Tutorial 3

CS2106: Introduction to Operating Systems

Motivation for Scheduling

- OS needs to manage the allocation of the CPU to multiple processes.
- Goal: Maximize CPU utilization, minimize CPU idle time.

CPU

macOS

Operating System

zoom

Processes

Recap: Scheduling Algorithms

Algorithm	Batch / Interactive	Pre-emptive	Starvation
First Come First Served	Batch	No	No
Shortest Job First		No	Yes
Shortest Remaining Time		Yes	Yes
Round Robin	Interactive	Yes	No
Priority Based		Yes	Yes
Multi Level Feedback Queue		Yes	Yes
Lottery Scheduling		Can be both	No

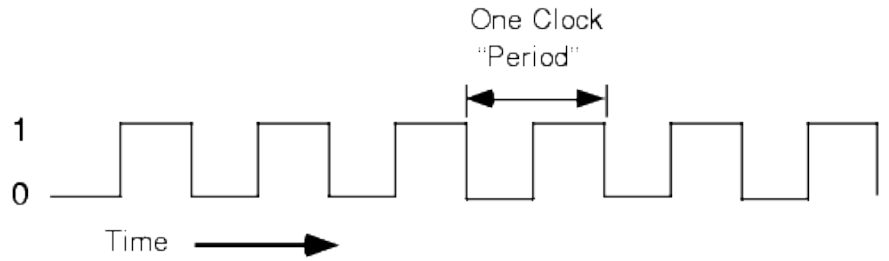
CPU Starvation: A situation where a process is unable to access the CPU for an extended period of time because other processes with higher priority are continuously being executed

Recap: Scheduling Algorithms

- **Non Pre-emptive Scheduling:** A process stays scheduled (in running state) until it blocks or give up the CPU voluntarily.
- **Pre-emptive Scheduling:** A running process can be suspended when
  1. It's uses up its allocated time slice in Round Robin (RR), the OS scheduler, suspends it and the next process in the ready queue is scheduled to run.
  2. A higher priority process arrives in Priority Scheduling and Multi-Level Feedback Queue.
  3. A shorter process with shorter remaining execution time arrives in Shortest Remaining Time.

Preemptive Scheduling: Timer Interrupt

CPU



- In preemptive scheduling, we want the CPU to be periodically interrupted.
  - So that we can perform context switching to another process when
    1. The current running process has used up its time slice.
    2. A new process of higher priority has arrived in the ready queue. (for Priority-Based Scheduling & MLFQ)
    3. A new process with shorter remaining CPU execution time arrives.
- Using the clock of the CPU, we can generate Timer Interrupt events at fixed intervals (e.g. 10ms)

Preemptive Scheduling: Timer Interrupt

- When a timer interrupt occurs, the current process execution is suspended and the interrupt handler runs.
- The interrupt handler will first save the current process context (registers, etc…) then invoke the Interrupt Service Routine (ISR).
- The ISR would usually invoke the Scheduler.

Recall from Lecture 2 Slide 68, that interrupts are asynchronous and when it happens, control is handed over to the handler routine, in this case during pre-emptive scheduling, the handler routine is the ISR which invokes the scheduler.

The OS will (1) Save Register/CPU state, (2) Perform the handler routine, (3) Restore Register/CPU and finally (4) Return from interrupt

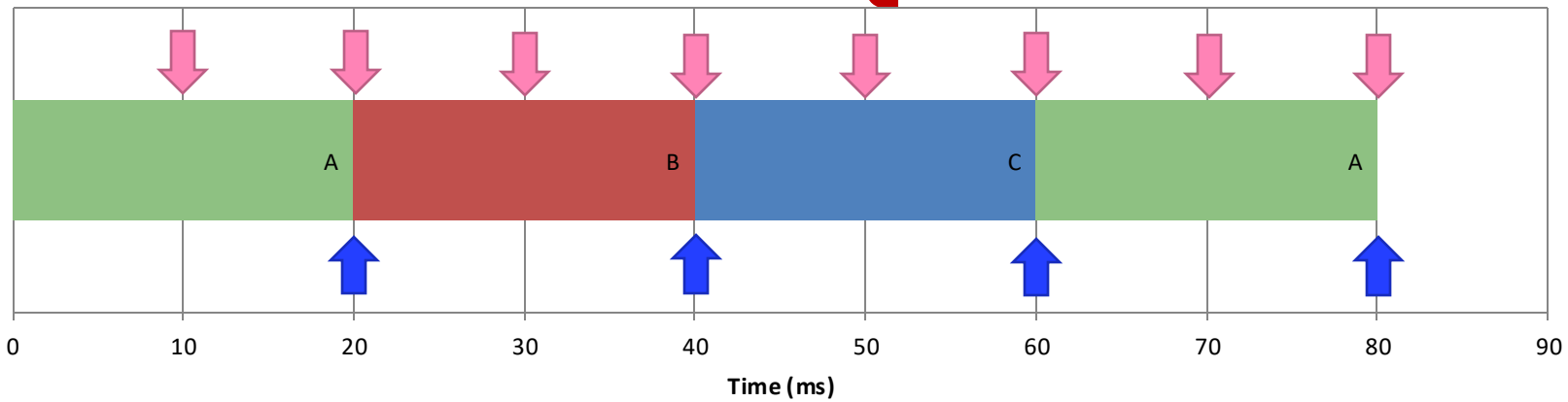
Preemptive Scheduling: Timer Interrupt

- The Scheduler would then decide the next process to be given CPU time or resume execution for the current process.
- This depends on the scheduling algorithm used.
- If the scheduler selects another process to be given CPU time, then it will need to do a context switch.

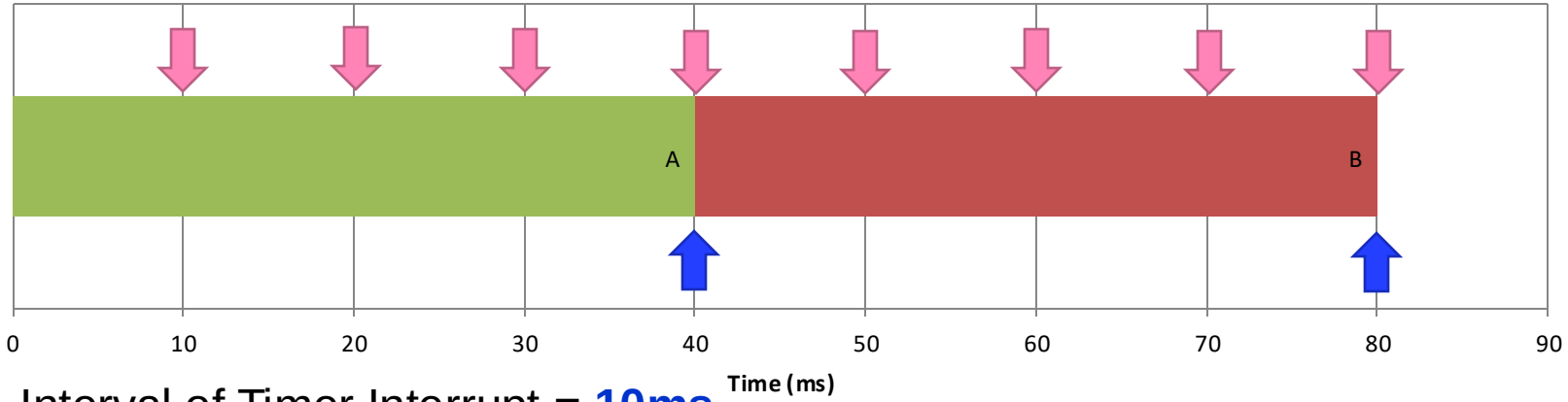
Recap: Preemptive Scheduling

- **Interval of Timer Interrupt (ITI)**
  - A timer interrupt goes off periodically based on the hardware clock.
  - This interval is usually set from 1ms (Jiffy in Linux) to 10ms
  - The timer interrupt is handled by the Interrupt Service Routine (ISR)
  - The ISR invokes the OS scheduler which will decide the next process to receive CPU time.
- **Time Quantum**
  - Execution duration given to a process
  - A multiple of the Interval of Timer Interrupt.

Illustration: ITI vs Time Quantum



Interval of Timer Interrupt = **10ms**
Time Quantum = **20ms**



Interval of Timer Interrupt = **10ms**
Time Quantum = **40ms**

Question 1

Question 1

You are given a 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.

Use ideas you have learned from Lecture 3: Process Scheduling to explain the program behavior in part (a) and (b).

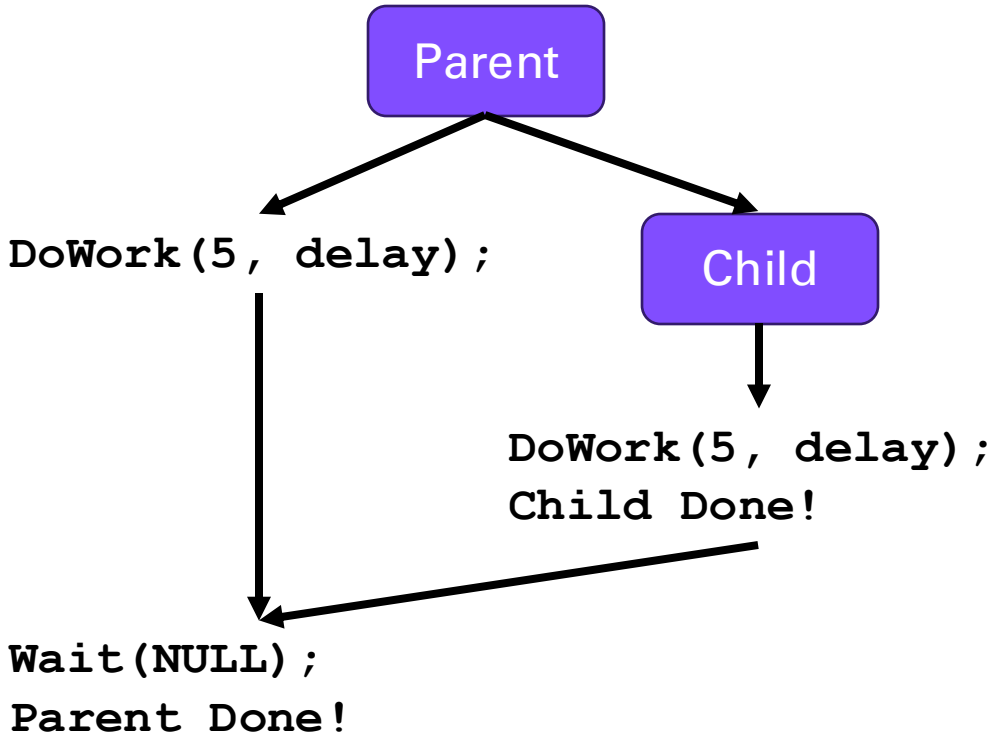
a) D = 1

b) D = 1000,000,000

Question 1

(a) D = 1

(b) D = 1,000,000,000




```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
  }
}
```

Question 1

(a) D = 1

- It is likely to 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.

Question 1

(b) D = 1,000,000,000

- It is likely you see an interleaving pattern.
- 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.

Question 1

(c) Find the smallest D that gives you the following interleaving output pattern.

- 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.

