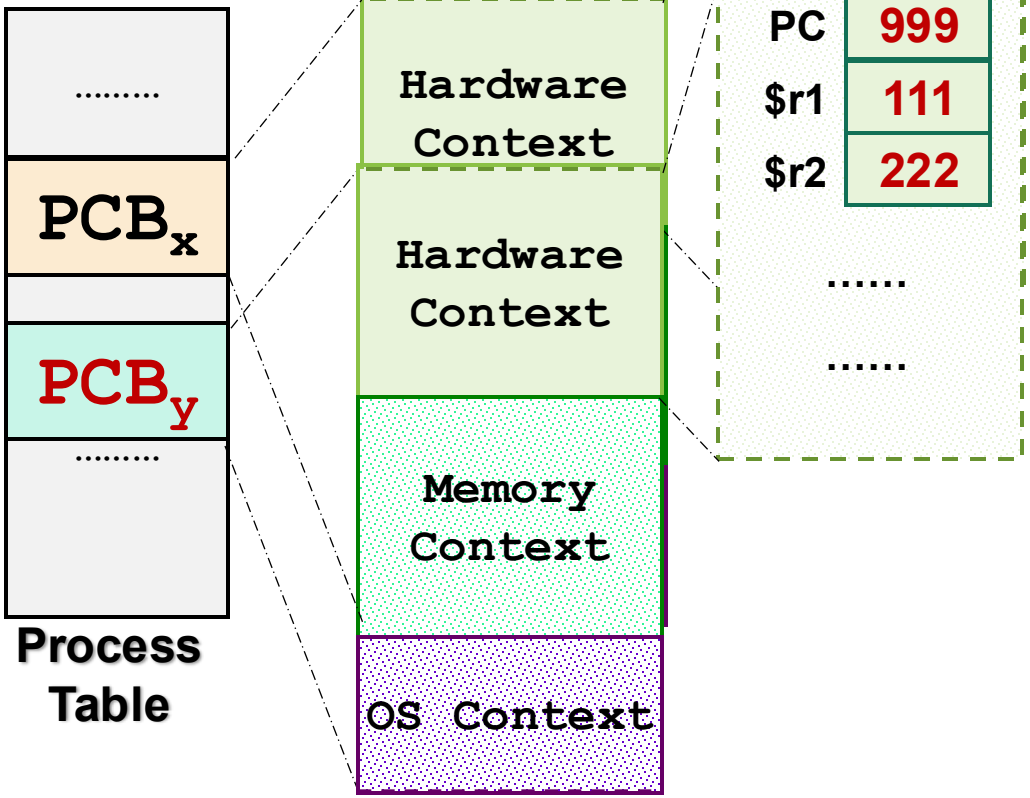
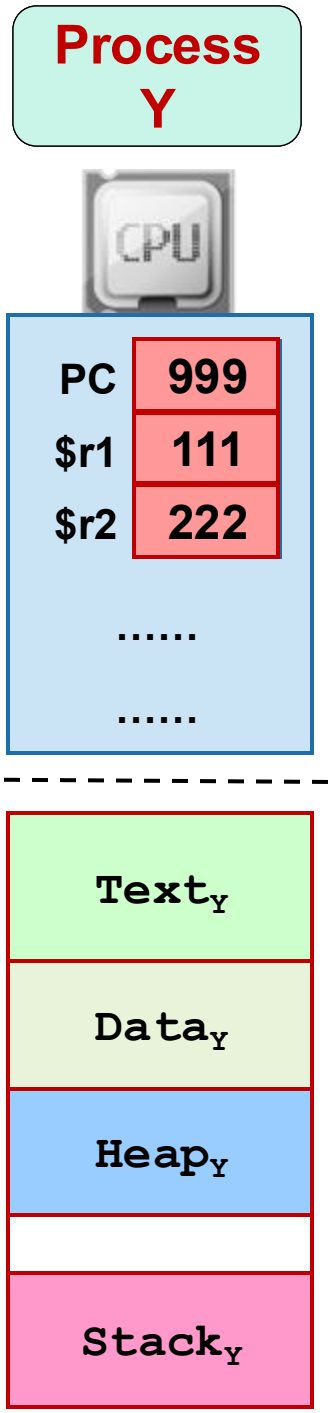
Topic Summary





Allocating new PID

- PIDs are numbered sequentially: the PID of a newly created process is normally the PID of the previously created process increased by one. Of course, there is an upper limit on the PID values
- When the kernel reaches such limit, it must start recycling the lower, unused PIDs. By default, the maximum PID number is 32,767 (PID_MAX_DEFAULT-1); the system administrator may reduce this limit by writing a smaller value into the */proc/sys/kernel/pid_max* file. In 64-bit architectures, the system administrator can enlarge the maximum PID number up to 4,194,303.
- When recycling PID numbers, the kernel must manage a pidmap_array bitmap that denotes which are the PIDs currently assigned and which are the free ones. Because a page frame contains 32,768 bits, in 32-bit architectures the pidmap_array bitmap is stored in a single page. In 64-bit architectures, however, additional pages can be added to the bitmap when the kernel assigns a PID number too large for the current bitmap size. These pages are never released.
- For details Linux implementation, refer to this link: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/pid.c?h=v6.14-rc2

Tutorial 3

Question 1

Interleaving execution?

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:

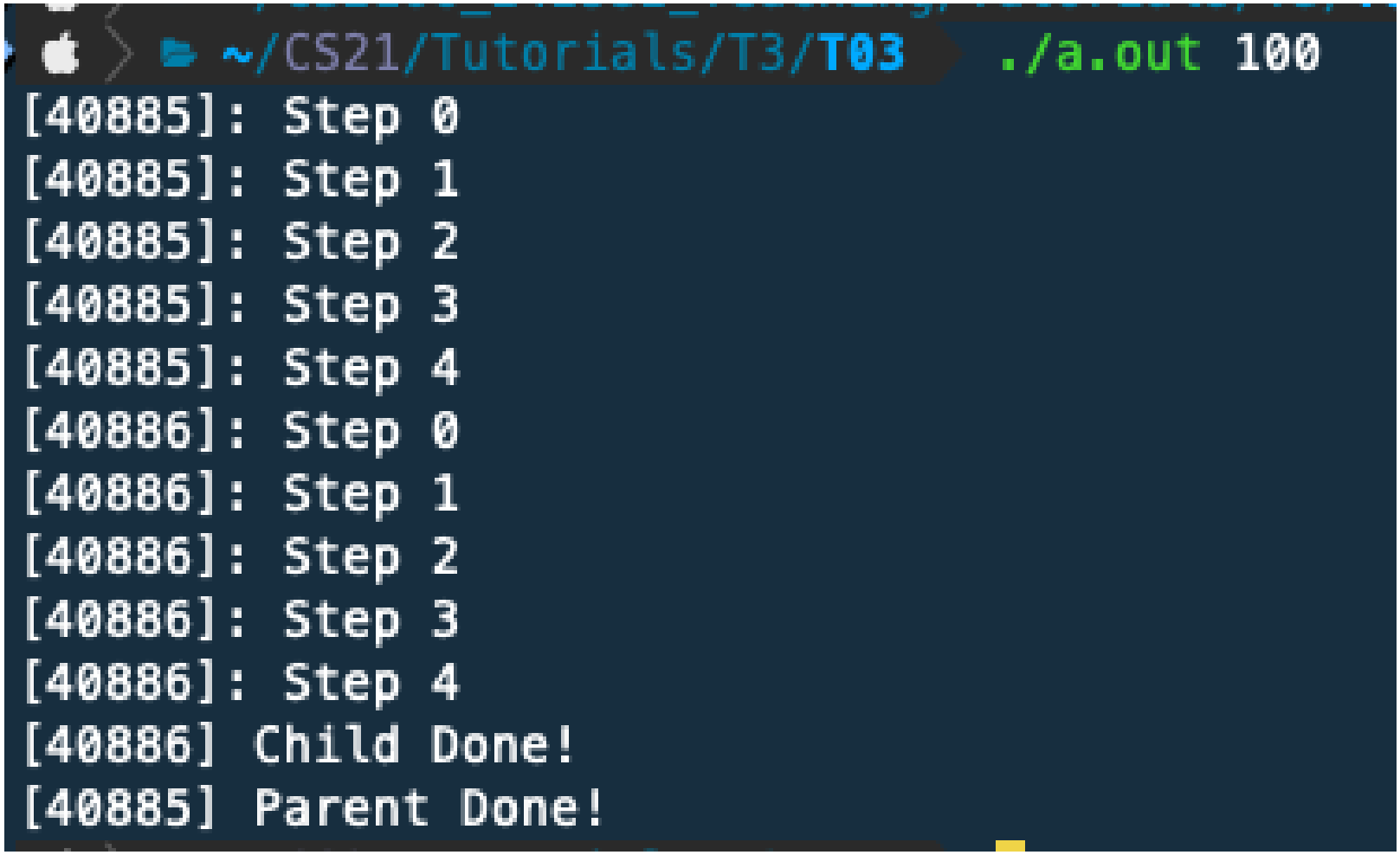
Interleaving Output Pattern
[6183]: Step 0 **[6184]: Step 0** [6183]: Step 1 [6184]: Step 1 **[6183]: Step 2** **[6184]: Step 2** [6183]: Step 3Q [6184]: Step 3 **[6183]: Step 4** **[6184]: Step 4** [6184] Child Done! [6183] Parent Done!

What do you think "**D**" represents?

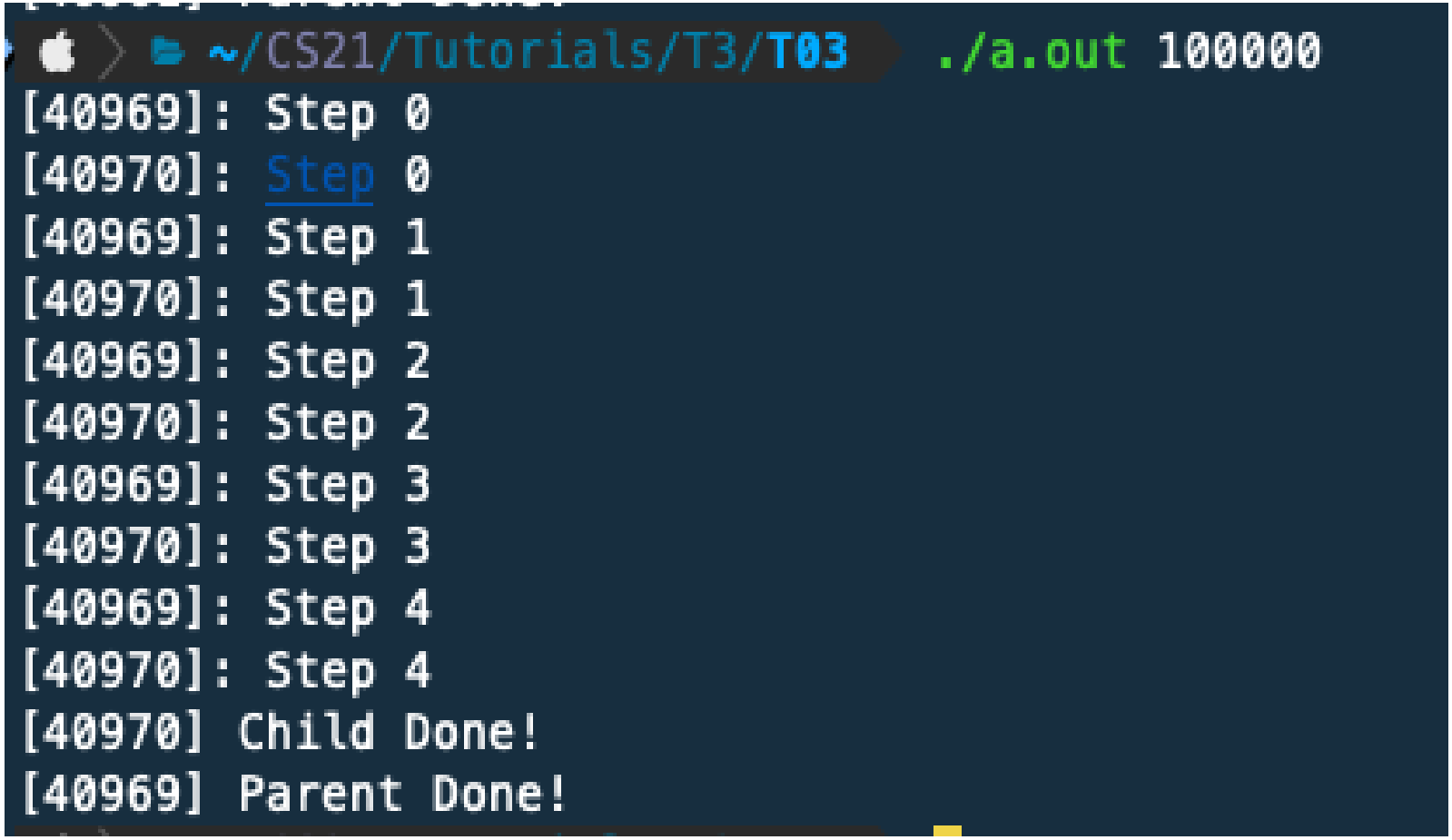
Interleaving execution?

```
7   void DoWork(int iterations, int delay)
8   {
9       int i, j;
10
11      int x = 1;
12
13      for (i = 0; i < iterations; i++){
14          printf("[%d]: Step %d\n", getpid(), i);
15          for (j = 0; j < delay; j++); //introduce some fictional work
16      }
17
18      // printf("[%d] Doing reallllll work!\n", getpid());
19
20  }
```

Interleaving execution?



