CS2106 Operating Systems
Semester 1 2021/2022

Tutorial 3 Suggested Solutions
Process Scheduling

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

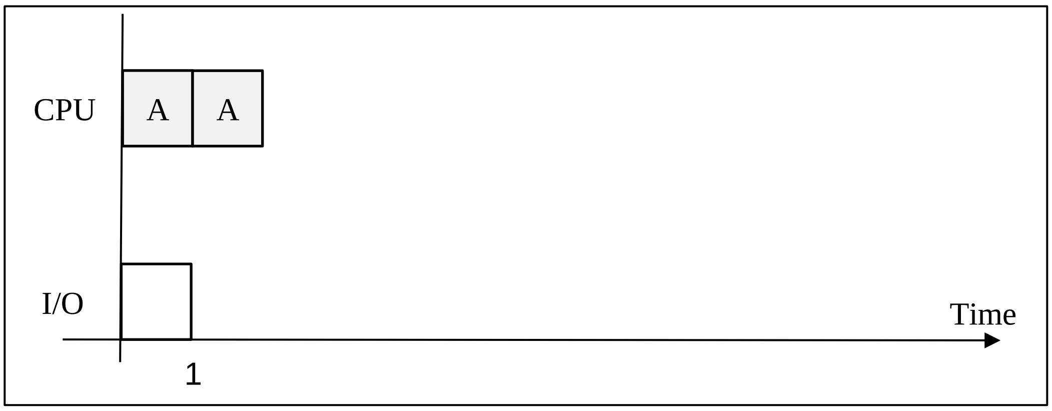
Program A, Arrives at time 0
Behavior (C**X** = Compute for **X** Time Units, IO**X** = I/O for **X** Time Units): C**3**, IO**1**, C**3**, IO**1**

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:



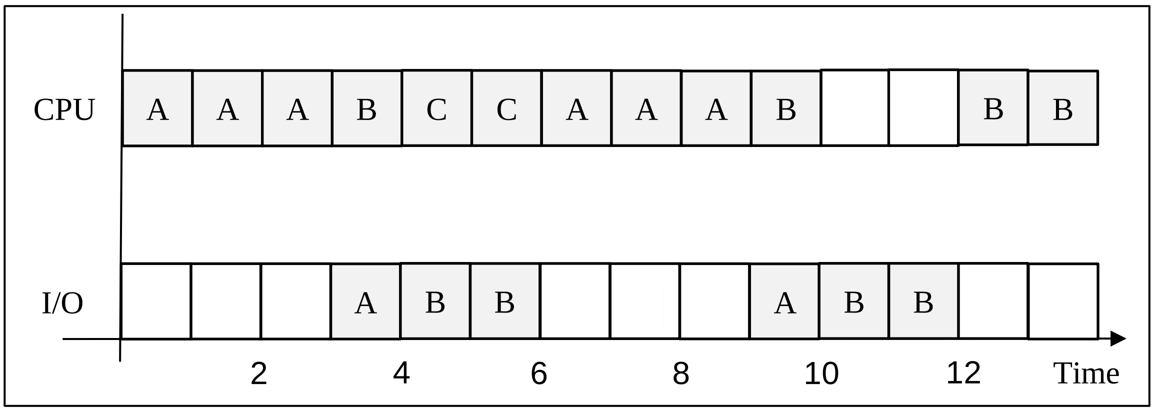
b) What are the turnaround time and the waiting time for program A, B and C? In this case, waiting time includes all time units where the program is ready for execution but could not get the CPU.

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.

ANS:

a. Note: Last IO for task B not shown (B ends at time 15)

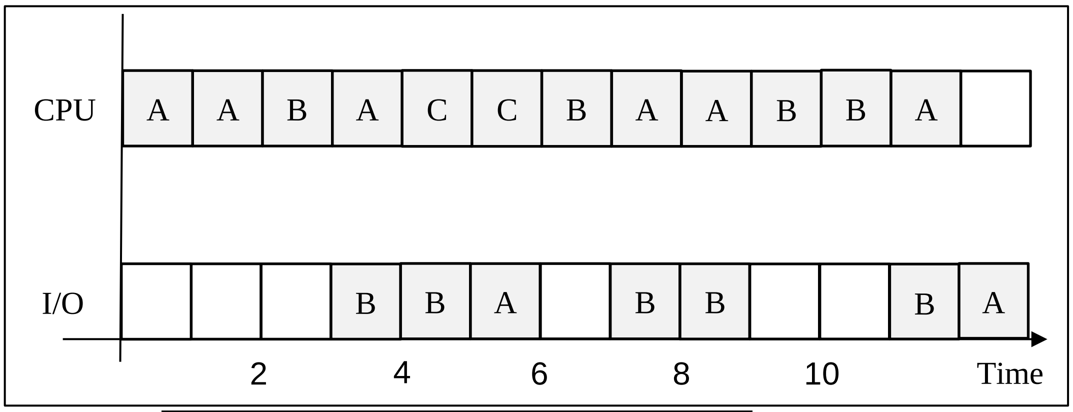


b.

	Turnaround Time	Waiting Time
A	**10**	**10 – 8 = 2**
B	**15**	**15 – 9 = 6**
C	**6 – 3 = 3**	**3 – 2 = 1**

c.

d.



Task	Response Time
A	0
B	2 – 0 = 2
C	4 – 3 = 1

The results highlight one of the strength of **pre-emptive** scheduling. With FIFO ordering, it is guaranteed that a task will get its time quantum in a finite amount of time (i.e. number of tasks arrived earlier).

2. (MLFQ) As discussed in the lecture, the simple MLFQ has a few shortcomings. Describe the scheduling behavior for the following two cases.