Question 2

Question 2

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**

Question 2(a): First-Come-First-Serve

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

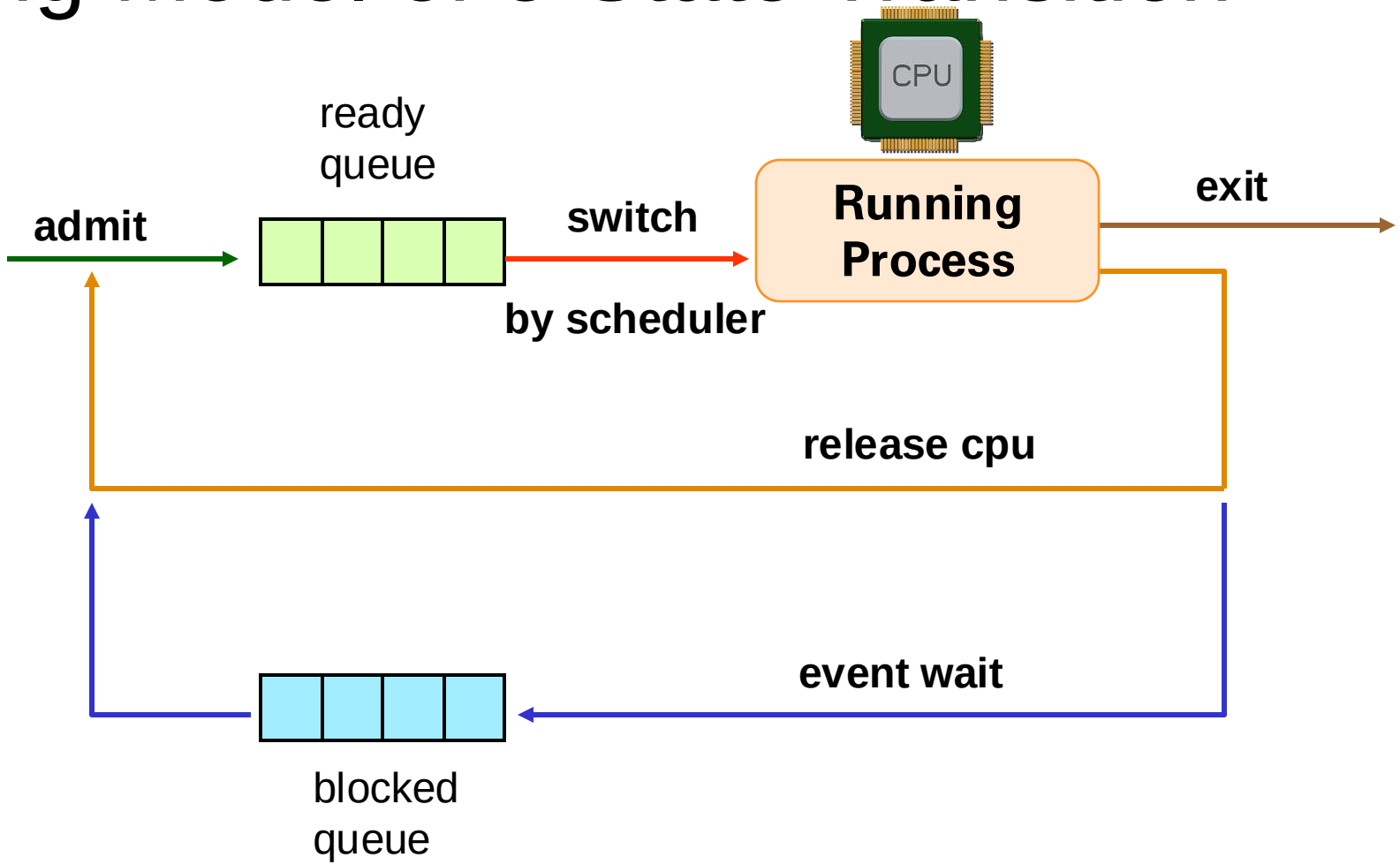
Below is a sample sketch up to time 1:

First Come First Serve

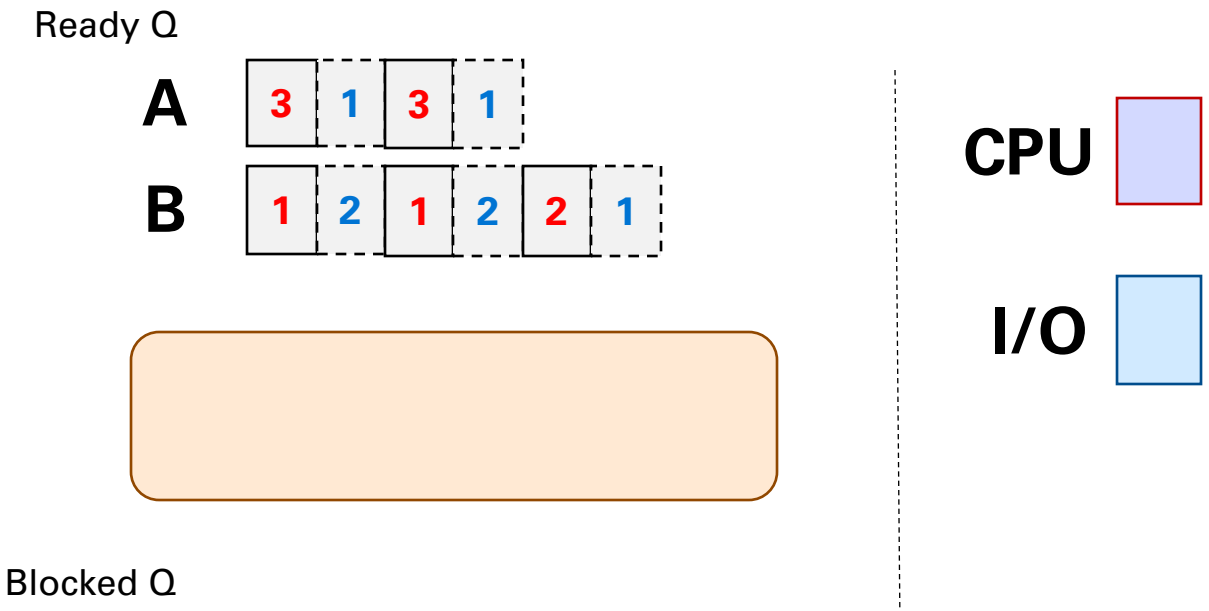
- We assume scheduler kicks in at the beginning of the time step whenever it is triggered

➔Show the result of the scheduling for the time step after scheduler finished its job

