Process Management

Process Scheduling

Lecture 4

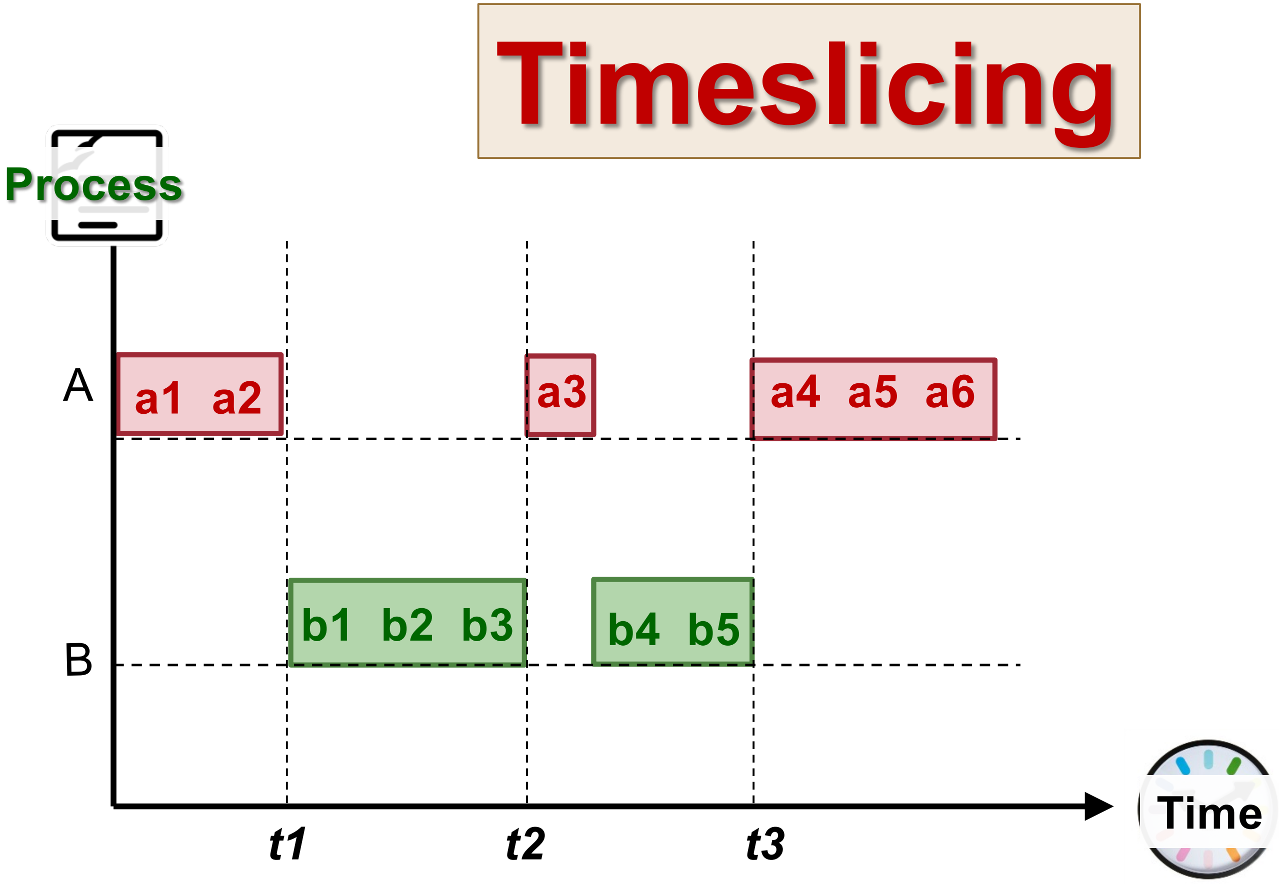
Overview

- **Concurrent Execution**
- **Process Scheduling**
  - Definition
  - Process behavior
  - Processing environment
  - Criteria for good scheduling
  - Procedure of process scheduling
- **Scheduling Algorithms**
  - For Batch Processing System
  - For Interactive System

Concurrent Execution

- Processes P1 and P2 execute **concurrently** if they both make progress within a short window of time (i.e., they run "in parallel")
  - Could be **virtual parallelism**
    - Illusion of parallelism, pseudo-parallelism
  - Could be **physical parallelism**
    - Multiple execution units (cores, CPUs, …) available in hardware
- You can assume the two forms of parallelisms are not distinguished in the following discussion

Concurrency Example (Simplistic)