Interleaving execution?



Tutorial 3

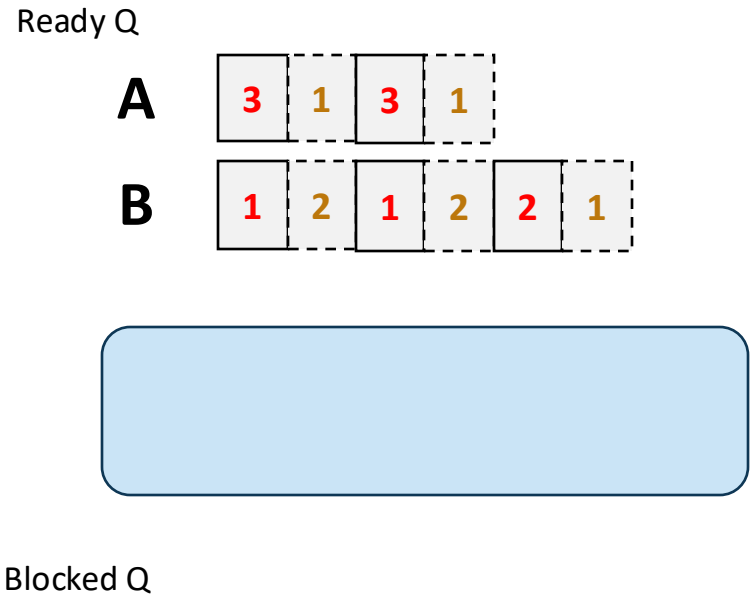
Question 2

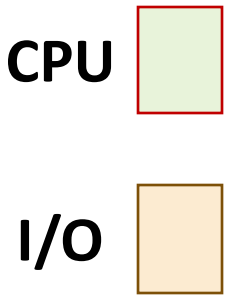
First Come First Serve

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

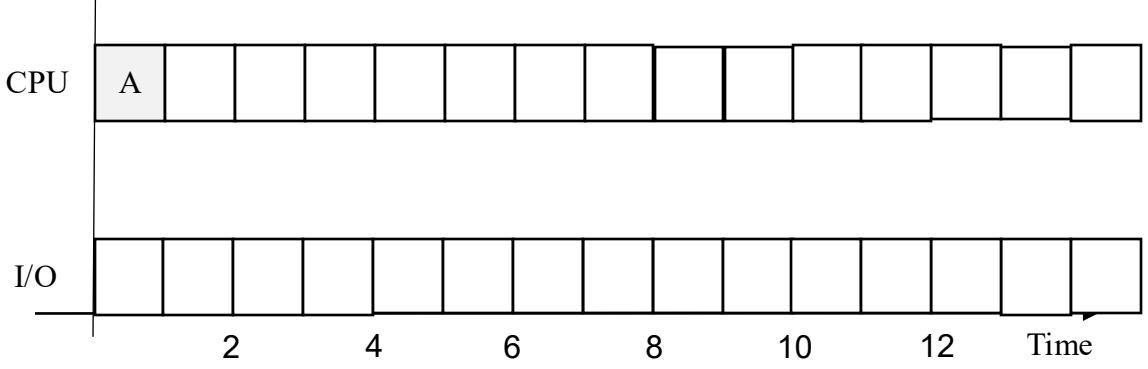
➔Show the result of the scheduling for the time step after scheduler finished its job