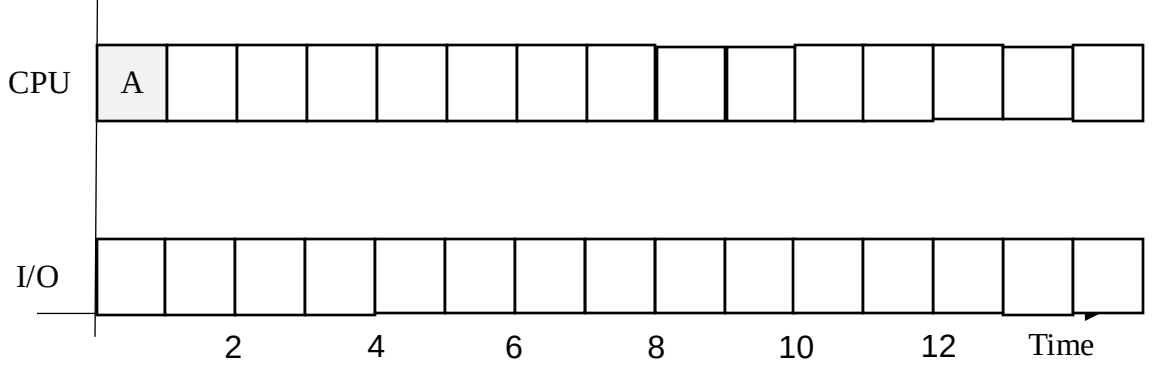
Queuing Model of 5 State Transition



Initialization



Time: 1



Ready Q

B 1 2 1 2 2 1

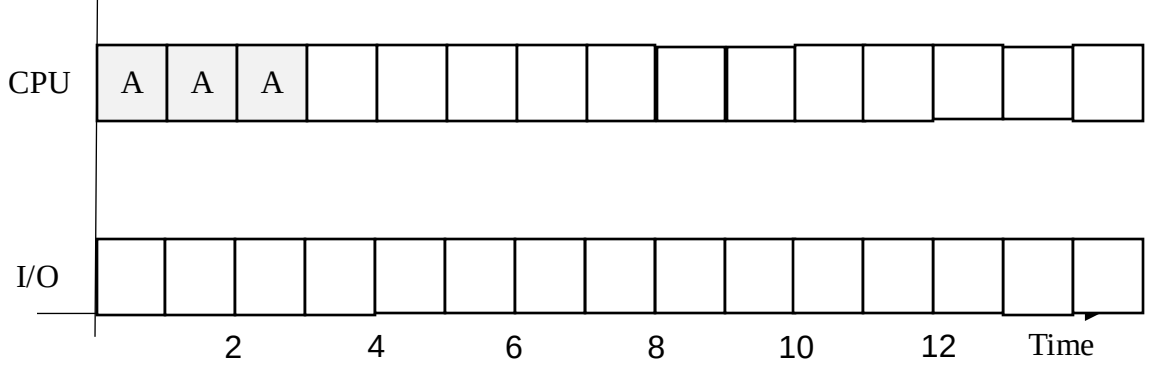
A is chosen as it is the first in Q

Blocked Q

CPU **A** 3 1 3 1

I/O

Time: **3**



Ready Q

B 1 2 2 1

C 2

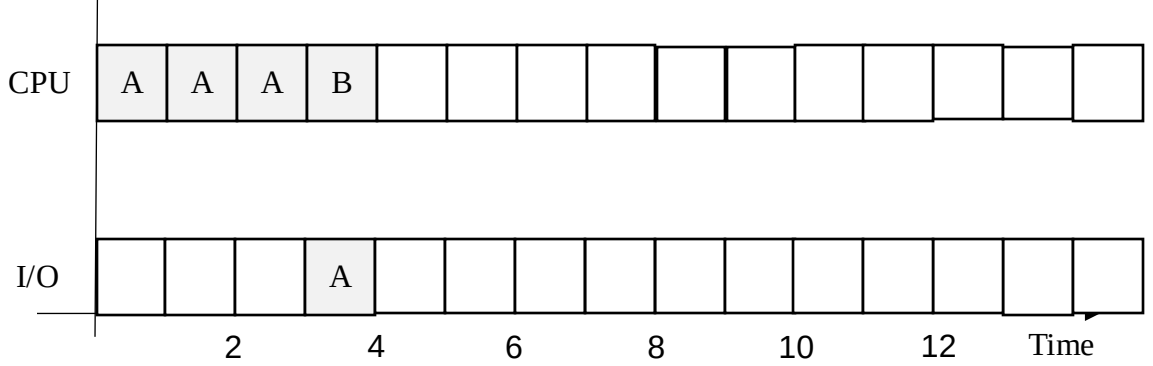
A still working, C arrives

Blocked Q

CPU **A** 1 1 3 1

I/O

Time: **4**



Ready Q

C 2

CPU **B** 1 2 1 2 2 1

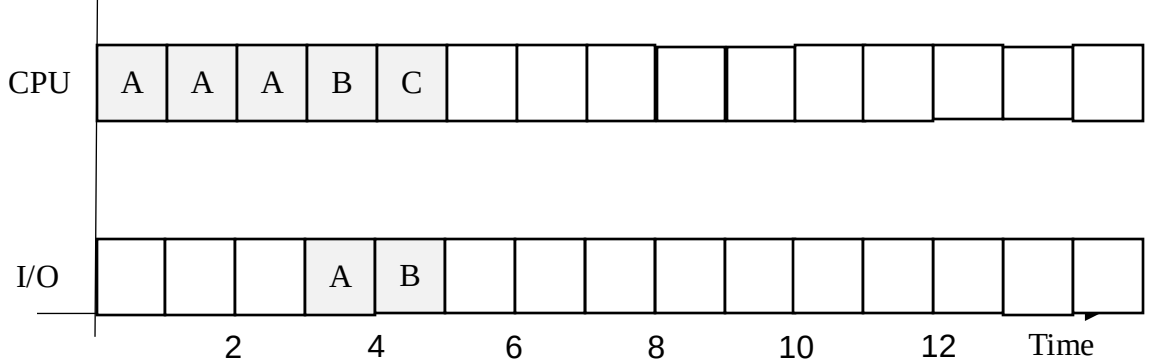
I/O **A**

A blocks, B get CPU

Blocked Q

A 1 3 1

Time: **5**



Ready Q

A 3 1

CPU **C** 2

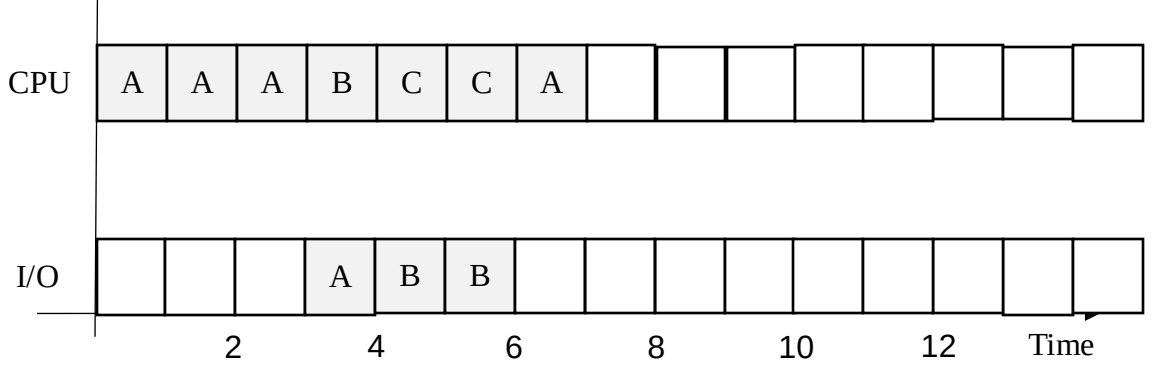
I/O **B**

B blocks, C get CPU

Blocked Q

B 2 1 2 2 1

Time: 7



Ready Q

B 1 2 2 1

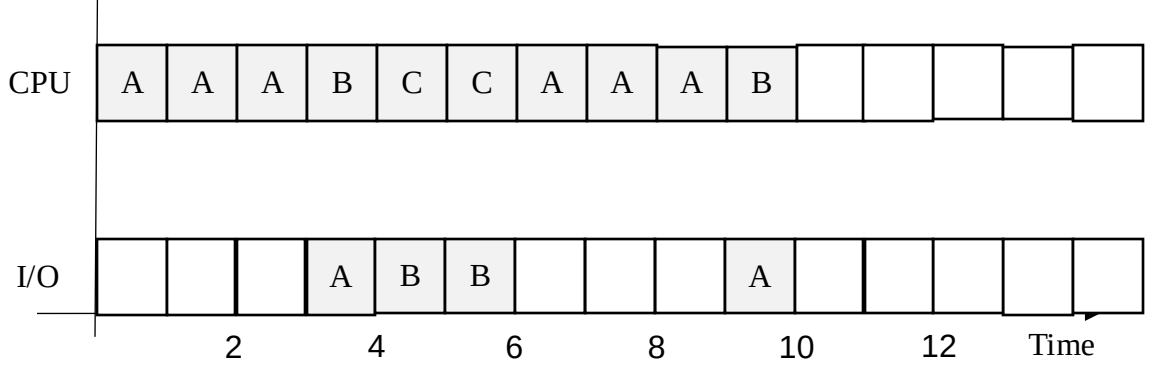
CPU **A** 3 1

I/O

C done; A get CPU; B unblocks

Blocked Q

Time: **10**



Ready Q

CPU **B** 1 2 2 1

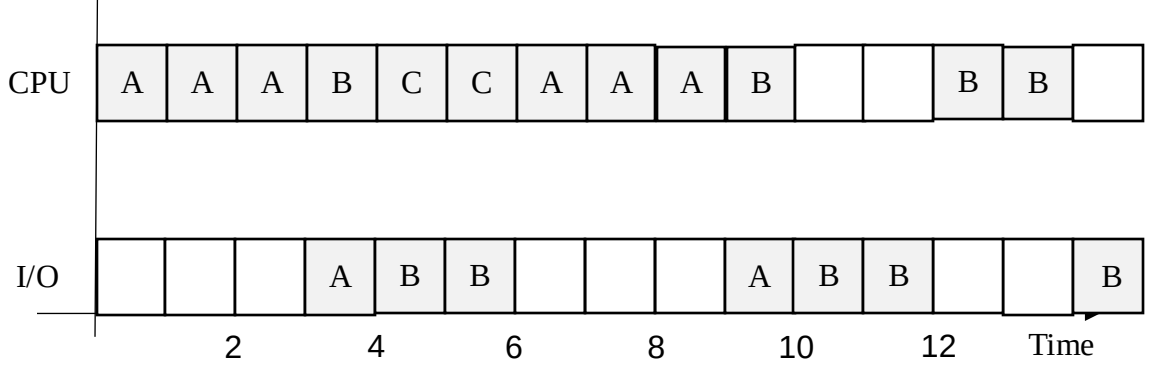
I/O **A**

A blocks; B get CPU

Blocked Q

A 1

Time: **15**

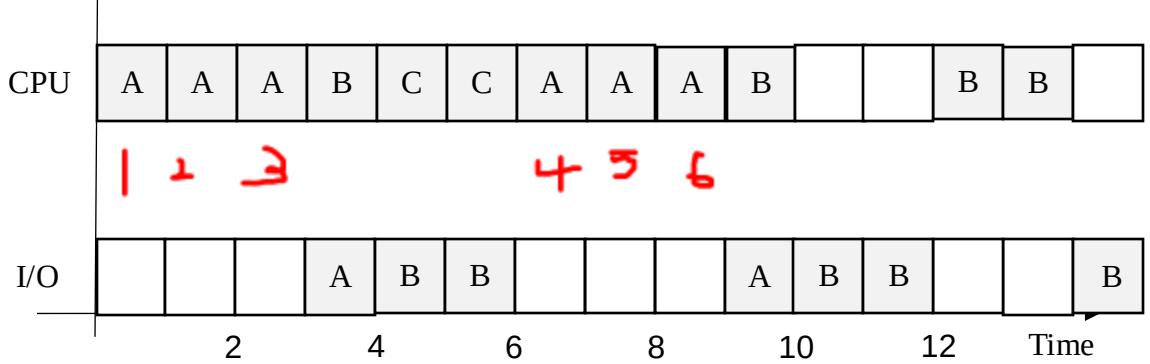


Question 2(b)

	Turnaround Time	Waiting Time
A		
B		
C		

Turnaround Time: Finish - Start Time
Waiting Time: Time spent waiting for CPU

Time: **15**



Question 2(b)

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

Turnaround Time: Finish - Start Time
Waiting Time: Time spent waiting for CPU

Question 2(c): Round Robin

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

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**

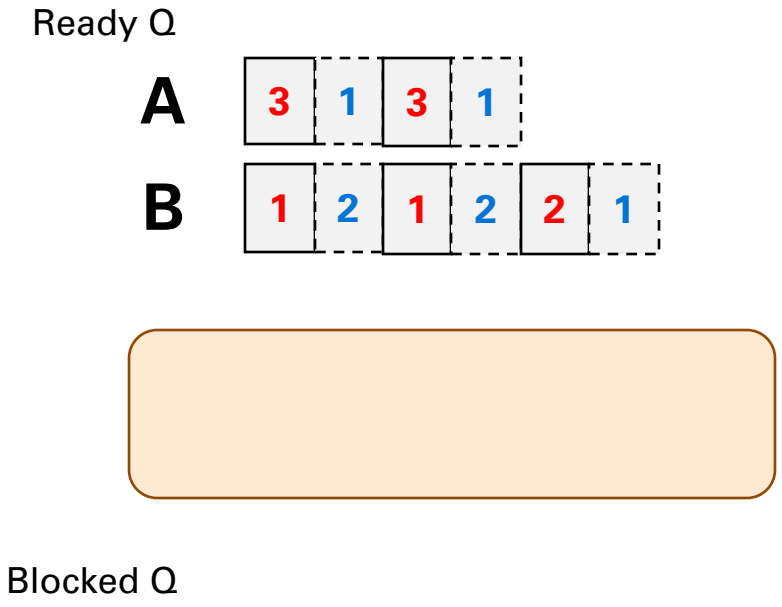
Program C, Arrives at time 3
Behavior: C**2**

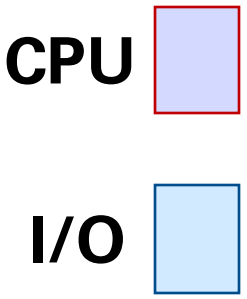
Round Robin

- We assume scheduler kicks in at the beginning of the time step whenever it is triggered

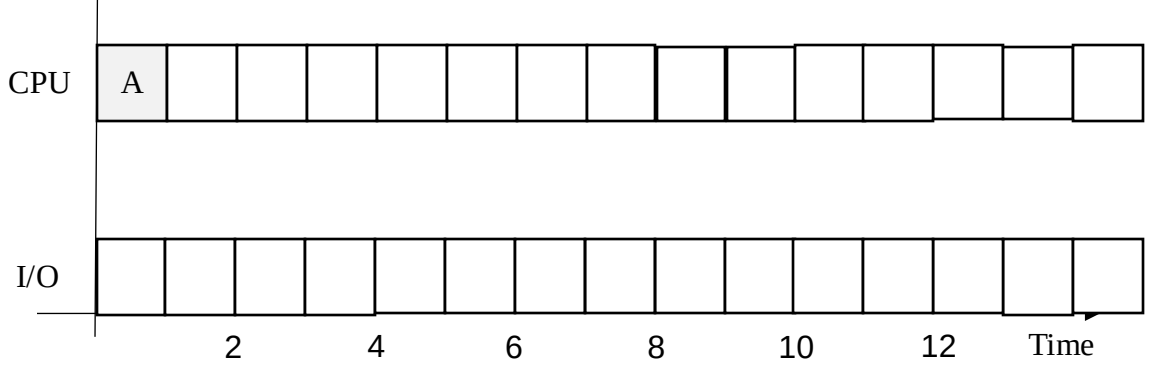
➔Show the result of the scheduling for the time step after scheduler finished its job