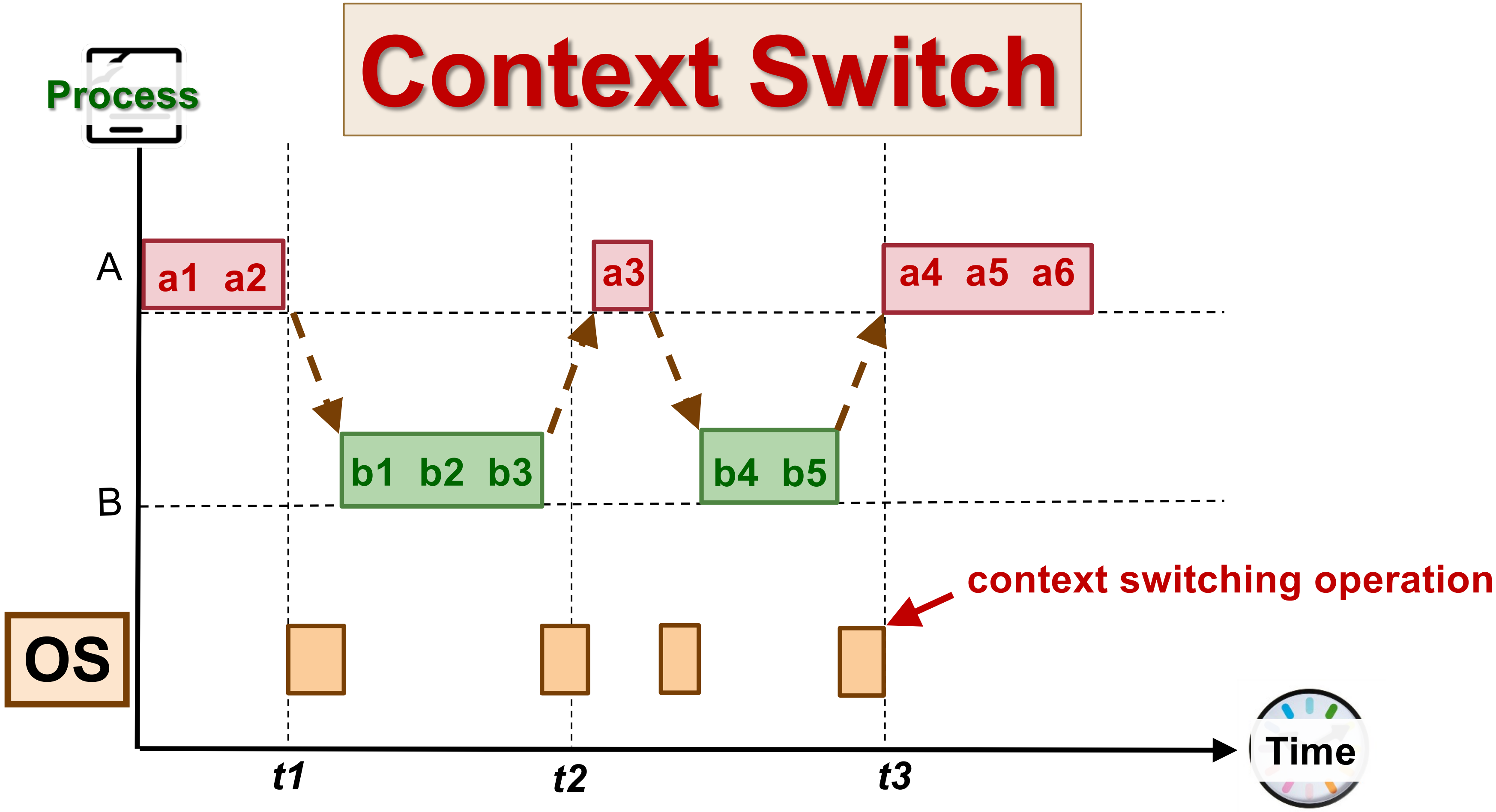
a1-a6:
Instructions of **process A**

b1-b5:
Instructions of **process B**

Concurrent execution on 1 CPU: interleaving of instructions from both processes

- Also called **time-slicing**

Interleaved Execution (context switch)

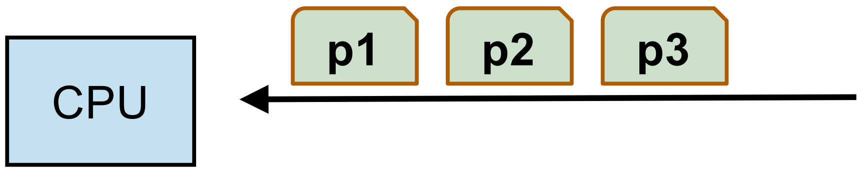


Multitasking requires switching contexts between A and B

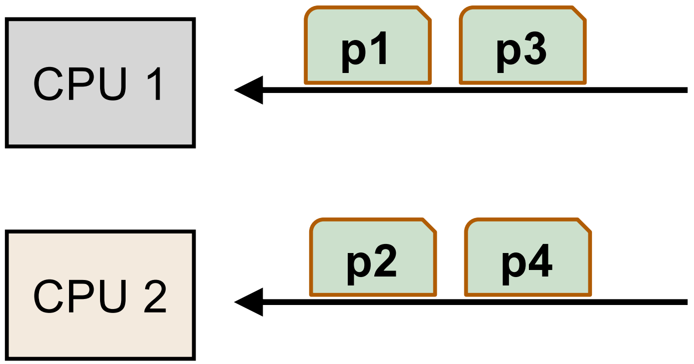
- OS incurs some overhead in the switching processes

