CS2106 Operating Systems
Semester 2 2021/2022

Week of 14th February 2022
Tutorial 4
More on Process Scheduling and IPC

1. [MLFQ] As discussed in the lecture, the simple MLFQ has a few shortcomings. Describe the scheduling behavior for the following two cases.

   a. (Change of heart) A process with a lengthy CPU-intensive phase followed by I/O intensive phase.

   b. (Gaming the system) A process repeatedly gives up CPU just before the time quantum lapses.

   The following are two simple tweaks. For each of the rules, identify which case (a or b above) it is designed to solve, then briefly describe the new scheduling behavior.

   i. (Rule – Accounting matters) The CPU usage of a process is now accumulated across time quanta. Once the CPU usage exceeds a single time quantum, the priority of the task will be decremented.

   ii. (Rule – Timely boost) All processes in the system will be moved to the highest priority level periodically.

2. [Adapted from AY1920S1 Midterm – Evaluating scheduling algorithms] 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?

3. [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 issue 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.]

4. [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*)

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*)