Initialization





Time: 1



Ready Q

B 1 2 1 2 2 1

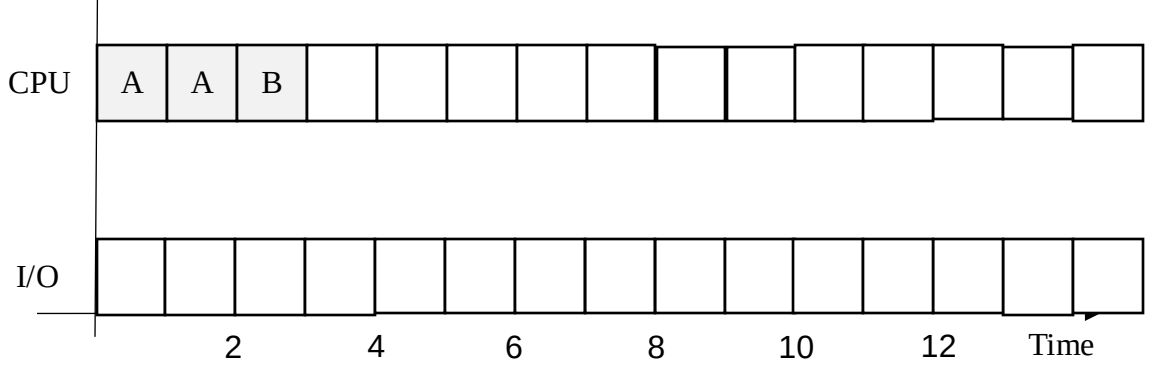
A is chosen as it is the first in Q

Blocked Q

CPU **A** 3 1 3 1

I/O

Time: **3**



Ready Q

A 1 1 3 1

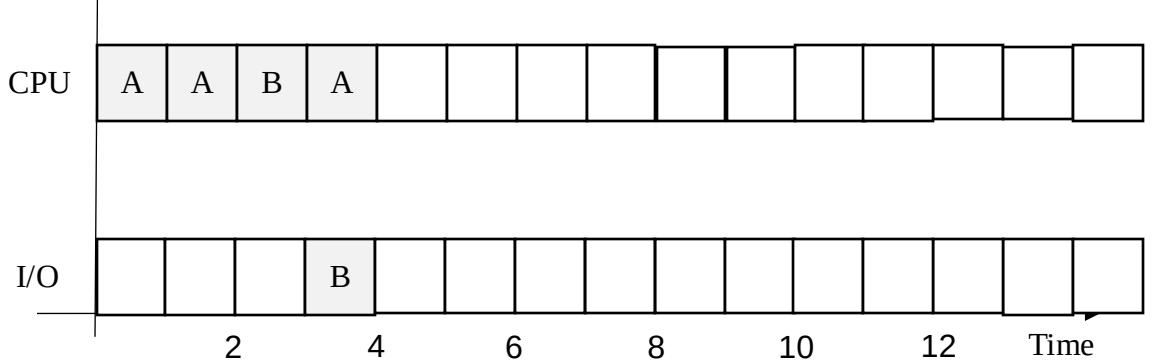
A is removed from CPU after 2 TUs.

Blocked Q

CPU **B** 1 2 1 2 2 1

I/O

Time: **4**



Ready Q

C 2

CPU **A** 1 1 3 1

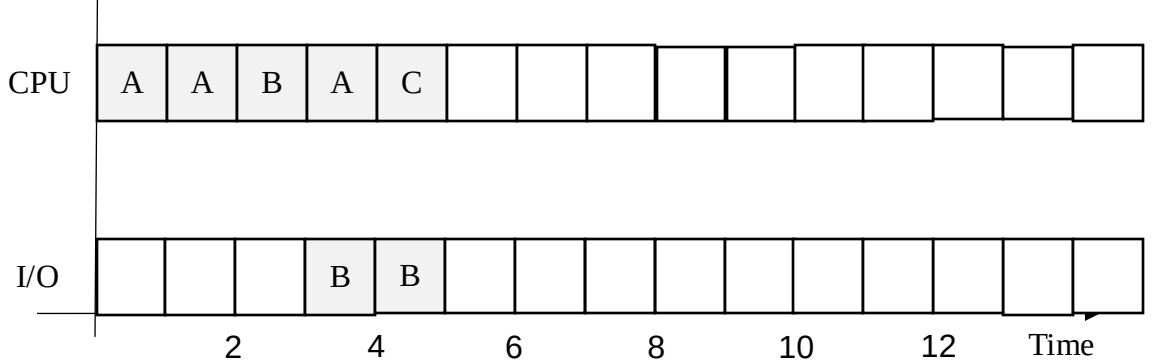
I/O **B**

A get chosen (first in Q);
B blocks; C arrives

Blocked Q

B 2 1 2 2 1

Time: **5**



Ready Q

CPU **C** **2**

I/O B

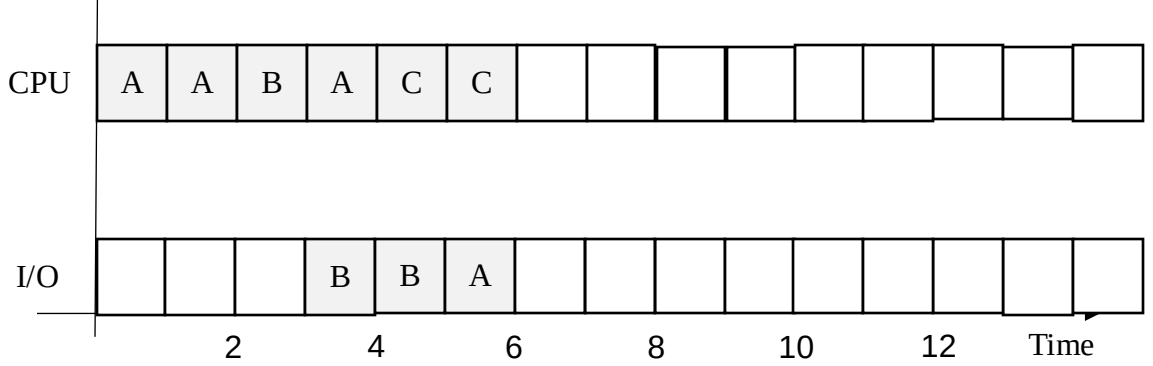
C is the only ready process

Blocked Q

B 1 1 2 2 1

A 1 3 1

Time: **6**



Ready Q

B 1 2 2 1

B unblocks; C continues

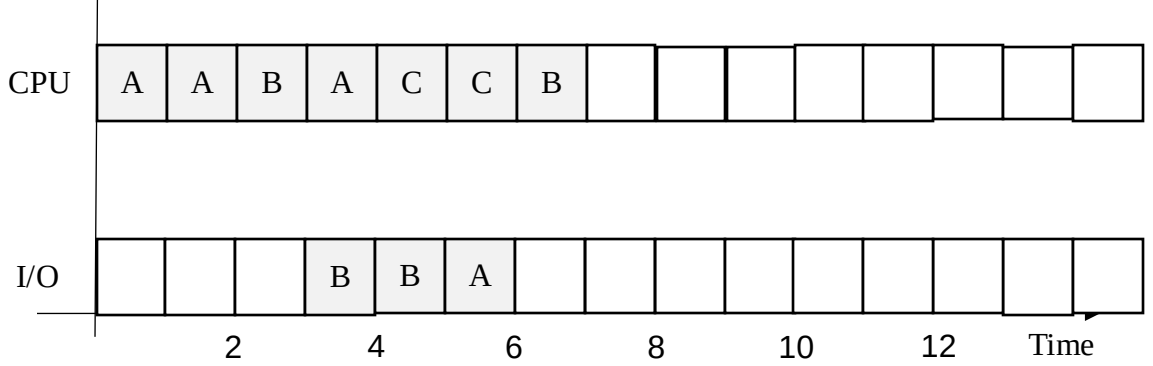
Blocked Q

A 1 3 1

CPU C 1

I/O A

Time: 7



Ready Q

A 3 1

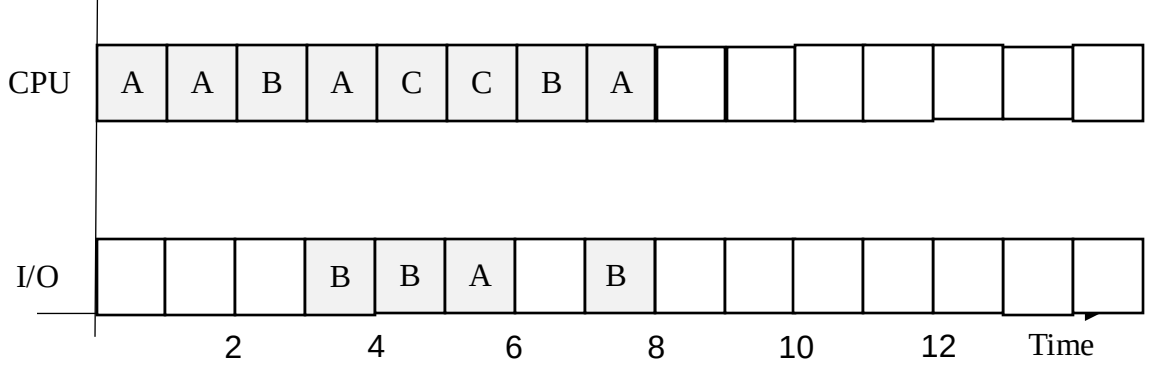
CPU **B** 1 2 2 1

I/O

B is chosen. A unblocks.

Blocked Q

Time: **8**



Ready Q

CPU **A** 3 1

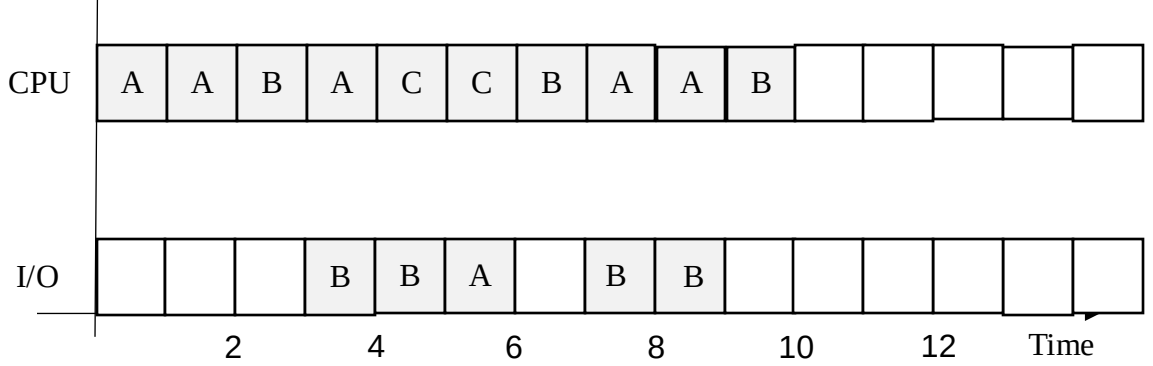
I/O **B**

B blocks.