Multitasking OS

- 1 CPU: time-sliced execution of tasks



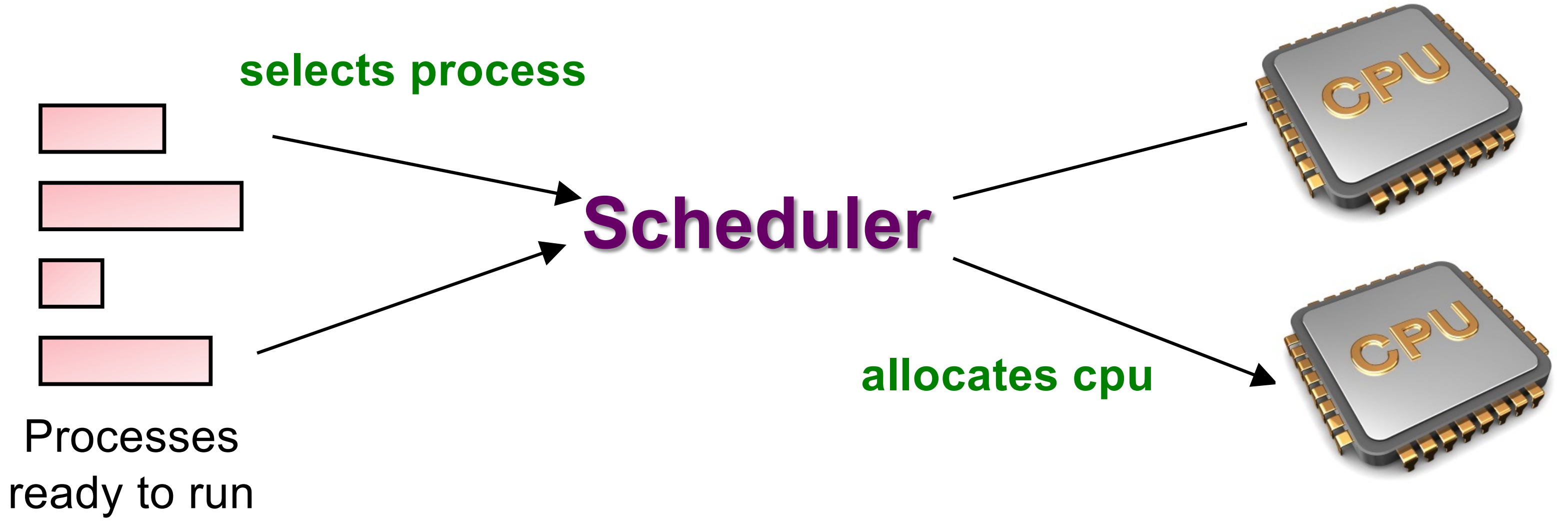
- Multi-processor: time-slicing on ***n*** CPUs



Scheduling in OS: A definition

- Problems with having multiple processes:
  - If [***#ready-to-run processes***] > [***#available_CPUs***], which process should be chosen?
  - Known as the **scheduling problem**
- Terminology:
  - **Scheduler**
    - the OS component that makes scheduling decisions
  - **Scheduling algorithm**
    - The algorithm used by scheduler

Scheduling: Illustration



- Each process requires different amount of CPU time
  - **Process behavior**
- Multiple ways to allocate CPU time
  - Defined by **scheduling algorithms**
  - Influenced by the **processing environment** (batch, interactive, real-time…)
- A number of **criteria to evaluate schedulers**

Process Behavior

- A typical process goes through phases of:

CPU-Activity:

- Computation
- E.g., number crunching
- **Compute-Bound Process** spends majority of its time here

IO-Activity:

- Requesting and receiving service from I/O devices
- E.g., Print to screen, read from file, wait for a key
- **IO-Bound Process** spends majority of its time here

3 Types of Processing Environment

1. **Batch Processing:**
   - Usually long-running without user intervention
   - No user → No interaction required → No need to be responsive
     - E.g., training deep neural networks on a supercomputer

2. **Interactive**:
   - With active user(s) interacting with system (e.g., MS Word)
   - Should be responsive, consistent in response time

3. **Real-time processing**:
   - Have a strict deadline to meet (e.g., aircraft controller)
   - Tasks are usually periodic

Criteria for Scheduling Algorithms

Many criteria to evaluate a scheduling algorithm

- Different processing environment have different criteria
- Some criteria may be mutually conflicting

Criteria for **all processing environments**:

- **Fairness**:
  - Ensuring fair sharing of CPU time
    - on a per-process basis OR
    - on a per-user basis
  - Also mean **no starvation**
- **Balanced Utilization of the System Resources:**
  - All components of the computing system should be utilized
    - In a balanced manner, without bottlenecks

When to perform scheduling?