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

   2. (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.

ANS:

a. The process can sink to the lowest priority during the CPU intensive phase. With the low priority, the process may not receive CPU time in a timely fashion during the I/O phase, which degrades the responsiveness. The general shape of the timing chart is the same as the example 1 shown in lecture.

b. If a process gives up / blocks before the time quantum lapses, it will retain its priority. Since all processes enter the system with the highest priority, a process can keep its high priority indefinitely by using this trick and receive disproportionately more CPU time than other processes.

Amendments to MLFQ

i. This tweak fixes case (b). The trick in (b) works because the scheduler is “memory-less”, i.e. the CPU usage is counted from fresh every time a process receives a time quantum. If the CPU usage is accumulated, then a CPU intensive process will still exhaust the allowed time quantum and be demoted in priority. This will prevent the process from hogging the CPU.

ii. This tweak is for case (a). By periodically boosting the priority of all processes (essentially treat all process as “new” and hence have highest priority), a process with different behavior phases may get a chance to be treated correctly even after it has sank to the lowest priority.

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 );` 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?

ANS:

```
RunningTask.TQLeft--;
if (RunningTask.TQLeft > 0) done!
//Check for another task to run
if ( ReadyQ.isEmpty() )
    //renew time quantum
     RunningTask.TQLeft = TimeQuantum;
     done!

//Need context switching
TempTask = ReadyQ.dequeue();
//current task goes to the end of queue
```

```
ReadyQ.enqueue( RunningTask );

TempTask.TQLeft = TimeQuantum;
SwitchContext( RunningTask, TempTask );
```

b. The process will block during its time quantum and only get switched out once all of its time quantum has expired. The CPU remains idle until the process is switched out.

When the process is blocked on I/O or any other system level events, the process needs to make a **system call**, i.e. OS will be notified. So, it is possible to let the OS intercept such events and calls the scheduler directly from the system call routines. The timer interrupt is not involved in this process. Thus, the code should not be modified.

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.

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

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'.*

ANS:

a. It is likely you see all steps from one process get printed before another. When the delay is very small, the total work done across the 5 iterations is less than the time quantum given for a process. Hence, the process can finish all iterations before get swapped out.
b. It is likely you see interleaving pattern similar to (c). Each iteration in DoWork() now takes (multiple) time quanta to finish. Since each process will be swapped out once the time quantum expires, the printing will be in an interleaved pattern.
   [Note to instructor: Ask what happens if we increase the D further. Ensure they see that it could be **multiple time quanta** for each iteration]
c. The amount of time to loop D times and the cost of the printing is likely to be the time quantum used on your machine. Typical time quantum value is 10ms to 100ms.

Instructors can use `taskset --cpu-list 0 sudo perf stat ./Behaviors 2000000` (specifically the sudo perf stat portion) to show students the number of context switches, just to demonstrate that the theory is in line with practice.

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:	

ANS:

Process A			
Sequence	**Predicted**	**Actual**	**Percentage Error**
1	**10**	9	11.1%
2	**9.5**	8	**18.75%**
3	**8.75**	8	**9.38%**
4	**8.375**	7	**19.64%**
5	**7.6875**	6	**28.13%**
		Average Error:	**17.40%**

Process B			
Sequence	**Predicted**	**Actual**	**Percentage Error**
1	**10**	8	25%
2	**9**	14	**35.71%**
3	**11.5**	3	**283.33%**
4	**7.25**	18	**59.72%**
5	**12.625**	2	**531.25%**
		Average Error:	**187.00%**

Essentially, **α** represents the significance of the immediate past value, while (1- **α**) represent the significance of the past history. A higher **α** cause predicted value to fluctuate closer to the last value, while a lower **α** allow a predicted value that is closer to the historical value.

(Note that in this case, $\alpha = ½$ so the predicted value is actually ½ * Actual_n + ¼ * Actual_{n-1} + 1/8 * Actual_{n-2} + …, which explains the term "exponential")

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 2020 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.

ANS:

a. The older scheduling takes $O(N)$ time, where N is the number of runnable tasks. The latter is $O(1)$, as it is bounded by the number of priority levels (140), which is a constant.

b. This ensures all task get a chance to execute, regardless of priority level (when all higher priority tasks have expired, the low priority tasks will get their chance). The duration for running all tasks in the "active" set is known as a time epoch in Linux. Note that for brevity sake, we ignored an important point: If a task is "interactive" enough based on the heuristic mentioned in (d), it actually get added back to the "Active" set at the same priority level, i.e. it will get another time slice in this time epoch.

c. This is a form of balancing. A lower priority task gets picked less frequently but can run for a longer time. Also, Linux scheduler essentially degrades the priority of computationally intensive tasks to favor interactive tasks in the spirit of MLFQ. For a computationally intensive task, a longer time slice is exactly what it needed.

d. By taking active execution into account instead of just the blocking statistics, this allow the scheduler to distinguish:

- Largely computationally intensive process.
- Largely I/O intensive process, i.e. interactive process.
- Process that switches between the computation and I/O.

So, the interactive processes will get the bonuses they need to improve response time. At the same time, those "not-quite" interactive processes which alternate between heavy computation and I/O will not be awarded unnecessarily.

Instructors' Note: This scheduling algorithm is known as the $O(1)$ *scheduler*, and was used in Linux from releases 2.6.0 to 2.6.22. It was subsequently replaced by the *Completely Fair Scheduler* (CFS) in 2007, which remains the scheduler in use to date. Instead of using run queues, the CFS employs a red-black tree to schedule tasks. The scheduler mentioned in part (a) is even older – the $O(n)$ *scheduler*.