Blocked Q

B 2 2 1

Time: **10**



Ready Q

A 1 1

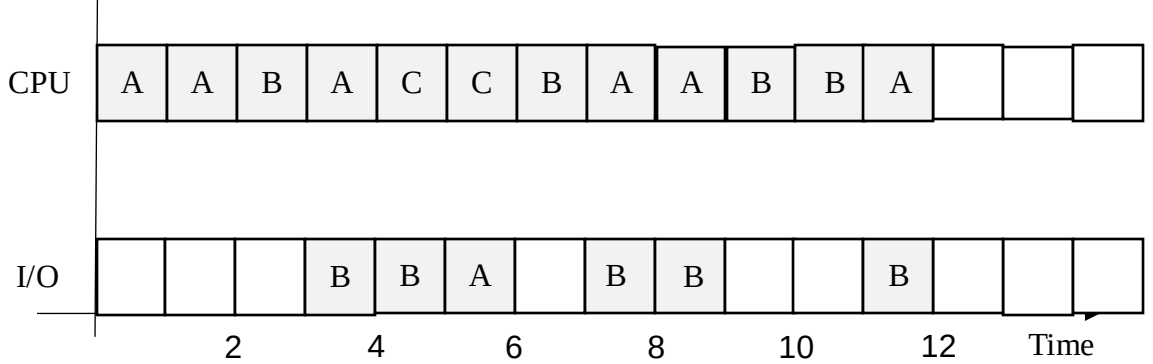
CPU **B** 2 1

I/O

A vacates CPU. B unblocks & get chosen.

Blocked Q

Time: **12**



Ready Q

CPU **A** 1 1

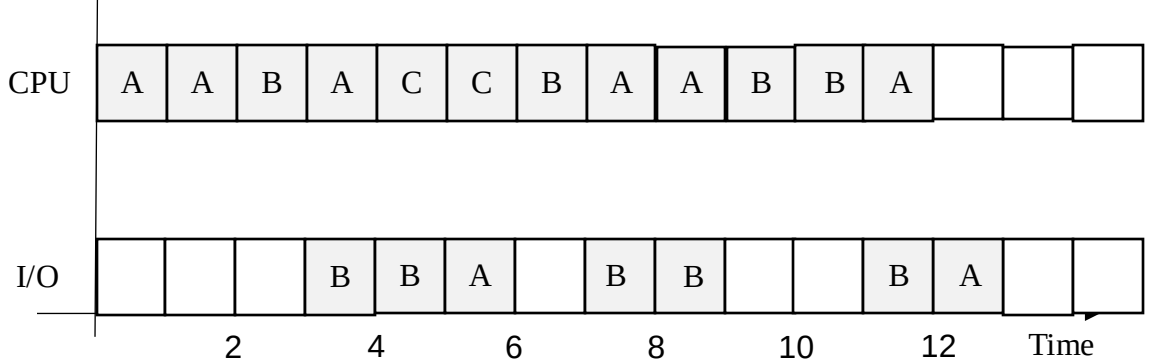
I/O **B**

B blocks.

Blocked Q

B 1

Time: **13**



Ready Q

CPU

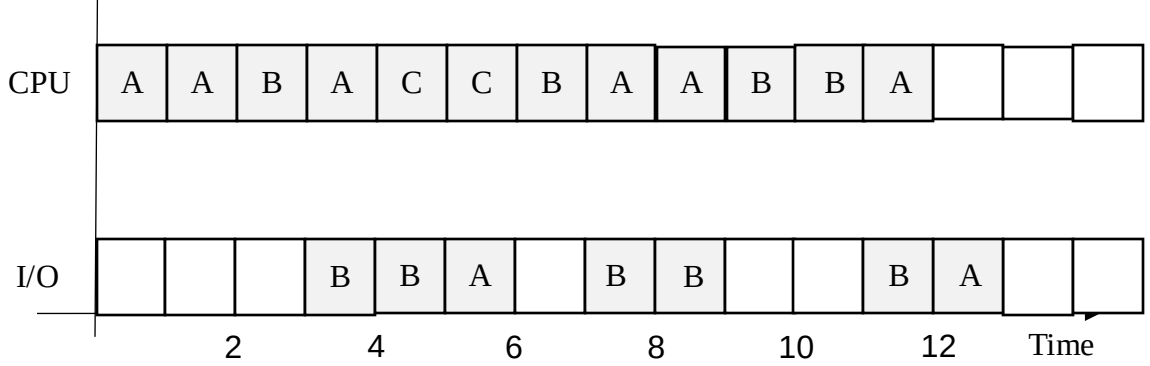
I/O **A**

B done; A blocks

Blocked Q

A 1

Time: **14**



Question 2(d)

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

Response Time: Time between request and response by system

Question 3

Question 3: MLFQ

Consider the standard 3 levels MLFQ scheduling algorithm with the following parameters:

- Time quantum for all priority levels is 2 time units (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.)

Question 3: MLFQ

Give the CPU schedule for the following 2 tasks.

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**

In this question, both Task A and B arrive at time T=0.
Task A is ahead of Task B in the Ready Queue.

MLFQ Rules

- **Basic rules:**
  1. If Priority(A) > Priority(B) ➔ A runs
  2. If Priority(A) == Priority(B) ➔ A and B run in RR

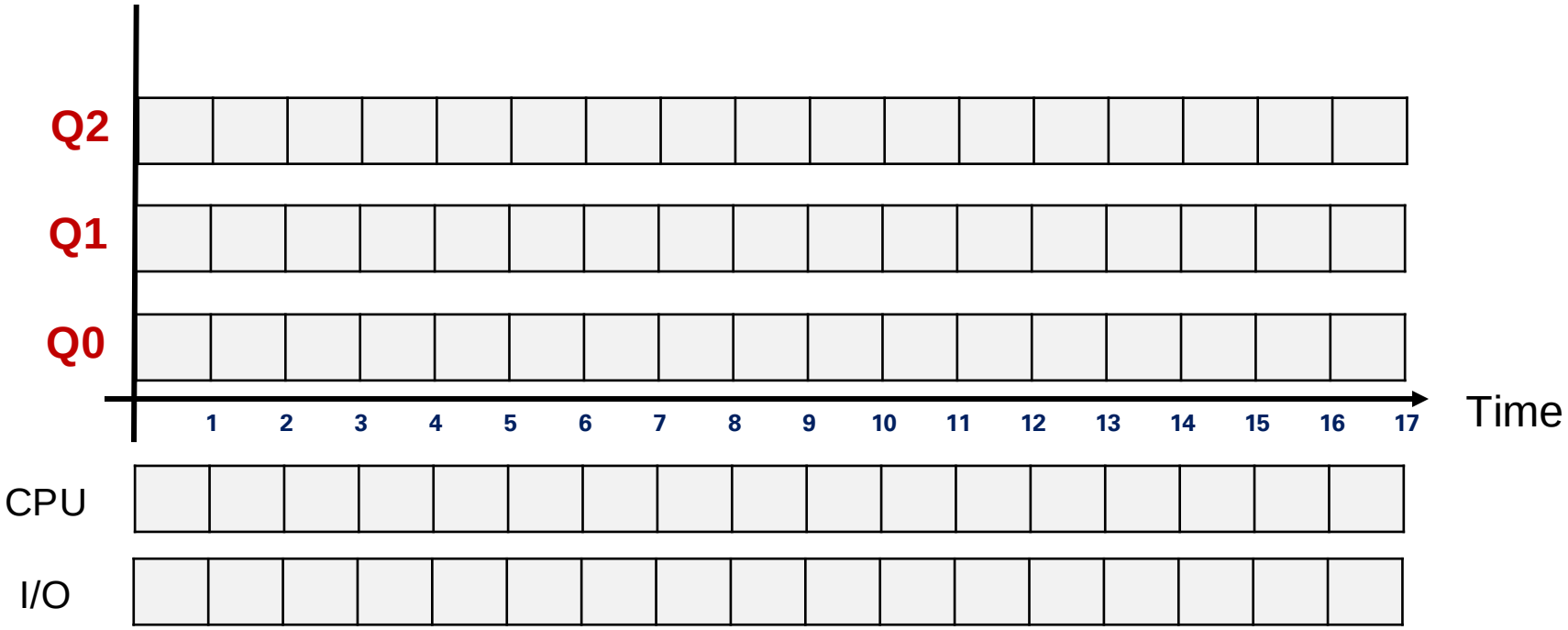
- **Priority Setting/Changing rules:**
  1. New job ➔ Highest priority
  2. If a job fully utilized its time slice ➔ Priority Reduced
  3. If a job give up / blocks before it finishes the time slice ➔ priority retained.

Question 3

Arrive At	Process	Behavior	Priority
0	A	**C3**, IO1, **C3**	**2**
0	B	**C1**, IO1, **C1**, IO1, **C1**	**2**

Time Quantum = 2 TU
Interval Between Interrupt = 1 TU

Time = 0

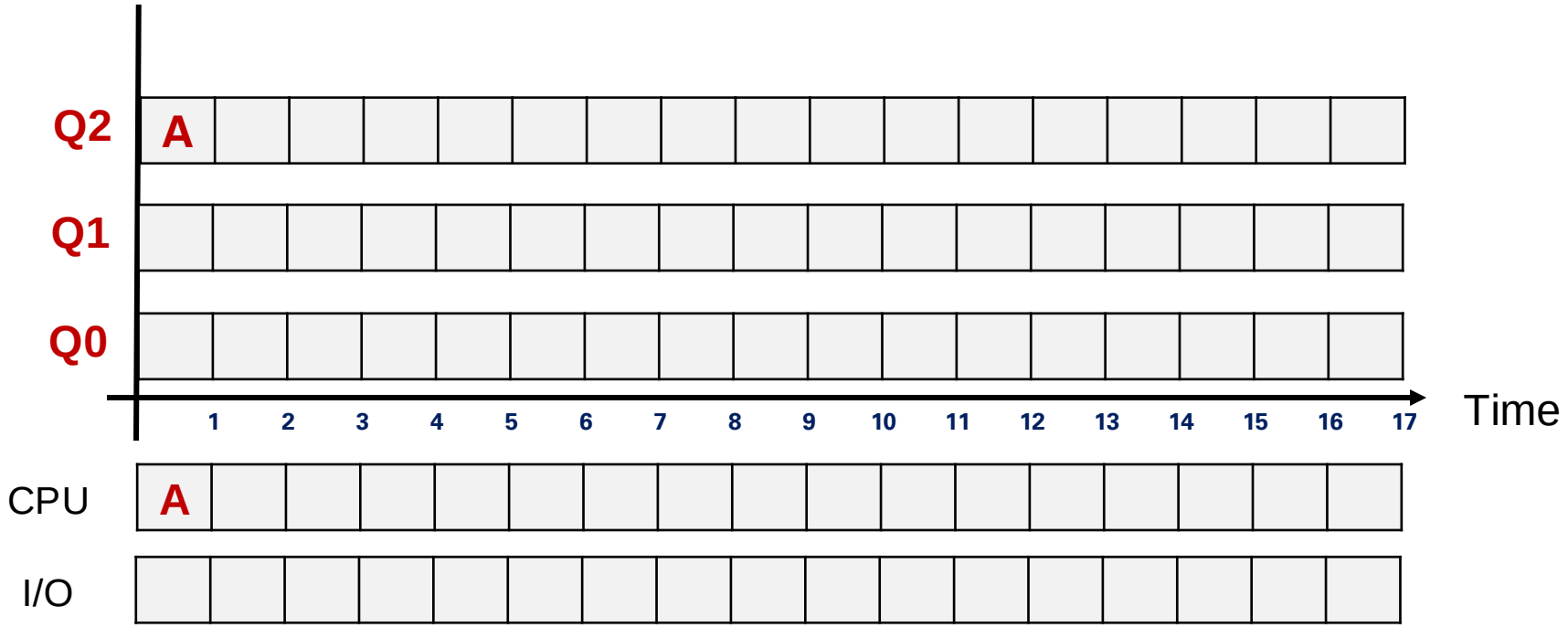


Question 3

Arrive At	Process	Behavior	Priority
0	A	**C3**, IO1, **C3**	**2**
0	B	**C1**, IO1, **C1**, IO1, **C1**	**2**