Two types of scheduling policies:

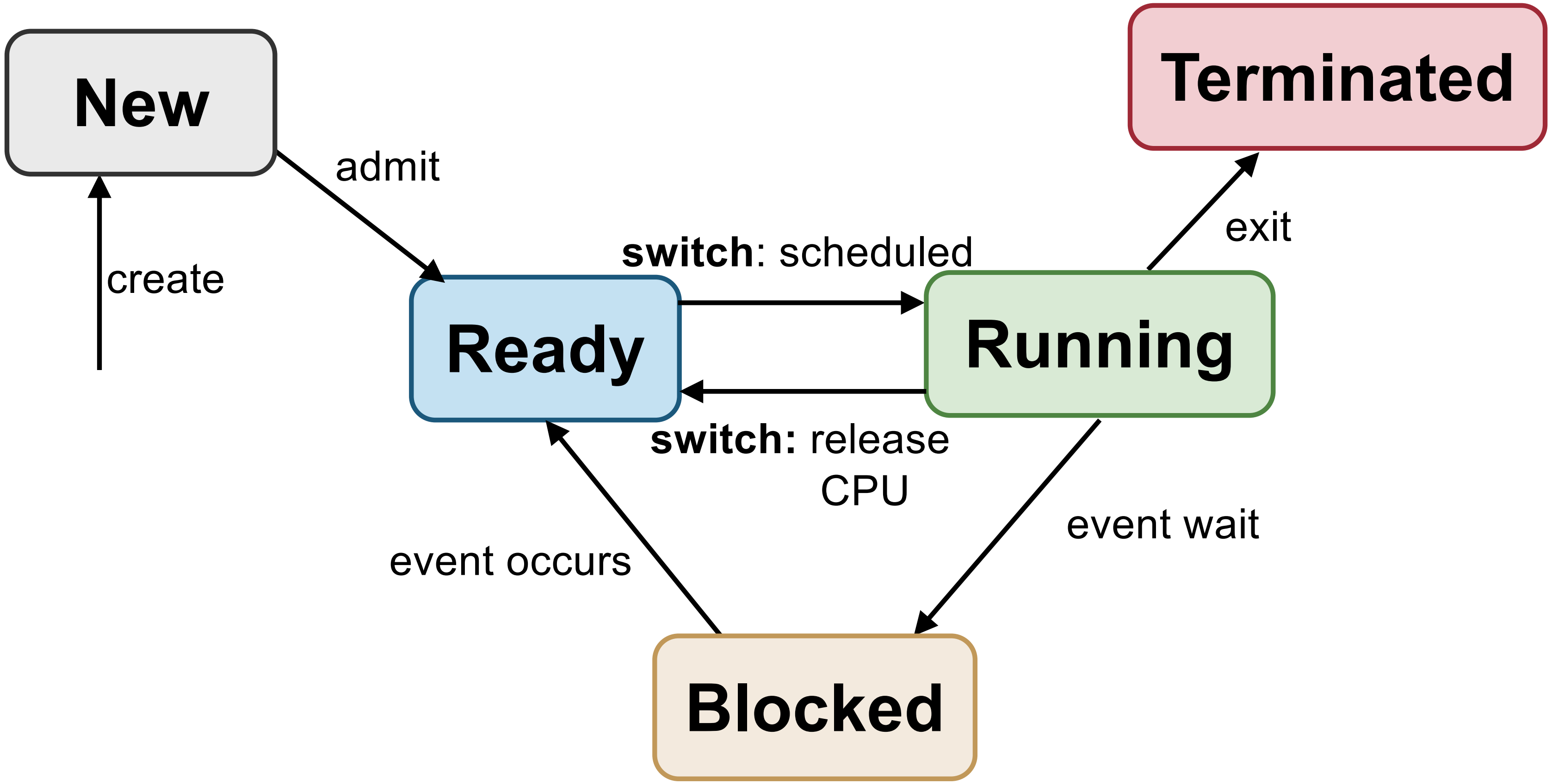
- Defined by **when** scheduling is triggered

- **Non-preemptive** (a.k.a. *cooperative*)
  - A process stayed scheduled (in **RUNNING** state) until:
    1. it blocks, OR (going to the **BLOCKED** state)
    2. gives up the CPU voluntarily (going to the **READY** state)

- **Preemptive**
  - CPU can be taken (*preempted*) from the running process at **ANY** time

  Typically, a process is given a **time quota** to run:
  - At the end of the **time quota** the process is suspended (goes to **READY** state)
    - Another process gets picked if available
  - It is possible for a process to block or finish/give up CPU early

