CS2106 Operating Systems
Semester 1 2021/2022
Tutorial 3
Process Scheduling

1. (Walking through Scheduling Algorithms) Consider the following execution scenario:

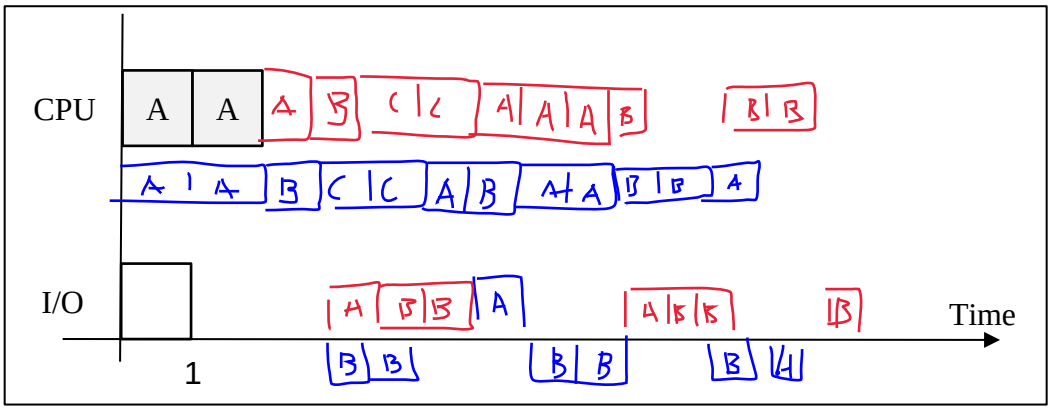
Program A, Arrives at time 0
Behavior (C**X** = Compute for **X** Time Units, IO**X** = I/O for **X** Time Units): C**3**, IO**1**, C**3**, IO**1**

Program B, Arrives at time 0
Behavior: C**1**, IO**2**, C**1**, IO**2**, C**2**, IO**1**

Program C, Arrives at time 3
Behavior: C**2**

a) Show the scheduling time chart with First-Come-First-Serve algorithm. For simplicity, we assume all tasks block on the same I/O resource.

Below is a sample sketch up to time 1:



b) What are the turnaround time and the waiting time for program A, B and C? In this case, waiting time includes all time units where the program is ready for execution but could not get the CPU.

c) Use **Round Robin** algorithm to schedule the same set of tasks. Assume time quantum of **2 time units**. Draw out the scheduling time chart. State any assumptions you may have.

d) What is the response time for tasks A, B and C? In this case, we define response time as the time difference between the arrival time and the first time when the task receives CPU time.

2. (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.

3. (Adapted from Midterm 1516/S1 – Understanding of Scheduler)

   a) Give the **pseudocode** for the **RR scheduler function.** For simplicity, you can assume that all tasks are CPU intensive that runs forever (i.e. there is no need to consider the cases where the task blocks / give up CPU). Note that this function is invoked by timer interrupt that triggers once every time unit.

   Please use the following variables and function in your pseudocode.

Variable / Data type declarations
Process **PCB** contains: **{ PID, TQLeft, … }** // TQ = Time Quantum, other PCB info irrelevant. **RunningTask** is the PCB of the current task running on the CPU. **TempTask** is an empty PCB, provided to facilitate context switching. **ReadyQ** is a FIFO queue of PCBs which supports standard operations like **isEmpty()**, **enqueue()** and **dequeue()**. **TimeQuatum** is the predefined time quantum given to a running task.
"Pseudo" Function declarations
`SwitchContext( PCBout, PCBin );`

Need to take note of the edge case where the current running task is the only task left, so no need to reduce priority

Save the context of the running task in ***PCBout***, then setup the new running environment with the PCB of ***PCBin***, i.e. vacating ***PCBout*** and preparing for ***PCBin*** to run on the CPU.

b) Suppose when a process gets blocked waiting for I/O and the process scheduler runs only at the next timer interrupt and chooses another process to run. Discuss any shortcomings that might arise? How would you solve those?

4. (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? Justify your answers.

Recall the identity of each method

Trend is trade-off between turnaround time and responsiveness

Questions for your own exploration

5. (Putting it together) Take a look at the given mysterious program **`Behavior.c`**. This program takes in one integer command line argument **D**, which is used as a **delay** to control the amount of computation work done in the program. For the part (a) and (b), use ideas you have learned from **Lecture 3: Process Scheduling** to explain the program behavior.