Time Quantum = 2 TU
Interval Between Interrupt = 1 TU

Time = 1

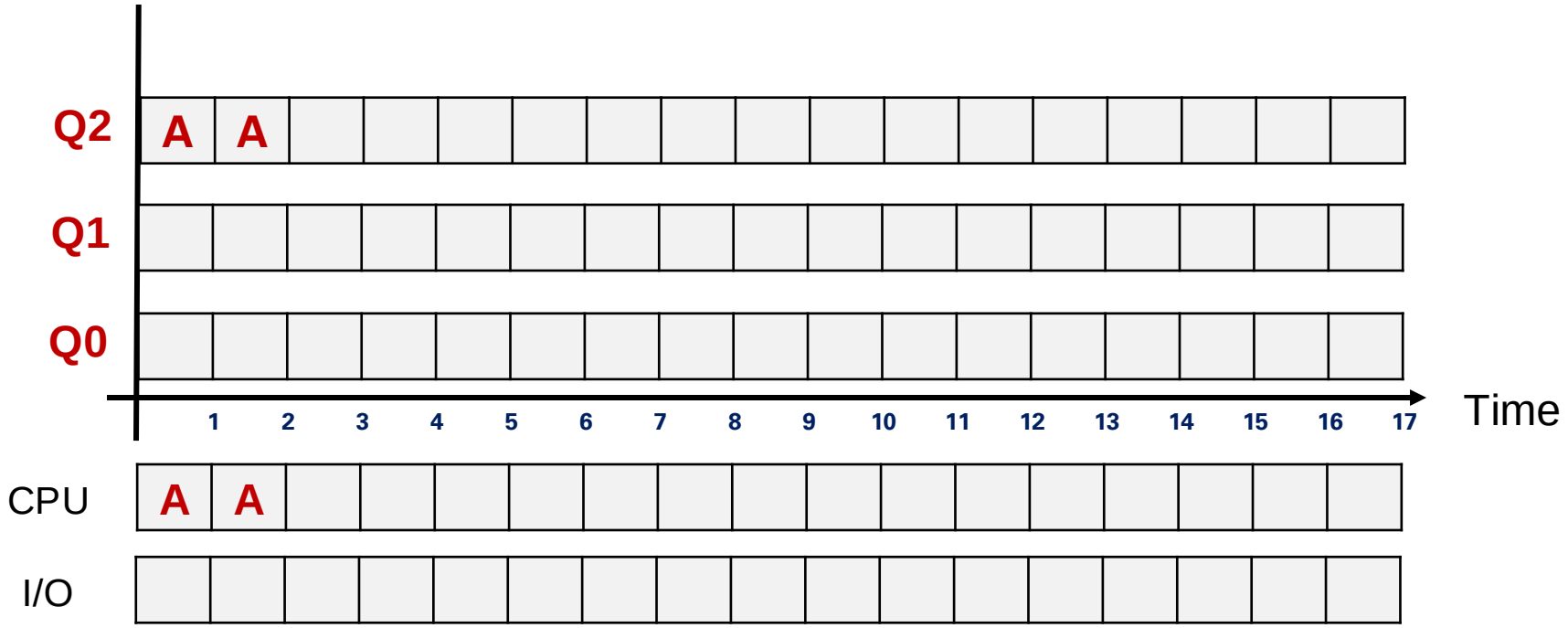


Question 3

Arrive At	Process	Behavior	Priority
0	A	**C3**, IO1, **C3**	**1**
0	B	**C1**, IO1, **C1**, IO1, **C1**	**2**

Time = 2

Time Quantum = 2 TU
Interval Between Interrupt = 1 TU

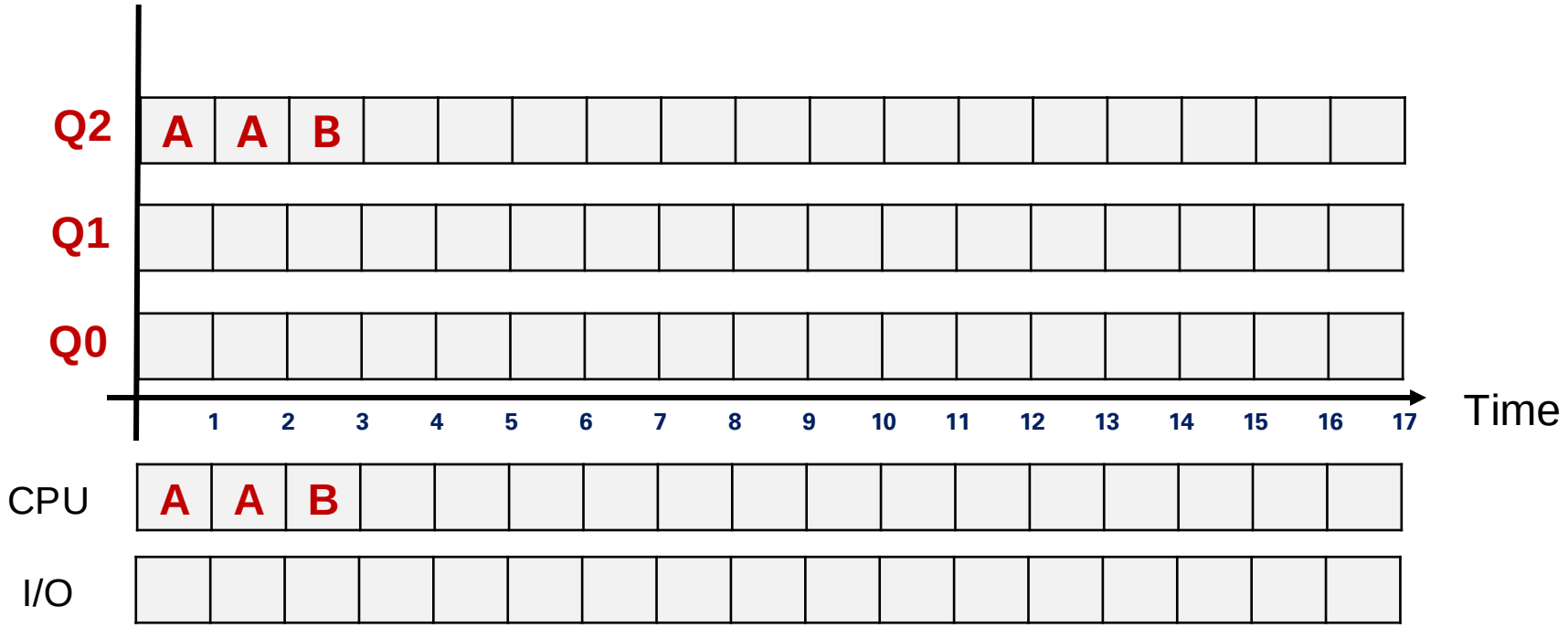


Question 3

Arrive At	Process	Behavior	Priority
0	A	**C3**, IO1, **C3**	**1**
0	B	**C1**, IO1, **C1**, IO1, **C1**	**2**

Time = 3

Time Quantum = 2 TU
Interval Between Interrupt = 1 TU

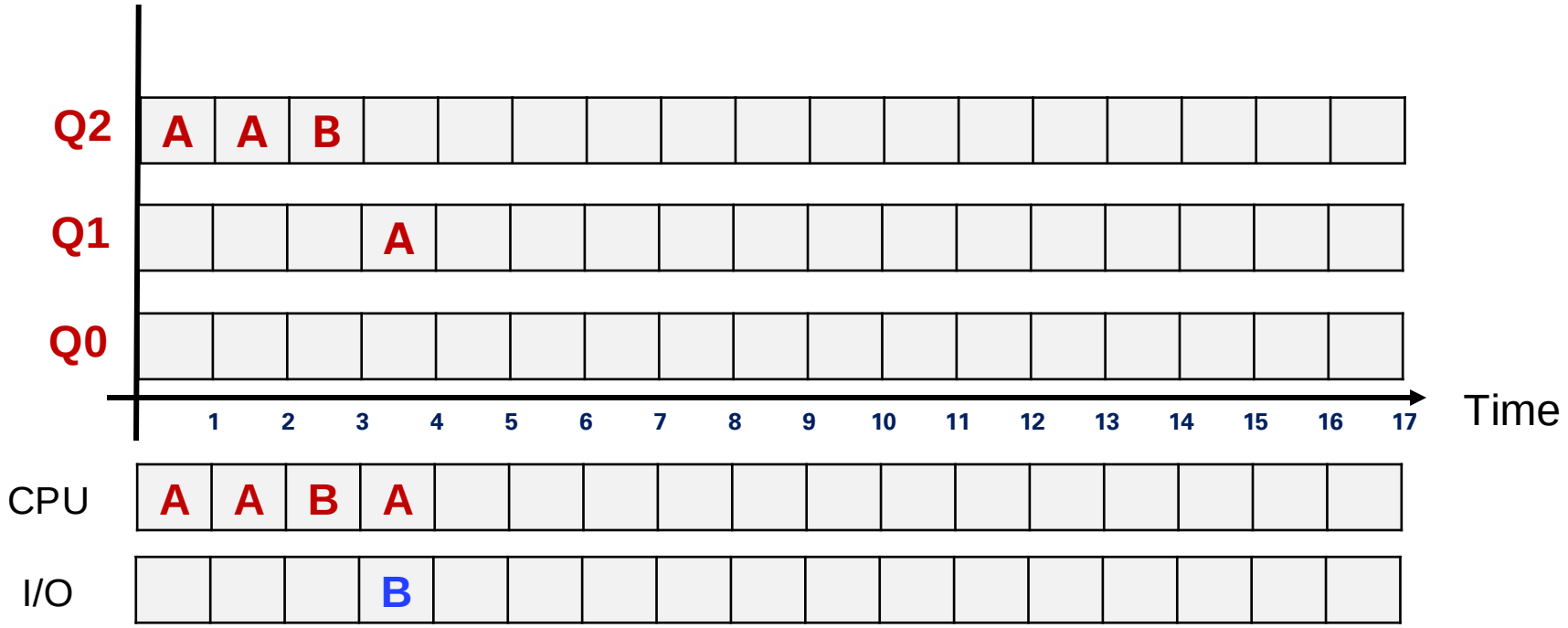


Question 3

Arrive At	Process	Behavior	Priority
0	A	**C3**, IO1, **C3**	**1**
0	B	**C1**, IO1, **C1**, IO1, **C1**	**2**