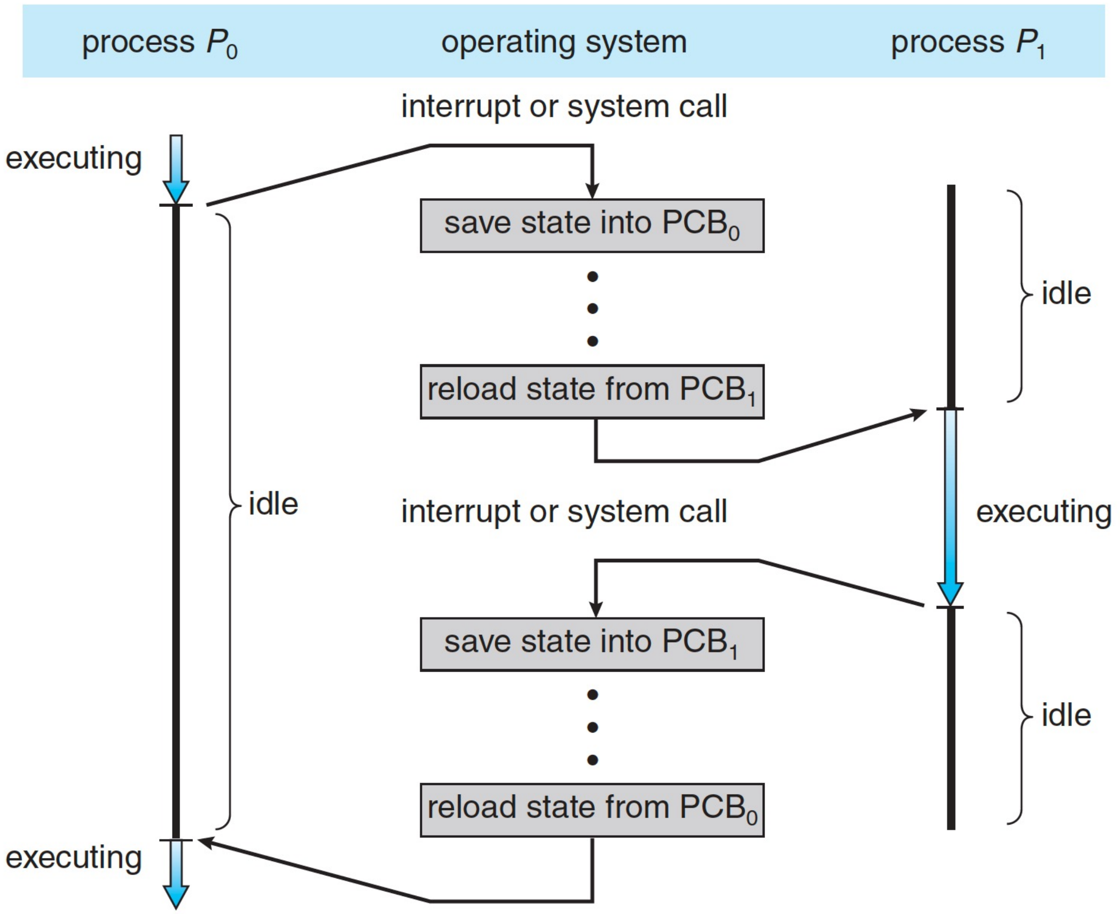
Generic 5-State Process Model



Scheduling a Process: Step-by-Step

1. - Scheduler is triggered (**OS takes over**)
2. - **If Context switch is needed:**
     - Context of current running process is saved and placed on blocked queue / ready queue
3. - Pick a suitable process **P** to run based on scheduling algorithm
4. - Setup the context for **P**
5. - Let process **P** run

Scheduling a Process: Step By Step



SCHEDULING FOR BATCH PROCESSING

Overview – Batch Processing

- On batch processing system:
  - No user interaction
  - Non-preemptive scheduling is predominant

- Scheduling algorithms are generally easier to understand and implement
  - There are variants/improvements are specialized for certain use cases

- Three algorithms covered:
  - **F**irst-**C**ome **F**irst **S**erved (**FCFS**)
  - **S**hortest **J**ob **F**irst (**SJF**)
  - **S**hortest **R**emaining **T**ime Next (**SRT**)

Criteria for batch processing

- **Throughput**:
  - Number of tasks finished per unit time (rate of task completion)
- **Turnaround time**:
  - Total wall clock time taken, i.e., ***finish_time – start_time***
  - Related to **waiting time**: time spent waiting for CPU
  - Average turnaround time matters more than individual
- **CPU utilization**:
  - Percentage of time when CPU is working on a task (as opposed to being idle)