Initialization





Time: 1



Ready Q

B | 1 | 2 | 1 | 2 | 2 | 1 |

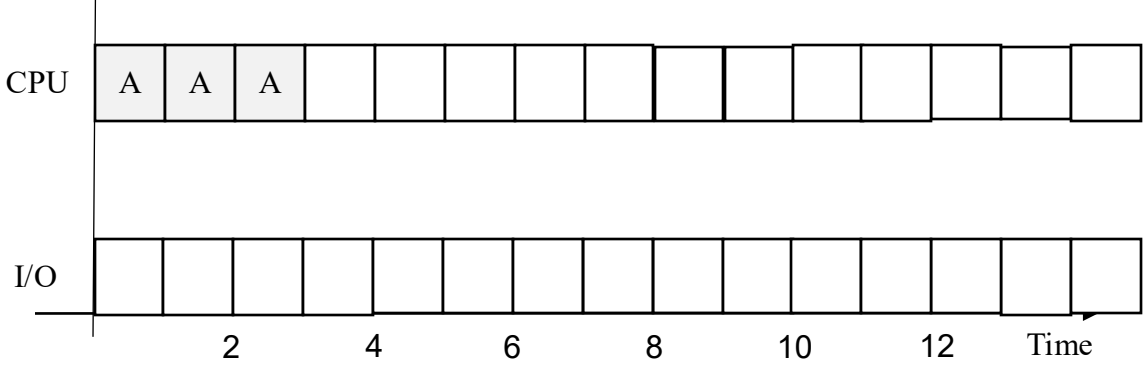
A is chosen as it is the first in Q

Blocked Q

CPU **A** | 3 | 1 | 3 | 1 |

I/O

Time: 3



Ready Q

B 1 2 2 1

C 2

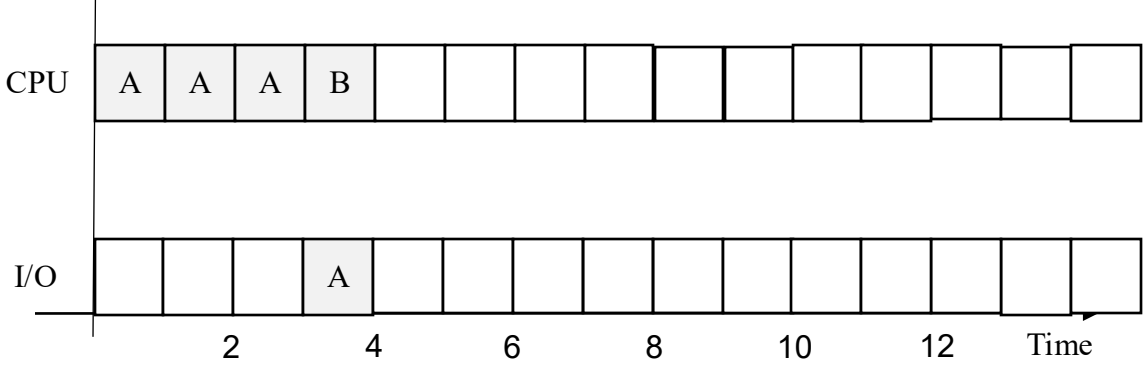
A still working, C arrives

Blocked Q

CPU **A** 1 1 3 1

I/O

Time: 4



Ready Q

C 2

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

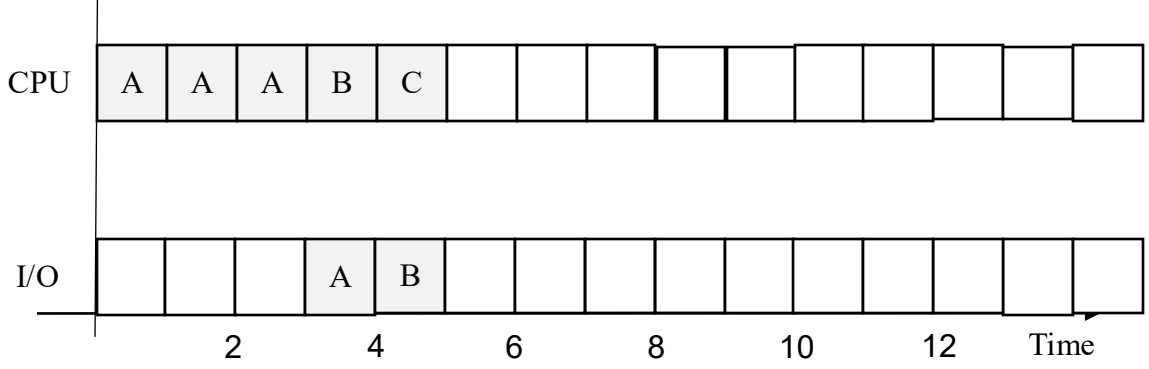
I/O **A**

A blocks, B get CPU

Blocked Q

A 1 3 1

Time: 5



Ready Q

A 3 1

CPU **C** 2

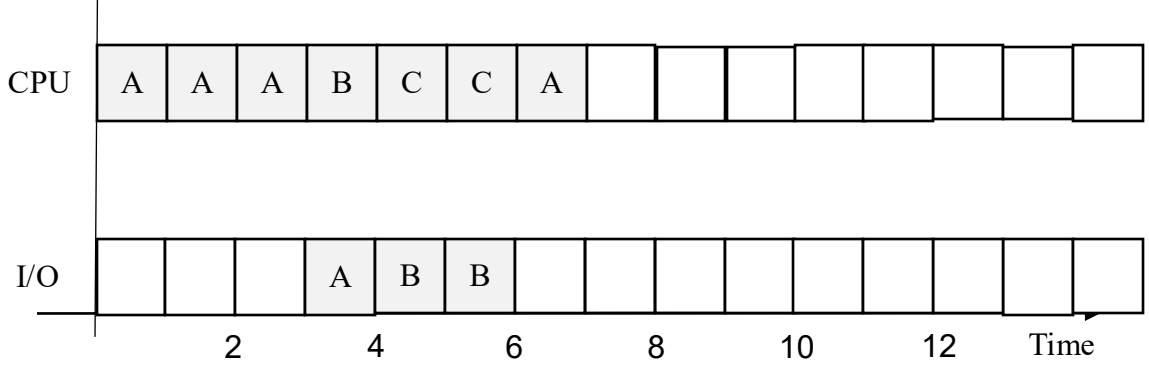
I/O **B**

B blocks, C get CPU

Blocked Q

B 2 1 2 2 1

Time: 7



Ready Q

B 1 2 2 1

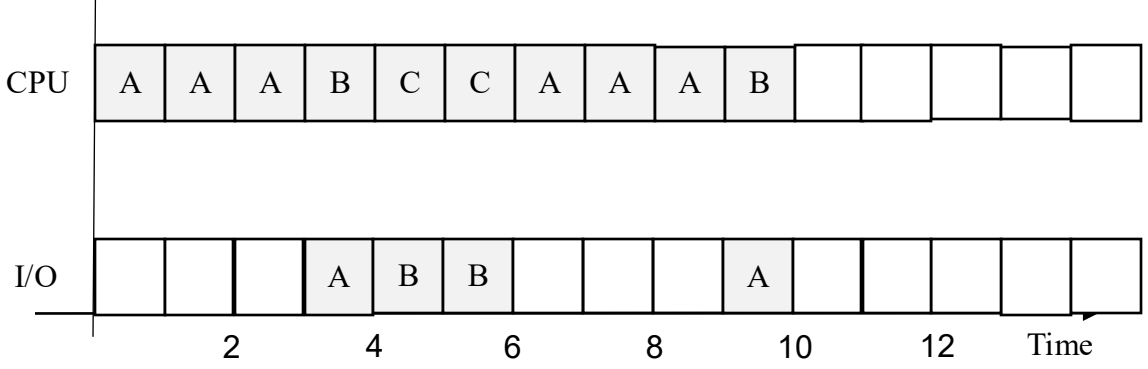
CPU **A** 3 1

I/O

C done; A get CPU; B unblocks

Blocked Q

Time: 10



Ready Q

B 1 2 2 1

