CS2106

Tutorial03

Q1 Scheduling Behaviour (Behaviors.c)

```
void DoWork(int iterations, int delay) {
    int i, j;
    for (i = 0; i < iterations; i++) {
        printf("[%d]: Step %d\n", getpid(), i);
        for (j = 0; j < delay; j++);    //introduce some fictional work
    }
}
int main(int argc, char* argv[]) {
    int childpid, delay;
    delay = atoi(argv[1]); childpid = fork();
    if (childpid == 0) { //1st child
        DoWork(5, delay);
        printf("[%d] Child Done!\n", getpid());
    } else {
        DoWork(5, delay);
        wait(NULL);
        printf("[%d] Parent Done!\n", getpid());
    }
    return 0;
}
```

Q1 Scheduling Behaviour

Use the command `taskset --cpu-list 0 ./Behaviors D`

Why do we need to restrict our program to run on only 1 CPU core?

1(a)

How many processes are there?

What do we see when D = 1?

1(a)

```
[27411]: Step 0
[27411]: Step 1
[27411]: Step 2
[27411]: Step 3
[27411]: Step 4
[27412]: Step 0
[27412]: Step 1
[27412]: Step 2
[27412]: Step 3
[27412]: Step 4
[27412] Child Done!
[27411] Parent Done!
```

Why is the pattern this way?

1(a)

```
[27411]: Step 0
[27411]: Step 1
[27411]: Step 2
[27411]: Step 3
[27411]: Step 4
[27412]: Step 0
[27412]: Step 1
[27412]: Step 2
[27412]: Step 3
[27412]: Step 4
[27412] Child Done!
[27411] Parent Done!
```

- 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.

1(b)

What do we see when D = 100,000,000?

1(b)

```
[46344]: Step 0
[46345]: Step 0
[46344]: Step 1
[46345]: Step 1
[46344]: Step 2
[46345]: Step 2
[46344]: Step 3
[46345]: Step 3
[46344]: Step 4
[46345]: Step 4
[46345] Child Done!
[46344] Parent Done!
```

Why is the pattern this way?

1(b)

```
[46344]: Step 0
[46345]: Step 0
[46344]: Step 1
[46345]: Step 1
[46344]: Step 2
[46345]: Step 2
[46344]: Step 3
[46345]: Step 3
[46344]: Step 4
[46345]: Step 4
[46345] Child Done!
[46344] Parent Done!
```

- There is an interleaving pattern.
- Each iteration in DoWork() now likely 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.

1(c)

Find the smallest D that gives you the interleaving output pattern

What do you think "D" represents?

1(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.

Q2 Scheduling Algorithm

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**

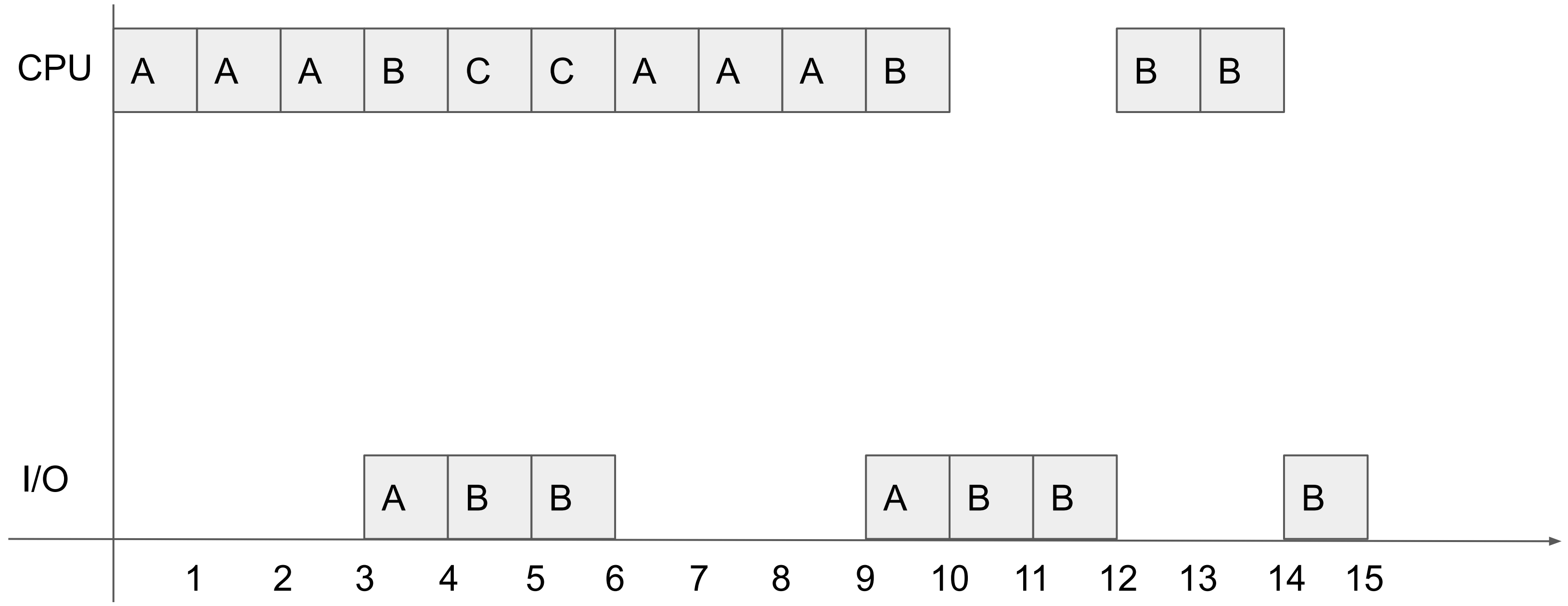
Program B, Arrives at time 0
Behavior: C**1**, IO**2**, C**1**, IO**2** C**2**, IO**1**

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

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.