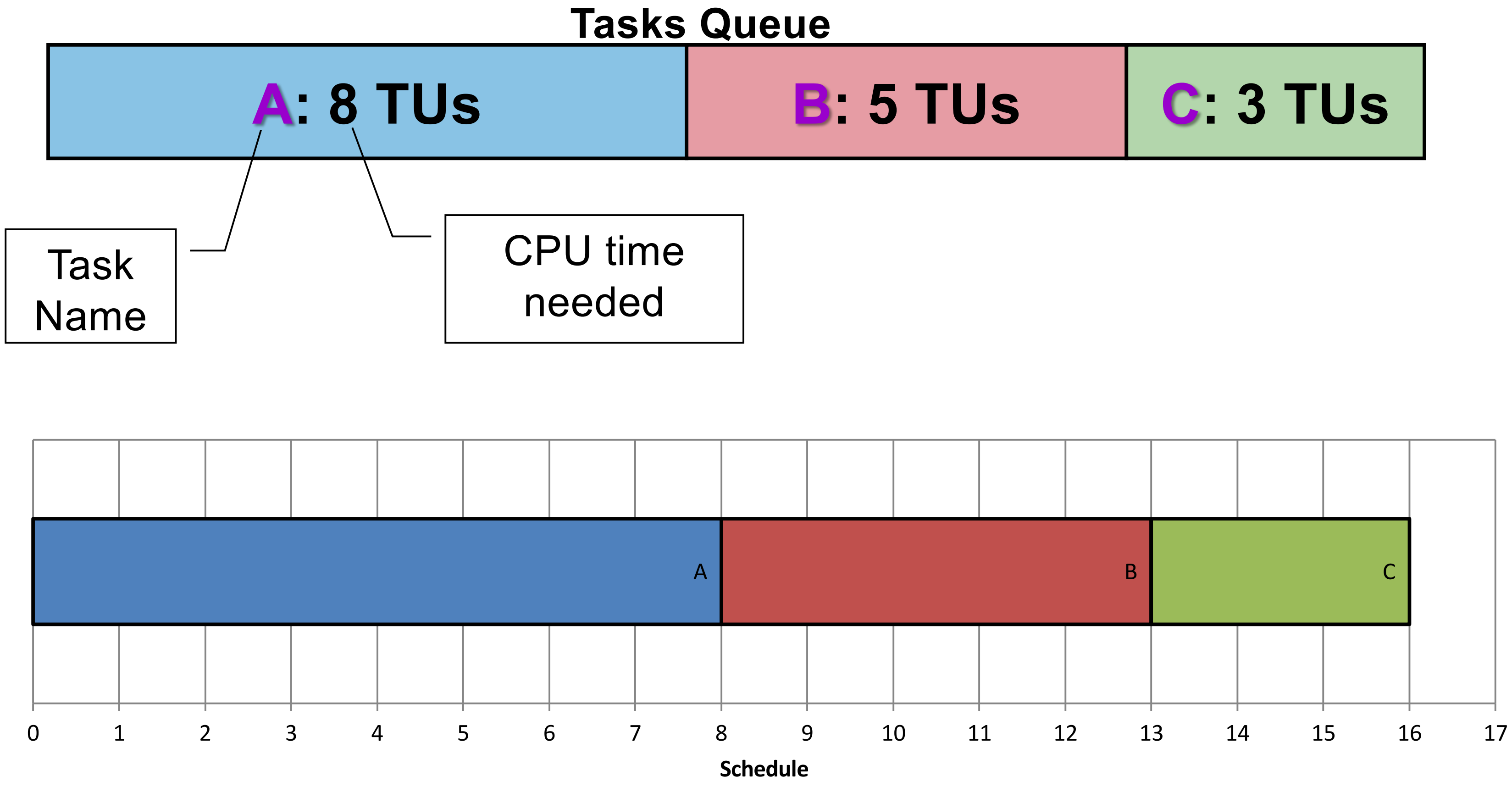
First-Come First-Served: **FCFS**

General Idea:

- Tasks are stored on a First-In-First-Out (FIFO) queue
  - based on arrival time
- Pick the first task in queue to run until:
  - The task is done OR the task is blocked
  - Blocked task is removed from the FIFO queue
    - When it is ready again, it is placed at the back of queue
      - Just like a newly arrived task
- Guaranteed to have **no starvation**:
  - The number of tasks in front of task X in FIFO is always decreasing

