CS2106 Operating Systems
Tutorial 3 Suggested Solutions
Process Scheduling

1. [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.

    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 quantum to finish. Since each process will be swapped out once the time quantum expires, the printing will be in interleaved pattern.
[Note to instructor: Ask what happen if we increase the D further. Ensure they see that it could be **multiple time quantum** for each iterations]

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.

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

Program A, Arrives at time 0
Behavior (C**X** = Computer 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:

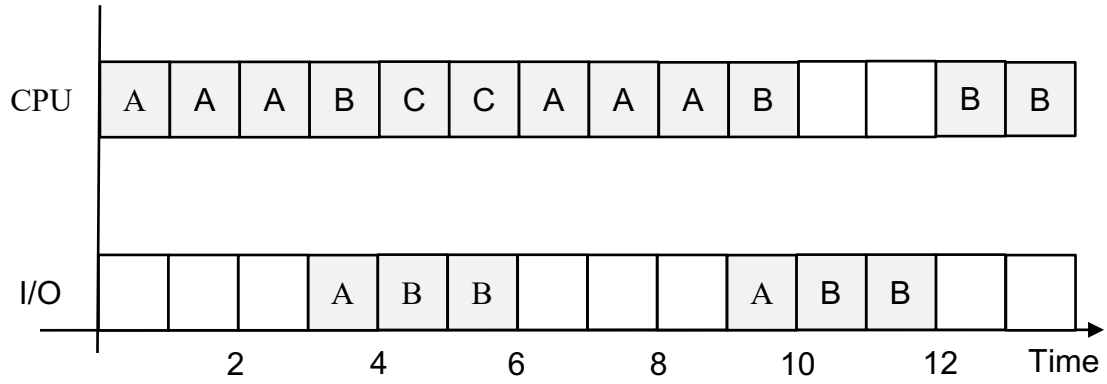
b. What are the turnaround time and the waiting time for program A, B and C? In this case, waiting time includes all time units where the program is ready for execution but could not get the CPU.

c. Use **Round Robin** algorithm to schedule the same set of tasks. Assume time quantum of **2 time units**.

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:

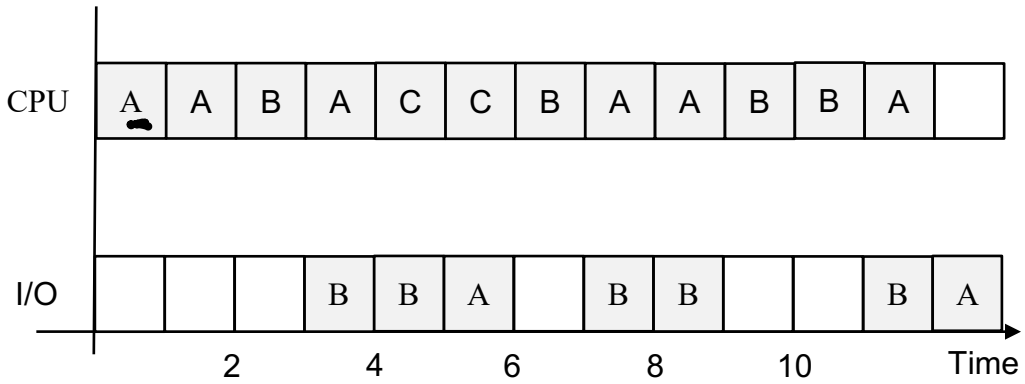
a. Note: Last IO for task B not shown (B ends at time 15)



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 strengths 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).

3. [Adapted from AY1617S1 Midterm – MLFQ] Consider the standard 3 levels MLFQ scheduling algorithm with the following parameters:
   - Time quantum for all priority levels is 2 **t**ime **u**nits (TUs).
   - Interval between timer interrupt is 1 TU.
   - The scheduler is **not** pre-emptive. I.e. a process gets to complete its time quantum even if a higher priority process is ready to run.

   a. Give the **CPU schedule** for the following 2 tasks. Use the given table as a template to fill in. Each box represents 1 time unit. The first time unit has been filled as an example.

Task A
Behavior: **CPU 3TUs**, *I/O 1TU*, **CPU 3TUs**

Task B
Behavior: **CPU 1TU**, *I/O 1TU*, **CPU 1TU**, *I/O 1TU*, **CPU 1TU**

Note that we only ask for the CPU schedule, you will have to keep track of the priority level of the tasks separately on your own.

CPU	A													
TU	1	2	3	4	5	6	7	8	9	10	11	12	13	14

ANS:

CPU	A	A	B	A	B	A	A	B	A					
TU	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Key points: Understanding of time quantum and ITI (not the same thing). Once a task is scheduled, it stays for the _time quantum_ unless it is stalled by I/O. There should not be any blanks in the schedule.

4. [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) Discuss how do you handle blocking of process on I/O or any other events. Key point: Should the code in (a) be modified (if so, how)? Or the handling should be performed somewhere else (if so, where)?

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 );
```

a. Note that for a process to access I/O devices or any other system level events, the process need to make a **system call**, i.e. OS will be notified. So, it is possible to let OS intercepts those events and calls the scheduler directly from the system call routines.

   The Linux Scheduler question shows another approach where the process is allowed to block during its time quantum and only get switched out when time quantum expires.

Additional Questions (For exploration only, not discussed in tutorial)

5. [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.

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 give up / blocks before the time quantum lapses, it will retain its priority. Since all process 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.

i. This tweak is to fix 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 overshoot the allowed time quantum and get a demotion 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.

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:

$$\mathbf{Predict_{N+1} = \alpha Actual_N + (1- \alpha)Predict_N}$$

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.

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 2018 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, your TA will pick several parts to discuss.

   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 task (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.

The priority adjustment is inversely proportional to the *sleep_avg*, i.e. high sleep value = large bonus, low sleep value = 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 is in O(N), where N is the number of runnable tasks. The latter is O(1), as it is bounded by the number of priority level (140, i.e. 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 a balancing. Lower priority task get picked less frequently but can run for a longer time. Also, Linux scheduler essentially degrades priority of computation intensive tasks to favor interactive tasks. For a computation intensive task, longer time slice is exactly what it needs.

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

- Largely computation 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 bonus it needs to improve response time. At the same time, those "not-quite" interactive process which alternate between heavy computation and I/O will not be awarded unnecessarily.