   Use the command `taskset --cpu-list 0 ./Behaviors D`
   This restricts the process to run on only one core.
   Warning: you may not have the `taskset` command on your Linux system. If so, install the `util-linux` package using your package manager (apt, yum, etc).

   Note: If you are using Windows Subsystem for Linux (WSL), make sure that you are using WSL2 kernel instead of WSL1. You can check by running `wsl -l -v` and upgrade using `wsl --set-version <distro-name> 2`

   a. **D** = 1.
   b. **D** = 100,000,000 (note: don't type in the "," ☺)

   c. Now, find the **smallest D** that gives you the following interleaving output pattern:

Interleaving Output Pattern
[6183]: Step 0
[6184]: Step 0
[6183]: Step 1
[6184]: Step 1
[6183]: Step 2
[6184]: Step 2
[6183]: Step 3
[6184]: Step 3
[6183]: Step 4
[6184]: Step 4
[6184] Child Done!
[6183] Parent Done!

   What do you think "**D**" represents?

   *Note: "**D**" is machine dependent, you may get very different value from your friends'.*

6. (Predicting CPU time) In the lecture, the ***exponential average*** technique is briefly discussed as a way to estimate the CPU time usage for a process. Let us try to see this technique in action. Use **Predicted_0 = 10 TUs** and **α = 0.5**. Predicted_0 is the estimate used when a process is first admitted. All subsequent predictions use the formula:

$$\textbf{Predict}_{N+1} = \alpha\textbf{Actual}_N + (1-\alpha)\textbf{Predict}_N$$

Calculate the error percentage (**Abs(Actual – Predict) / Actual * 100%**) to gauge the effectiveness of this simple technique. CPU time usage of two processes are given below, fill in the table as described and explain the differences in error percentage observed.

Process A			
Sequence	**Predicted**	**Actual**	**Percentage Error**
1	**10**	9	11.1%
2		8	
3		8	
4		7	
5		6	
		Average Error:	

Process B			
Sequence	**Predicted**	**Actual**	**Percentage Error**
1	**10**	8	25%
2		14	
3		3	
4		18	
5		2	
		Average Error:	

7. (Scheduler Case Study - Linux) Let us look at the Linux scheduler, which is at the heart of one of the most widely used server OS (100% of the 2019 top 500 supercomputers in the world use Linux!) Instead of a full coverage, we will pick and choose several aspects to discuss, depending on the available time in your tutorial.

   Linux scheduler (in kernel 2.6.x) can be understood as a MLFQ variant. There are 140 priority levels (0 = highest, 139 = lowest) split into two ranges (real time task has priority 0 to 99 and time sharing task has priority 100 to 139). For our purpose, we will consider only time sharing tasks (i.e. normal user processes).

a) In older Linux kernel, the scheduler maintains a single linked list to keep track of all runnable tasks. When picking a task, this list is iterated through to find the task with the

highest priority. In kernel 2.6.x, an array of 140 linked lists (i.e. each priority level has a linked list) is maintained instead. Assuming everything else remains unchanged, what is the benefit of this change?

b) In the scheduler, there are two sets of tasks:

- "**Active tasks**": Tasks ready to run.
- "**Expired tasks**": Tasks which have exhausted their time quantum but runnable (i.e. they are not blocked).

Based on the priority level, each task on the "Active" set will eventually get a time quantum to run. If the task gives up early or exhausted its time quantum, it will be placed on the "Expired" set. When the "Active" set is empty, the scheduler will then swap the two sets, i.e. the "Expired" set is now the new "Active" set. What do you think is the benefit of this design?

c) The time quantum (known as ***time slice*** in Linux terminology) is not a constant value, instead it is proportional to the priority level (i.e. priority level 100 has the shortest time slices while 139 has the longest). What do you think is the rationale?

d) The scheduler applies penalty (up to +5) or bonus (up to -5) to the task's priority level depending on the execution behavior. So, a task at priority level 110 can be placed at level 105 (received bonus) or level 115 (penalized) between scheduling. This adjustment is based on a value *sleep_avg* kept with the task. This value:

- Increased by the amount of time the process is sleeping (i.e. blocked).
- Decreased by the amount of time the process actively runs.

A high sleep value corresponds to a large bonus while a low sleep value would cause a large penalty. What is the rationale of this mechanism?

Disclaimer: To fit a huge case study in a "short" tutorial question requires heavy simplifications. So, please do not take this question as the complete algorithm description.