➔ task X will get its chance eventually

First-Come First-Served: Illustration



The average total waiting time for 3 tasks: (0 + 8 + 13)/3 = 7 time units

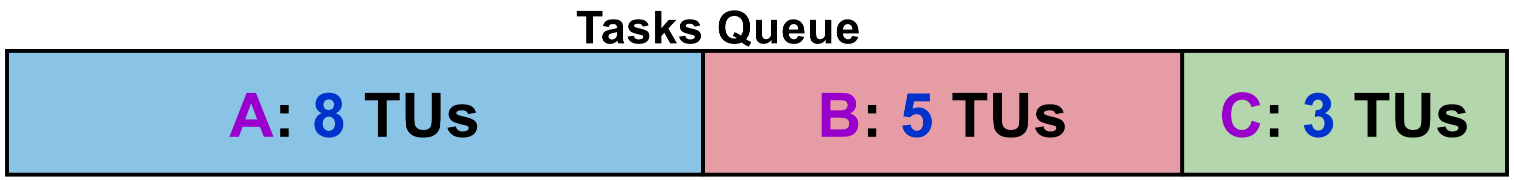
First-Come First-Served: Shortcomings

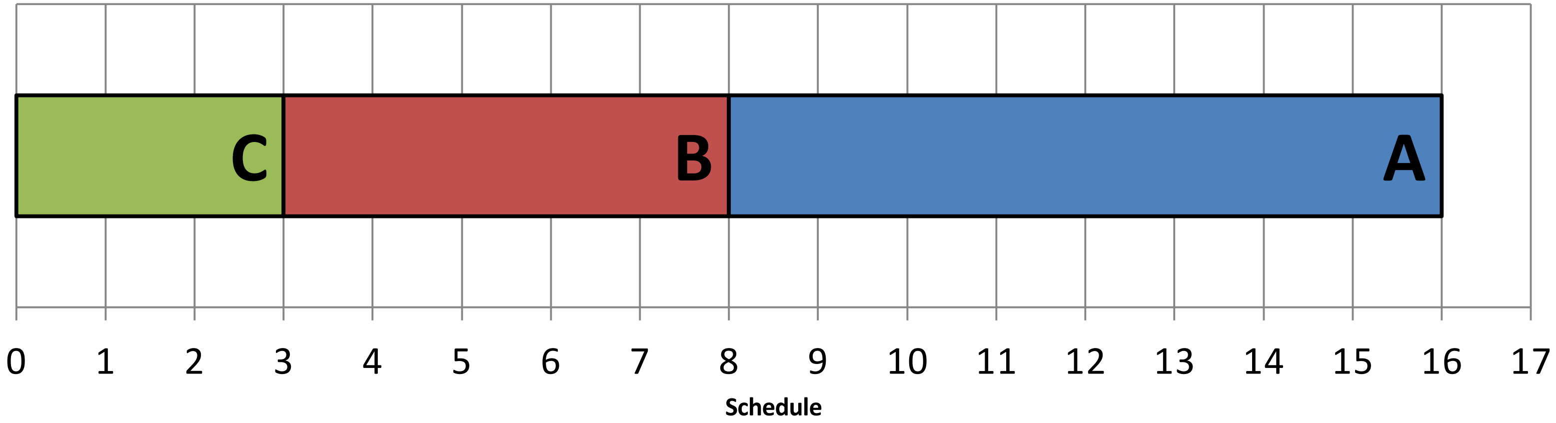
- Simple reordering can reduce the **average waiting time**!
- **Convoy Effect:**
  - First task (task **A**) is CPU-Bound and followed by a number IO-Bound tasks $\mathbf{X_1, X_2, \ldots, X_n}$
  - Tasks **A** running
    - All tasks $\mathbf{X_i}$ waiting in ready queue (I/O device sitting idle)
  - Tasks **A** blocked on I/O
    - All tasks $\mathbf{X_i}$ execute quickly and blocked on I/O (CPU sitting idle)

Shortest Job First: SJF

- General Idea:
  - Select task that needs the **shortest amount of CPU time**
    - Before it blocks or releases CPU or terminates
- Notes:
  - Need to know **total CPU time** for a task in advance
    - Have to "guess" if this info is not available
  - Given a fixed set of tasks:
    - Minimizes **average waiting time**
  - Starvation is possible:
    - Biased towards short jobs
    - Long job may never get a chance

Shortest Job First: Illustration

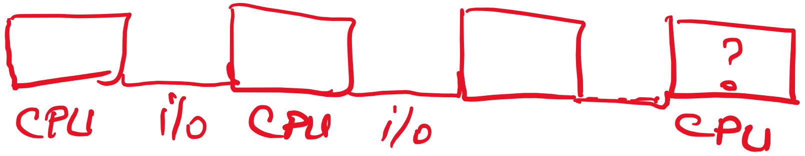




- The average total waiting time for 3 tasks
  - (0 + 3 + 8)/3 = **3.66** Time Units
- Can be shown that SJF **guarantees** smallest average waiting time

Shortest Job First: **Predicting CPU Time**

- Guess the **future CPU** time by the **previous CPU-Bound phases**
  - A task usually goes through several phases of CPU-Activity
  - Possible to guess the future CPU time requirement by the previous CPU-Bound phases



- Common approach (*Exponential Average*):

$$\texttt{Predicted}_{n+1} = \alpha\texttt{Actual}_{n} + (1-\alpha)\,\texttt{Predicted}_{n}$$