CPU **B** 1 2 2 1

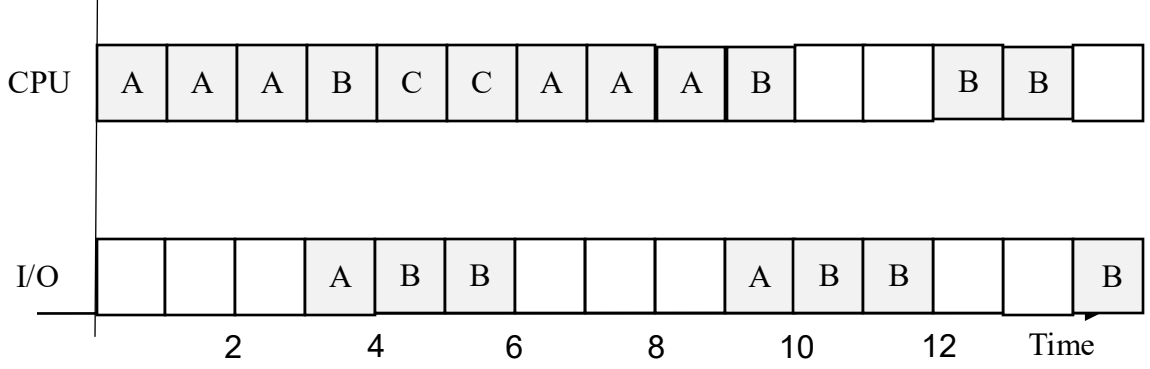
I/O **A**

A blocks; B get CPU

Blocked Q

A 1

Time: 15



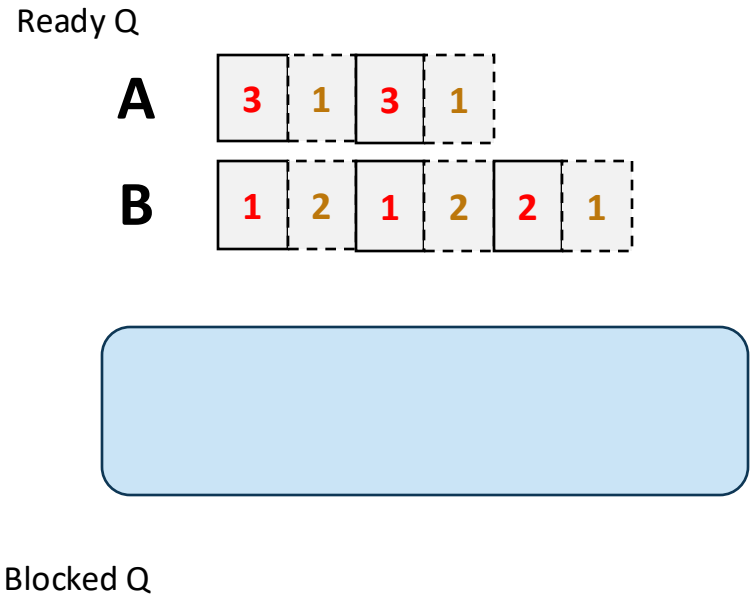
	Turnaround Time	Waiting Time
A	10	10 – 8 = 2
B	15	15 – 9 = 6
C	6 - 3 = 3	3 – 2 = 1

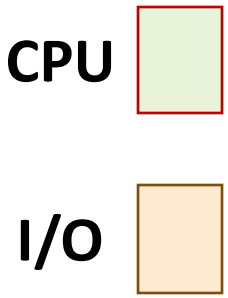
Round Robin

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

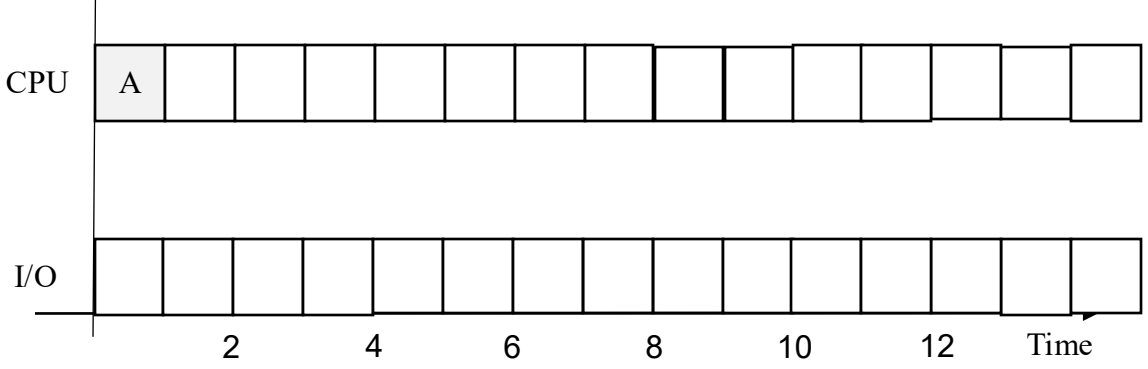
➔Show the result of the scheduling for the time step after scheduler finished its job

Initialization





Time: 1



Ready Q

B | 1 | 2 | 1 | 2 | 2 | 1 |

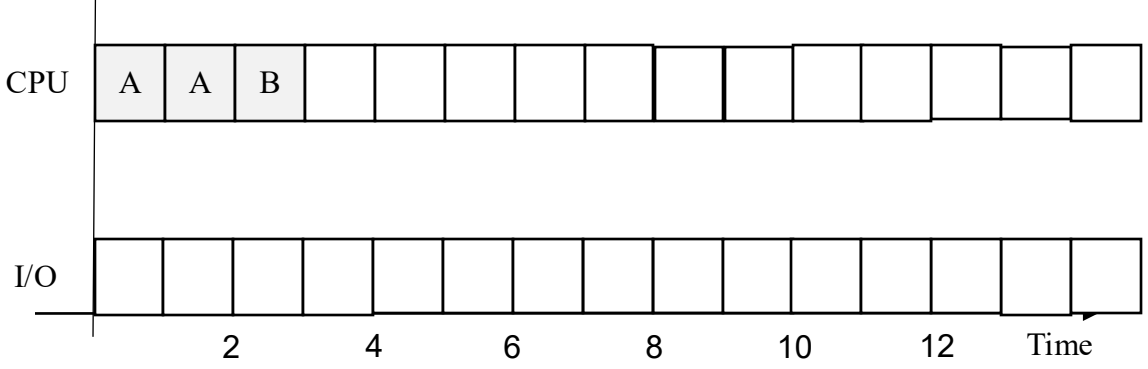
A is chosen as it is the first in Q

Blocked Q

CPU **A** | 3 | 1 | 3 | 1 |

I/O

Time: 3



Ready Q

A 1 1 3 1

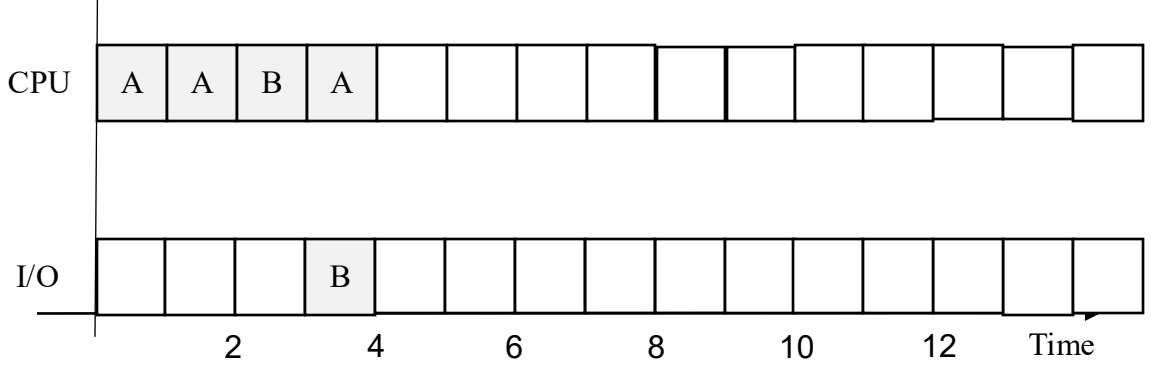
CPU **B** 1 2 1 2 2 1

I/O

A is removed from CPU after 2 TUs.