2(a)



2(b)

What are the turnaround times and the waiting times 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.

	Turnaround Time	Waiting Time
A		
B		
C		

2(b)

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

2(c)

Use Round Robin algorithm to schedule the same set of tasks.

Assume time quantum of 2 time units.

2(c)

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**

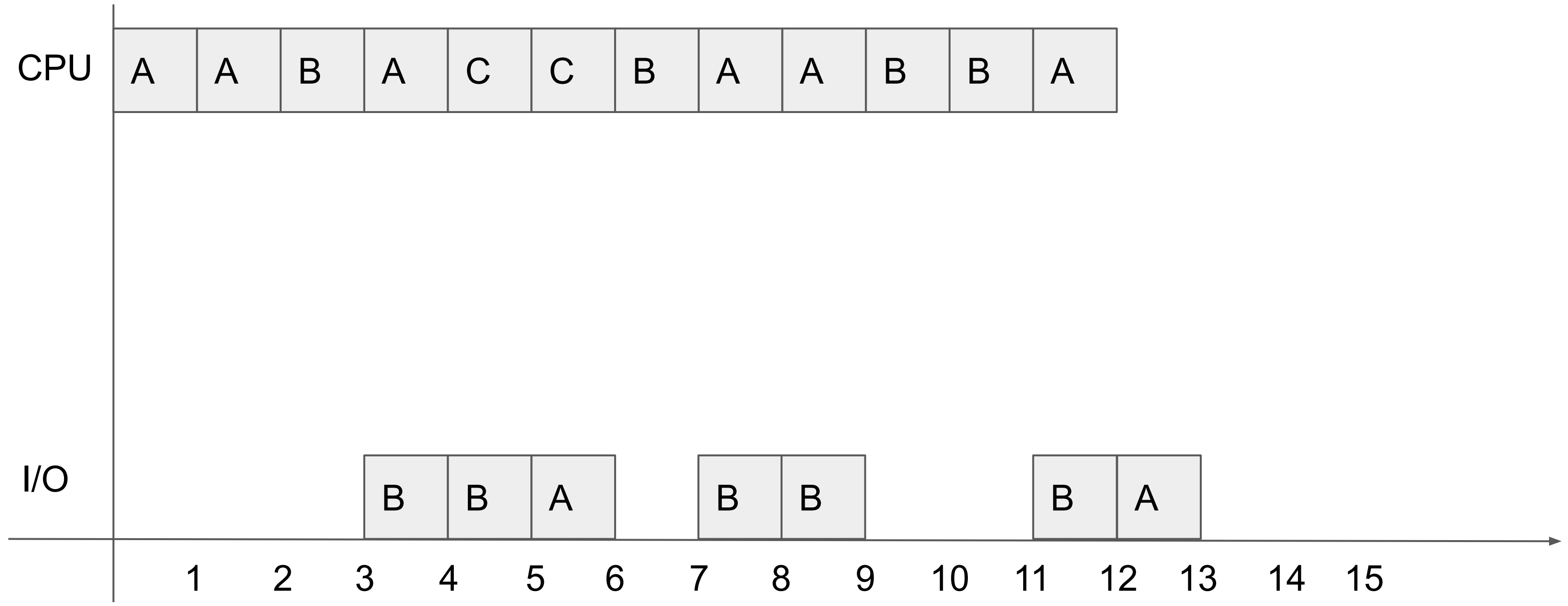
Program B, Arrives at time 0

Behavior:
C**1**, IO**2**, C**1**, IO**2** C**2**, IO**1**

Program C, Arrives at time 3

Behavior:
C**2**

2(c)



2(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

Task	Response Time
A	
B	
C	

2(d)

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

The results highlight one of the strength of preemptive 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).

Q3 RR Scheduler

Give the pseudocode for the RRscheduler function.

For simplicity, you can assume that all tasks are CPU intensive that run 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

Q3 RR Scheduler

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()**.
TimeQuantum 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.

3(a)

```
RunningTask.TQLeft--;
if (RunningTask.TQLeft > 0) done!

if ( ReadyQ.isEmpty() )      // Check for another task to run
    RunningTask.TQLeft = TimeQuantum;       // Renew time quantum
    done!

// Need context switching
TempTask = ReadyQ.dequeue();
ReadyQ.enqueue( RunningTask );    // Current task goes to the end of queue

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

3(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)?

3(b)

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.

It is possible to let OS intercepts those 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.

Q4 Exponential Average

Let us try to see exponential average 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:

$\text{Predict}(N+1) = \alpha\text{Actual}(N) + (1- \alpha)\text{Predict}(N)$

Calculate the error percentage (|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.

Q4

$$Predict(N+1) = \alpha Actual(N) + (1 - \alpha)Predict(N)$$

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

Q4

$$\text{Predict}(N+1) = \alpha\text{Actual}(N) + (1-\alpha)\text{Predict}(N)$$

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%**

Q4

$$Predict(N+1) = \alpha Actual(N) + (1- \alpha)Predict(N)$$

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

Q4

Predict(N+1) = αActual(N) + (1- α)Predict(N)

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%**

Q4

The prediction algorithm performs better for process A than that for process B

Process A is more predictable as the cpu burst period is more consistent