n` and `nChild`?
b. Run the program multiple times with various values for `n` and `nChild`. Observe the result when n is high (e.g. `n` = 10000 and `nChild` = 10). Explain the results.
c. Do you think (b) is caused by multi-core processors since processes can run in parallel? Would the issu n e persists if you use a single core processor? [You can run experiment to find out. Use the Linux command `taskset` to bind a process (and all its child) to a single processor core, e.g. "`taskset -c 5 ./a.out 10000 100`" will bind the executable to run on core no. 5.]

ANS:

a. We expect the printed value to be equal to `(1 + nChild) * n` at the end.
b. We see final value smaller than expected in some test cases when n is sufficiently high. When incrementing a variable,the process must perform the following steps:
    I.   Read the value from the memory
    II. Modify it, and
    III. Write the value back to the memory.

3. [Protecting the Shared Memory] Allowing processes to access and modify shared data whenever they please can be problematic! Therefore, we would like to modify the code in **shm.c** such that the output is deterministic regardless of how large **n** and **nChild** are. More precisely, we want the processes to take turns when modifying the result value that resides in the shared memory.

To this end, we will add another field in the shared memory, called the **order** value, that specifies which process' turn it is to increment the shared result. If the **order** value is 0, then the parent should increase the shared result, if the **order** value is 1, then the first child should increase it, if the **order** value is 2, then the second child and so on. Each process has an associated **pOrder** and checks whether the value **order** is equal to its **pOrder**; if it is, then it proceeds to increment the shared result. Otherwise, it waits until the order value is equal to its **pOrder**.

Your task is to modify the code in **shm_protected.c** to achieve the desired result. You will have to:
- Create a shared memory region with two locations, one for the shared result, and the other one for the order value;
- Write the logic that allows a process to modify the shared result only if the order value is equal to its **pOrder** (the skeleton already takes care of assigning the right **pOrder** to each process);
- Print the result value and cleanup the shared memory.

Why is **shm** faster than **shm_protected**? Why is running **shm_protected** with large values for nChild particularly slow? (Hint: You may want to take a look at the output of the Linux command *htop*)

ANS:

To guarantee that two processes won't try to modify the shared memory location at the same time, we must **serialize the access** to it, i.e., make it impossible for two processes to access it at the same time. Serializing the code will, of course, make the execution time longer.

The technique we use to prevent simultaneous access is called **busy waiting**. Each process is continuously checking whether its time to modify the variable has come. This requires the process to run on the CPU and consume CPU cycles (thus the name). Note that at the moment when it's Child X's time to increment the value, there are nChild – X processes doing busy waiting – each of these processes will get scheduled to run on the CPU but the execution of the program will not make any progress. Because of this, your program will take a long time to complete for large values of nChild.

Interesting topics for your own exploration

Unix Pipes (https://en.wikipedia.org/wiki/Dup_(system_call) – pretty good article with code(!) example).

Linux Message Queue (https://www.geeksforgeeks.org/ipc-using-message-queues/ - Simple code example to highlight the communication).

Shared Memory in Parallel Programming (https://en.wikipedia.org/wiki/Shared_memory - Generalized shared memory model that can work *across network*)