Time Quantum = 2 TU
Interval Between Interrupt = 1 TU

Time = 4

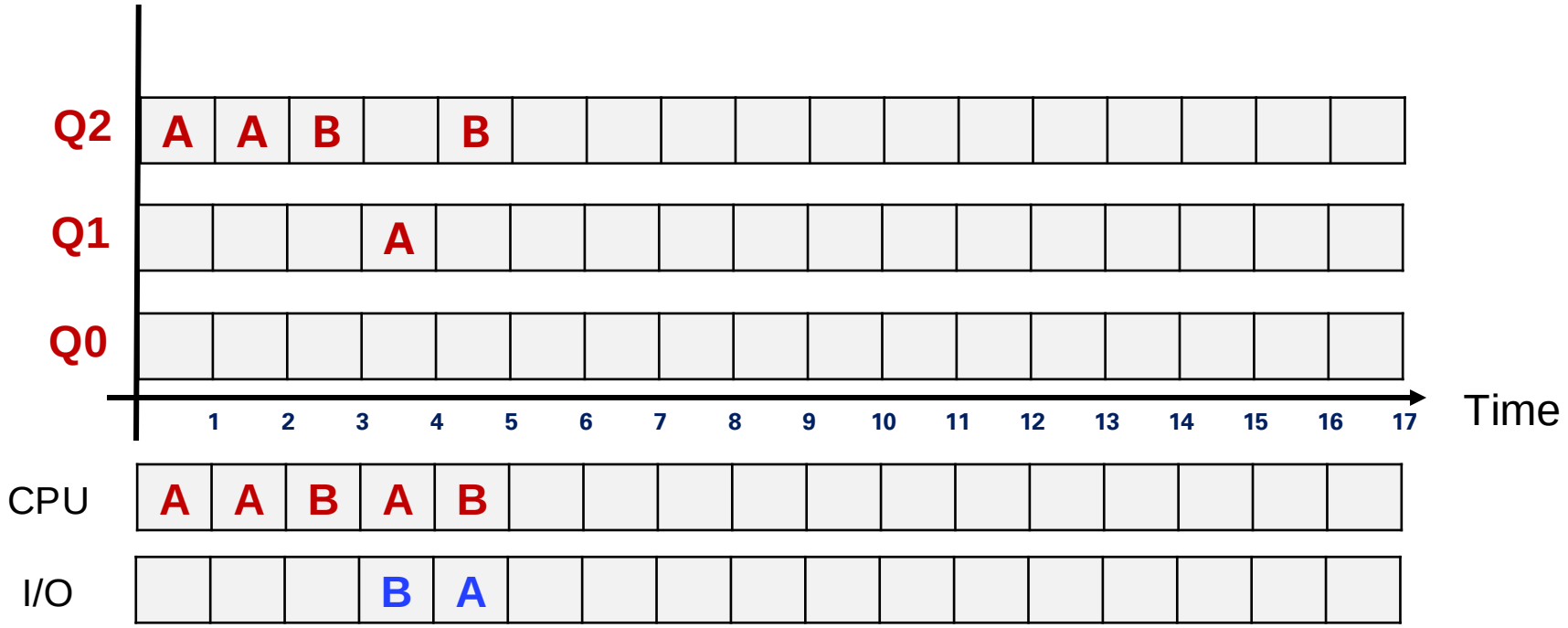


Question 3

Arrive At	Process	Behavior	Priority
0	A	**C3**, IO1, **C3**	**1**
0	B	**C1**, IO1, **C1**, IO1, **C1**	**2**

Time = 5

Time Quantum = 2 TU
Interval Between Interrupt = 1 TU

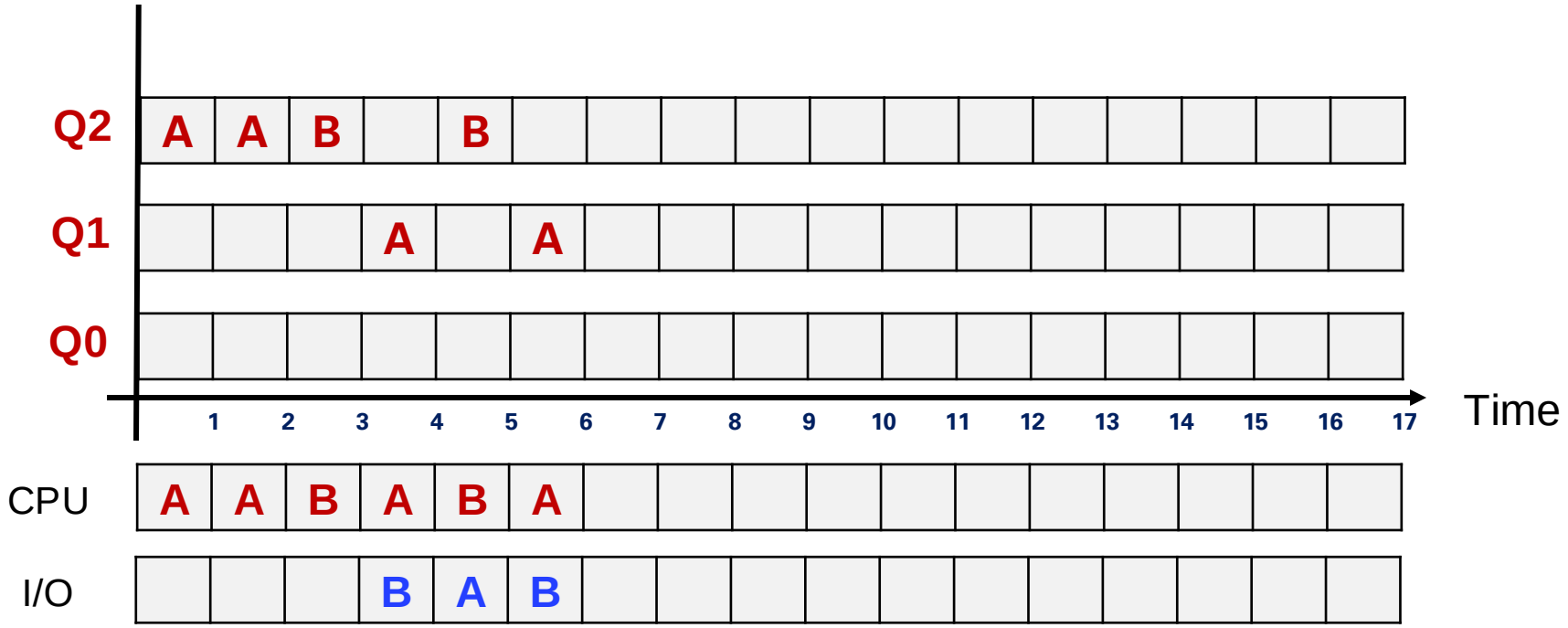


Question 3

Arrive At	Process	Behavior	Priority
0	A	**C3**, IO1, **C3**	**1**
0	B	**C1**, IO1, **C1**, IO1, **C1**	**2**

Time Quantum = 2 TU
Interval Between Interrupt = 1 TU

Time = 6

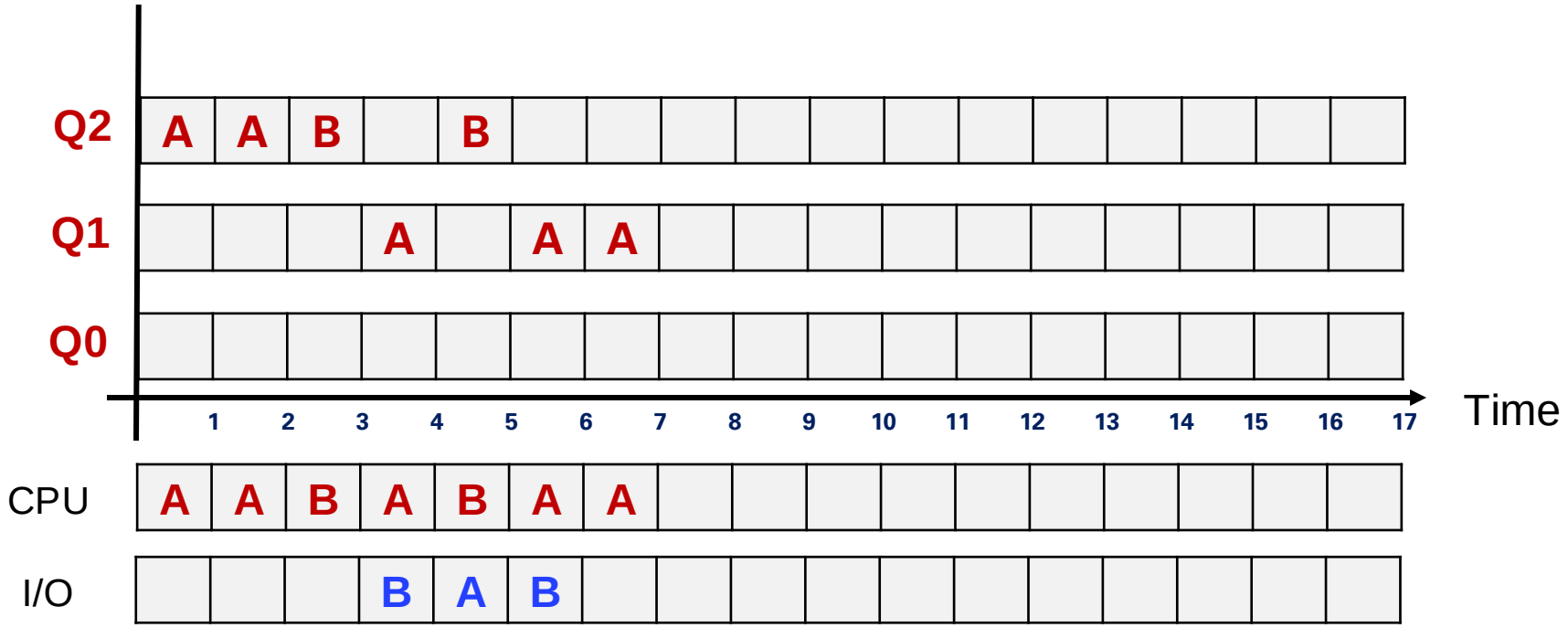


Question 3

Arrive At	Process	Behavior	Priority
0	A	**C3**, IO1, **C3**	**0**
0	B	**C1**, IO1, **C1**, IO1, **C1**	**2**