Blocked Q

Time: 4



Ready Q

C 2

CPU **A** 1 1 3 1

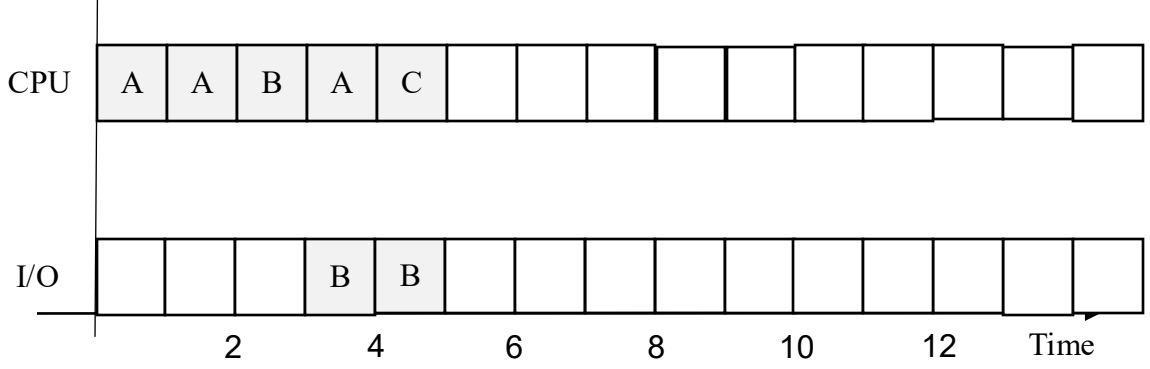
I/O **B**

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

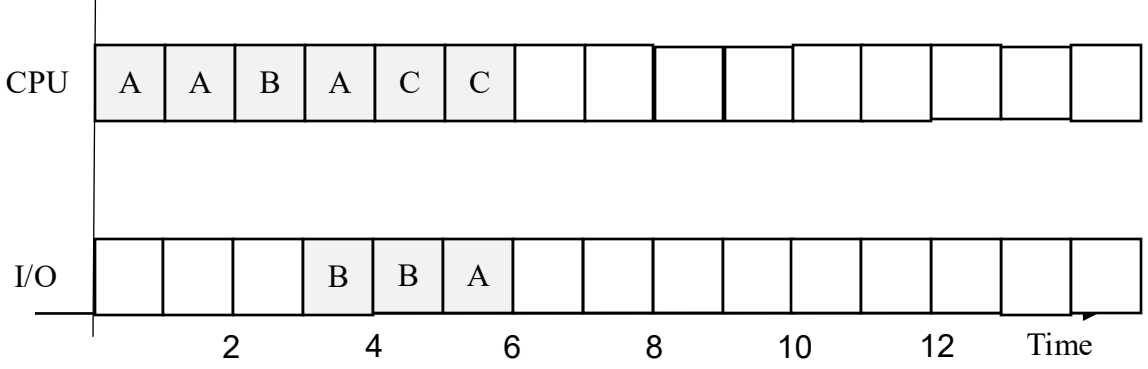
C is the only ready process

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

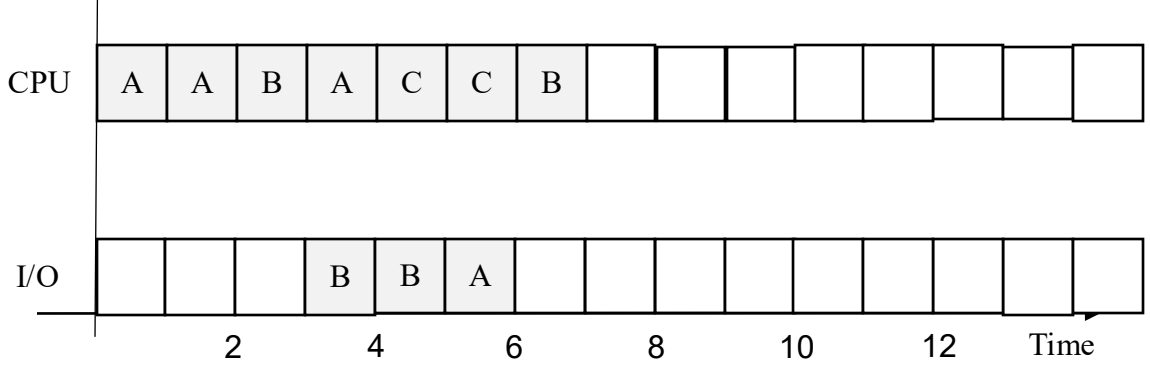
Blocked Q

A 1 3 1

CPU C 1

I/O A

Time: 7



Ready Q

A 3 1

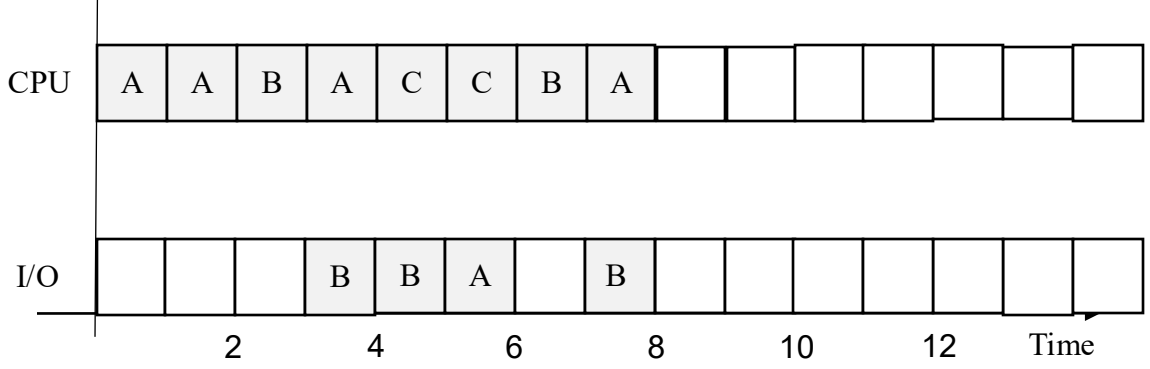
CPU **B** 1 2 2 1

I/O

B is chosen. A unblocks.

