CS2106 Operating Systems
Semester 2 2020/2021

Week of 8th February 2021
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.

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:

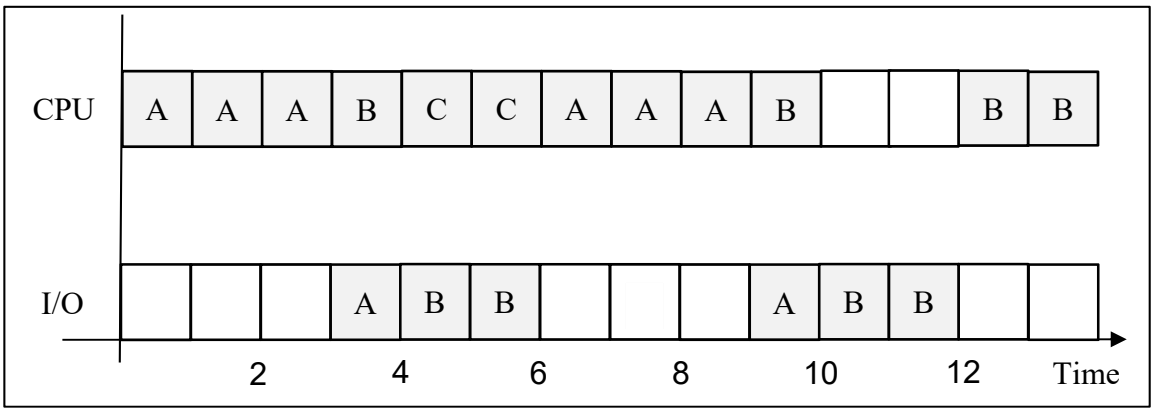
b. What are the turnaround time and the waiting time for program A, B and C? In this case, waiting time includes all time units where the program is ready for execution but could not get the CPU.

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:

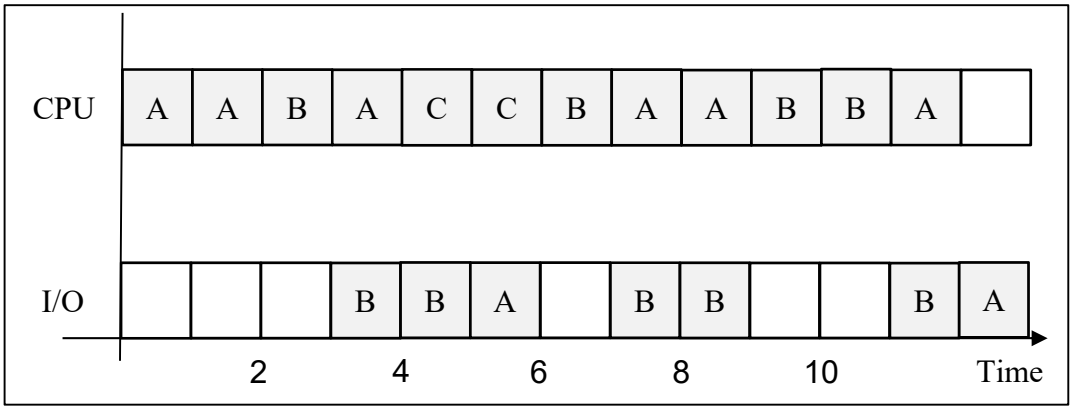
a. Note: Last IO for task B not shown (B ends at time 15)



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 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).

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.

   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 uses another approach where the process is allowed to block during its time quantum and only get switched out when time quantum expires.