Time Quantum = 2 TU
Interval Between Interrupt = 1 TU

Time = 7

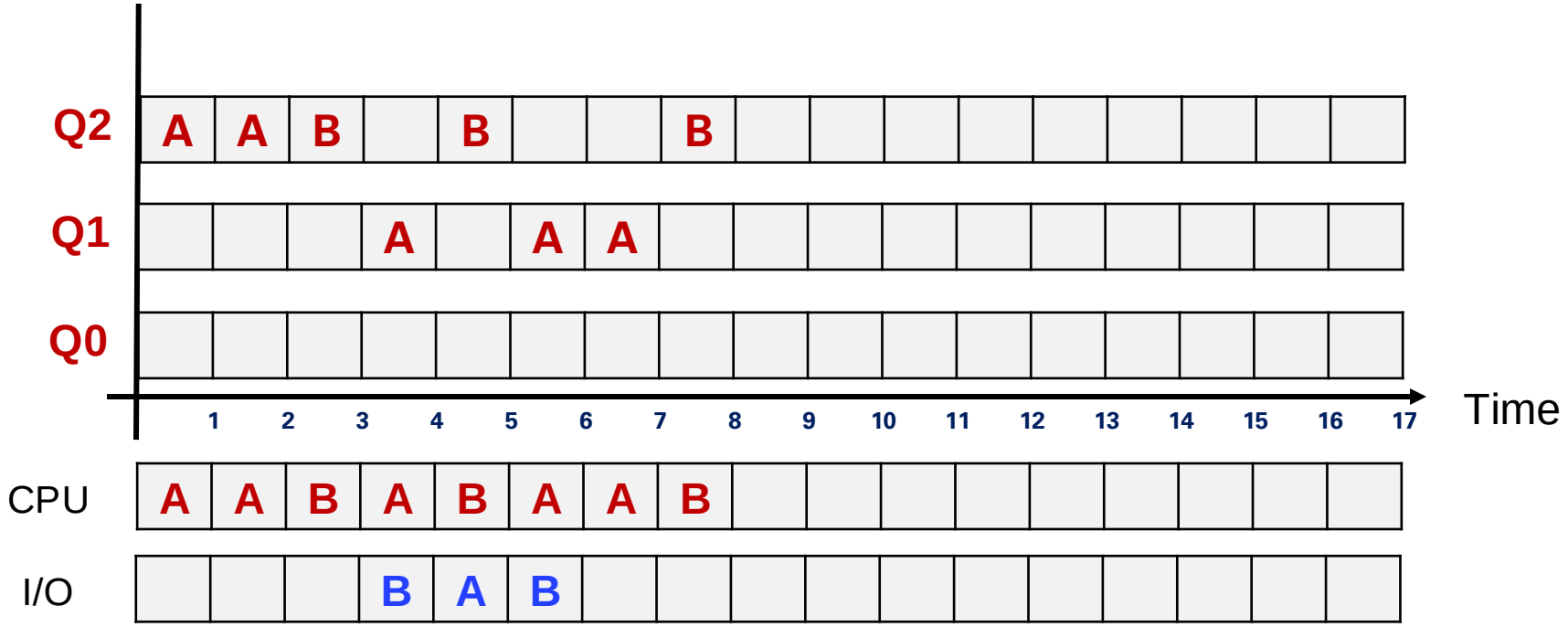


Question 3

Arrive At	Process	Behavior	Priority
0	A	**C3**, IO1, **C3**	**0**
0	B	**C1**, IO1, **C1**, IO1, **C1**	

Time Quantum = 2 TU
Interval Between Interrupt = 1 TU

Time = 8

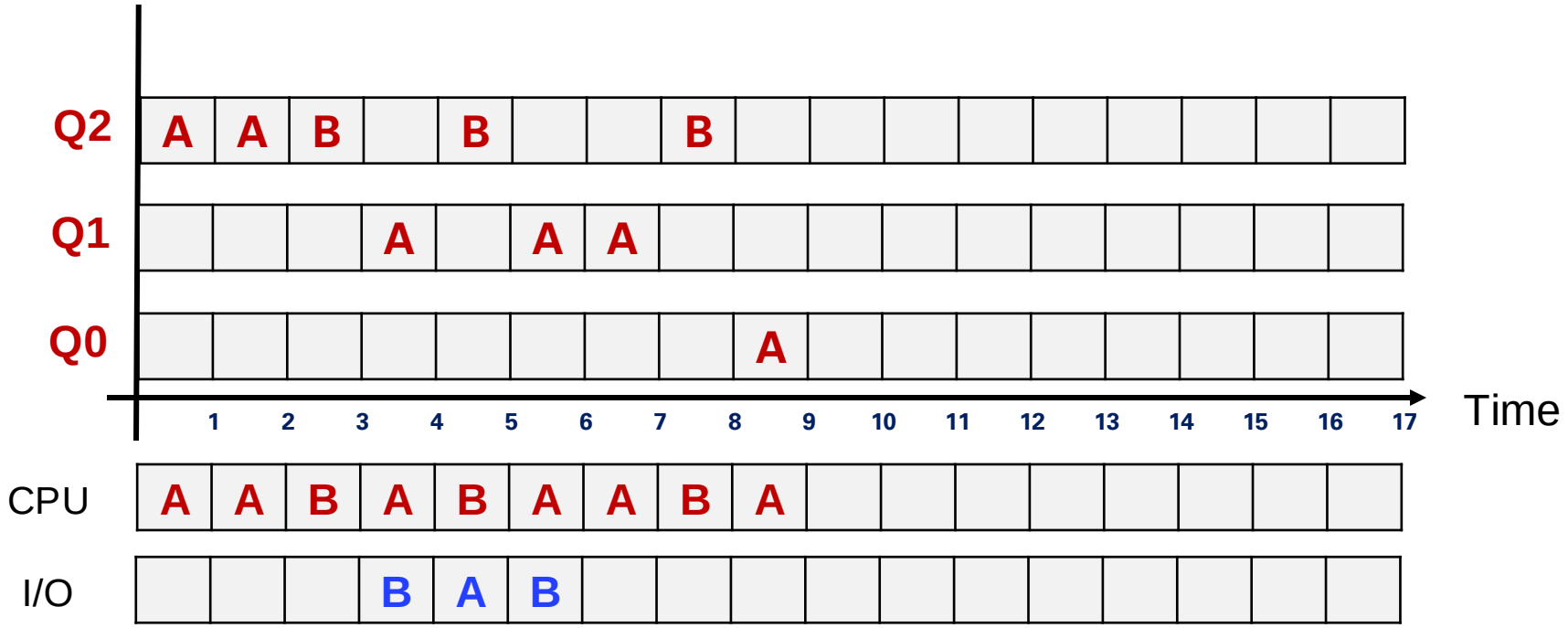


Question 3

Arrive At	Process	Behavior	Priority
0	A	**C3**, IO1, **C3**	
0	B	**C1**, IO1, **C1**, IO1, **C1**	

Time Quantum = 2 TU
Interval Between Interrupt = 1 TU

Time = 9



Question 4

Question 4(a)

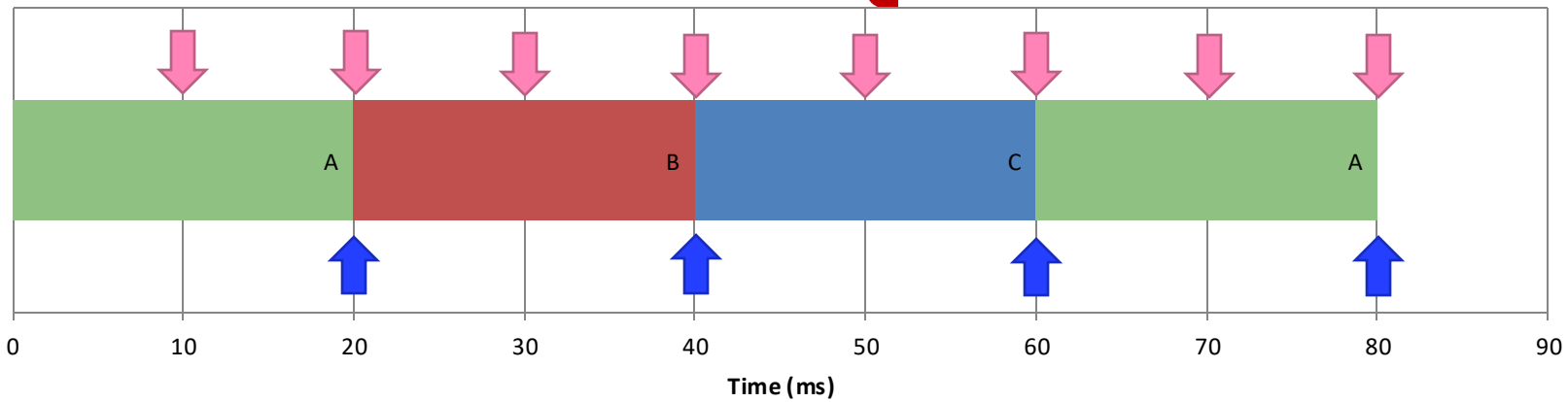
- Give the pseudocode for the Round Robin **scheduler function**.
- For simplicity, you can assume that all tasks are CPU intensive that runs forever (i.e. there is no need to consider the cases where the task blocks / give up CPU).
- Note that this function is invoked by timer interrupt that triggers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

Question 4(a)

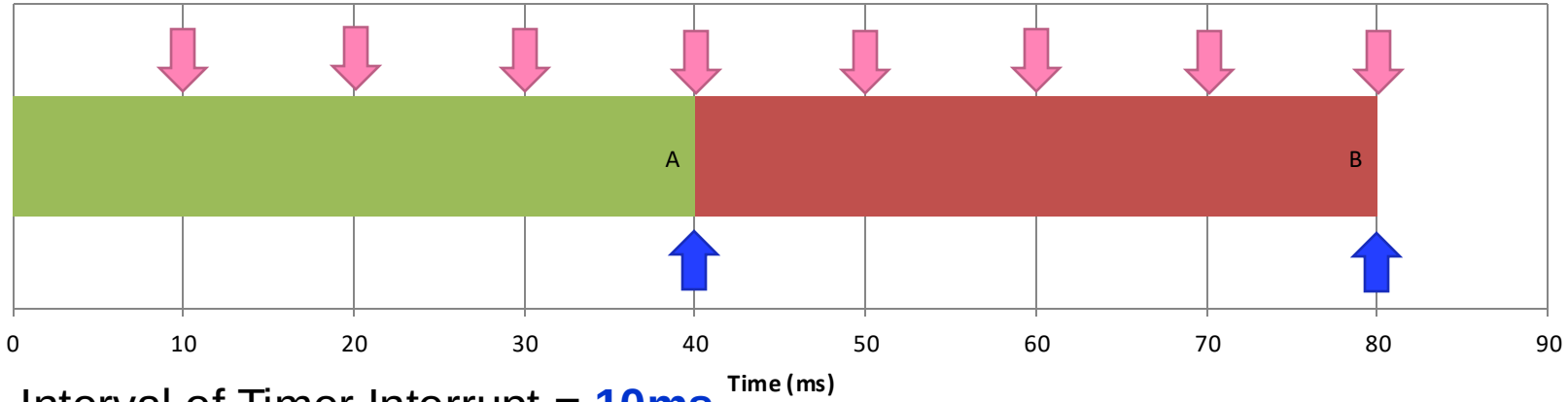
Please use the following variables and function in your pseudocode.

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.

Illustration: ITI vs Time Quantum



Interval of Timer Interrupt = **10ms**
Time Quantum = **20ms**



Interval of Timer Interrupt = **10ms**
Time Quantum = **40ms**

Question 4(a)

```
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 );
```

Question 4(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)?

Question 4(b)

Discuss how do you handle blocking of process on I/O or any other events.

- 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.
- This system call is intercepted by the OS, which places the process in a “blocked” state (in the blocked queue).
- Hence, the handling of I/O is managed by a dedicated system call or interrupt handler, not directly in the above round-robin scheduling loop.