Blocked Q

Time: 8



Ready Q

CPU **A** 3 1

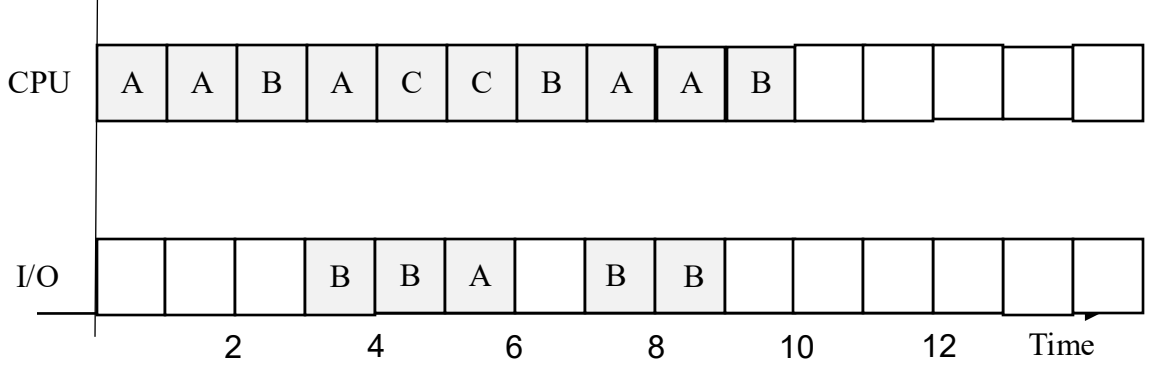
I/O **B**

B blocks.

Blocked Q

B 2 2 1

Time: 10



Ready Q

A 1 1

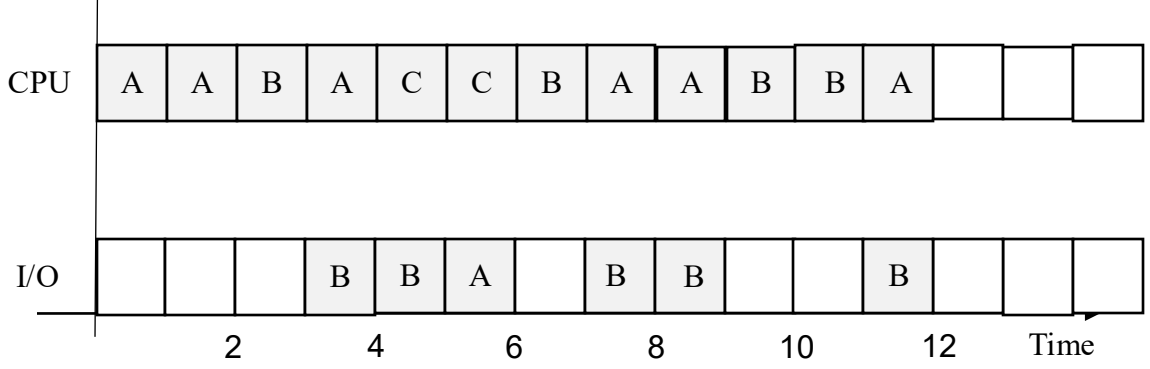
CPU **B** 2 1

I/O

A vacates CPU. B unblocks & get chosen.

Blocked Q

Time: 12



Ready Q

CPU **A** 1 1

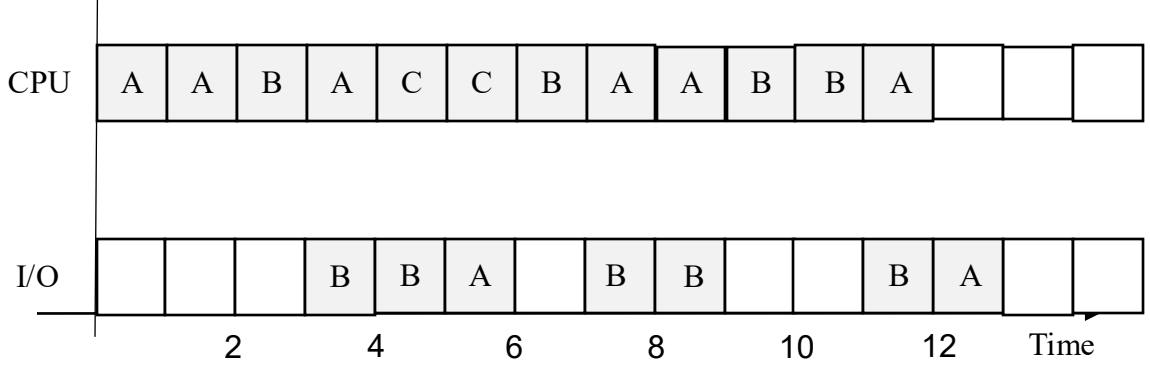
I/O **B**

B blocks.

Blocked Q

B 1

Time: 13



Ready Q

CPU

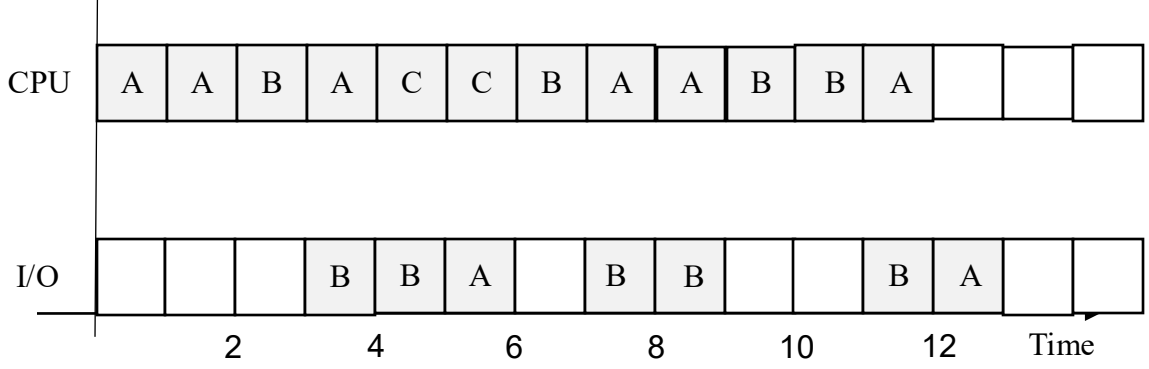
I/O **A**

B done; A blocks

Blocked Q

A 1

Time: 14



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

Tutorial 3

Question 3

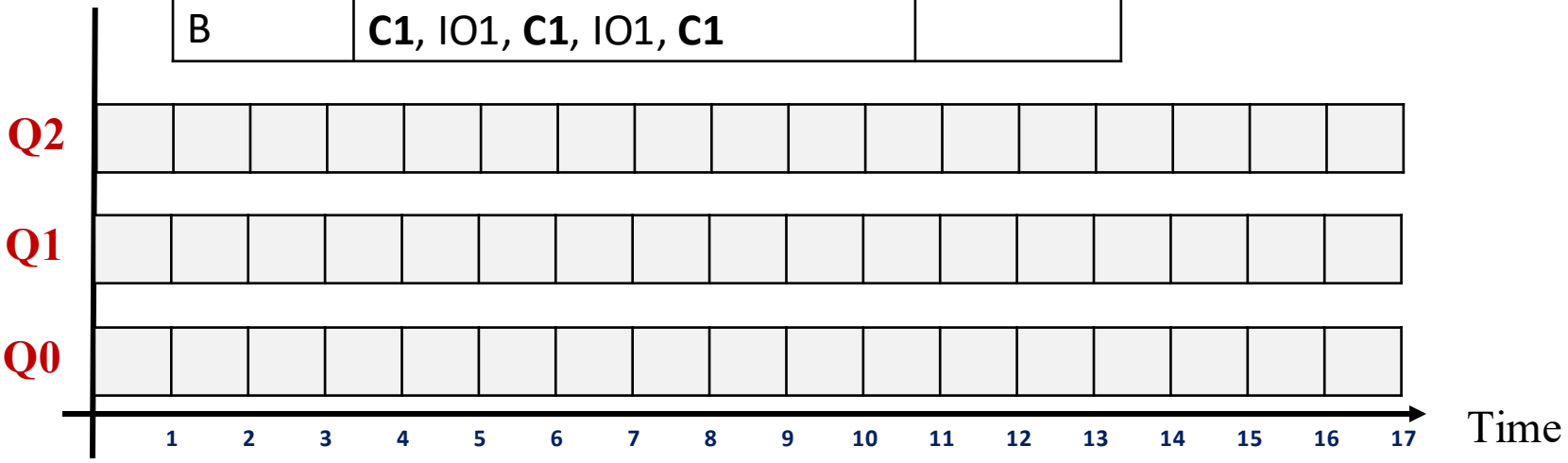
MLFQ: Rules?

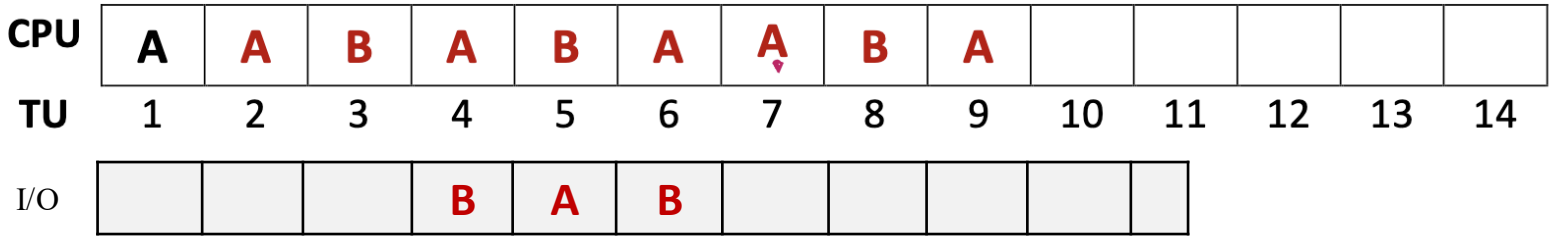
- **Basic rules:**
  1. If Priority(A) > Priority(B) ➔
  2. If Priority(A) == Priority(B) ➔

- **Priority Setting/Changing rules:**
  1. New job ➔
  2. If a job fully utilized its time slice ➔
  3. If a job give up / blocks ➔

Q3. MLFQ (TQ=2, ITI=1)

Process	Behavior	Priority
A	**C3**, IO1, **C3**	
B	**C1**, IO1, **C1**, IO1, **C1**	





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.

Tutorial 3

Question 4

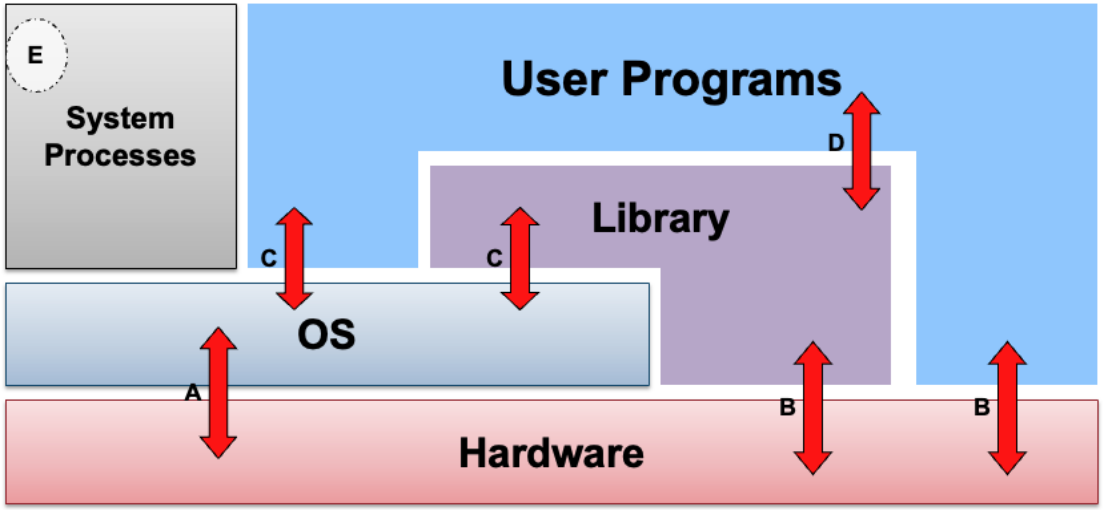
Pseudo-Code for RR Scheduler

```
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 );
```

Pseudo-Code for RR Scheduler



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.