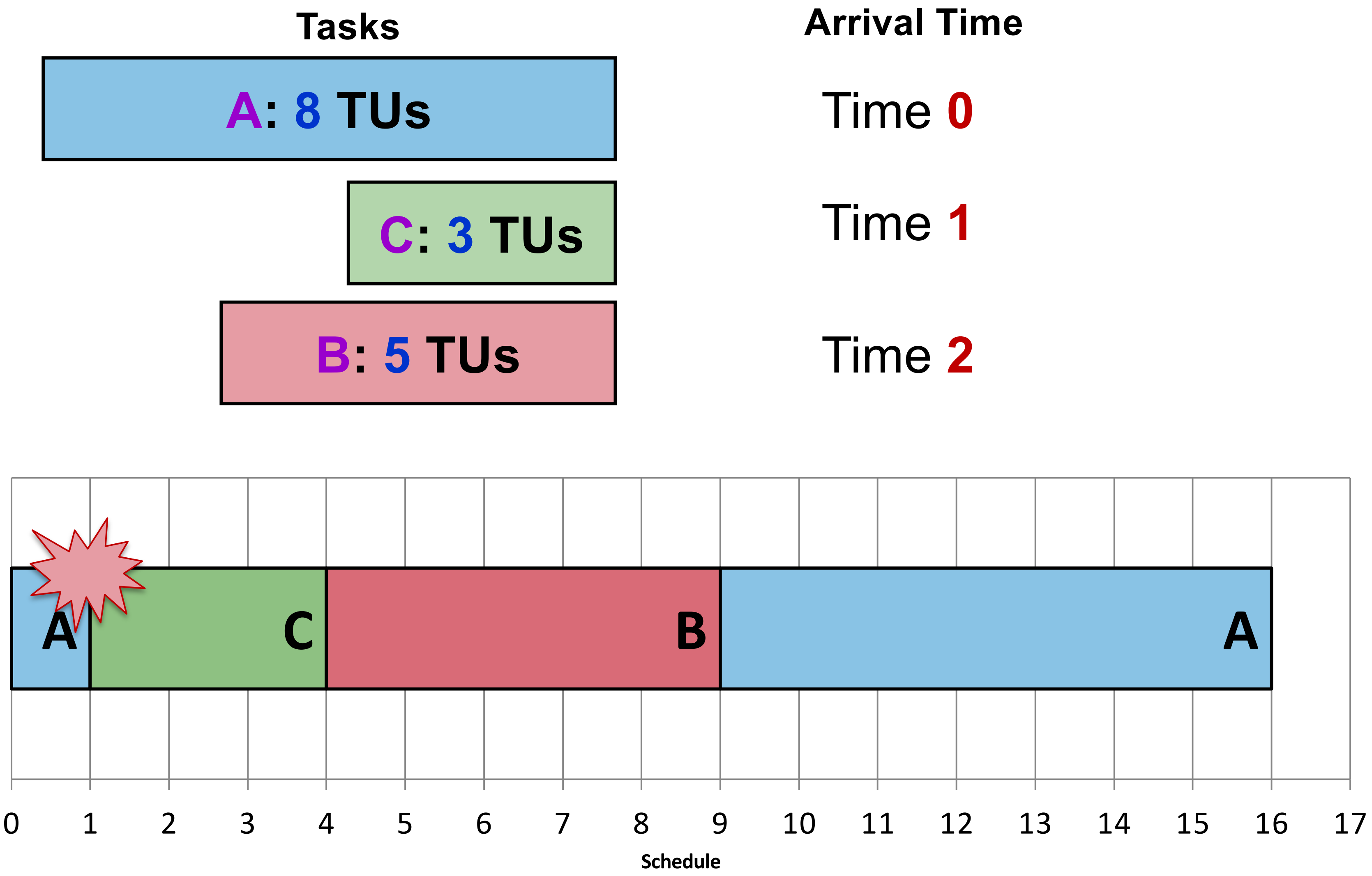
  - $\texttt{Actual}_n$ = The most recent CPU time consumed
  - $\texttt{Predicted}_n$ = The past history of CPU Time consumed
  - α = Controls the weight placed on recent event or past history
  - $\texttt{Predicted}_{n+1}$ = Latest prediction

Shortest Remaining Time: SRT

- General Idea:
  - Variation of **SJF**:
    - Use remaining time
    - **Preemptive**
  - Select job with shortest remaining time (or the expectation thereof)

- Notes:
  - New job with shorter remaining time **can preempt currently running job**
  - Provides good service for short job even when it arrives late

Shortest Remaining Time First: Illustration



SCHEDULING FOR INTERACTIVE SYSTEMS

Criteria for interactive environment

- **Response time**:
  - Time between request and response by system
- **Predictability**:
  - Variation in response time; less variation → better predictability
  - Predictability even more important in real-time environments

Preemptive scheduling algorithms are used to ensure good response time

→ Scheduler needs to run **periodically**

Ensuring **Periodic Scheduler Invocation**

- **Questions:**
  - How can the scheduler "take over" the CPU periodically?
  - How to ensure that user program cannot prevent the scheduler from executing?

- Ingredients for answer:
  - **Timer interrupt** = Interrupt that goes off periodically (based on a hardware clock)
  - OS ensures timer interrupt cannot be intercepted by any other program (or any other interrupt!)

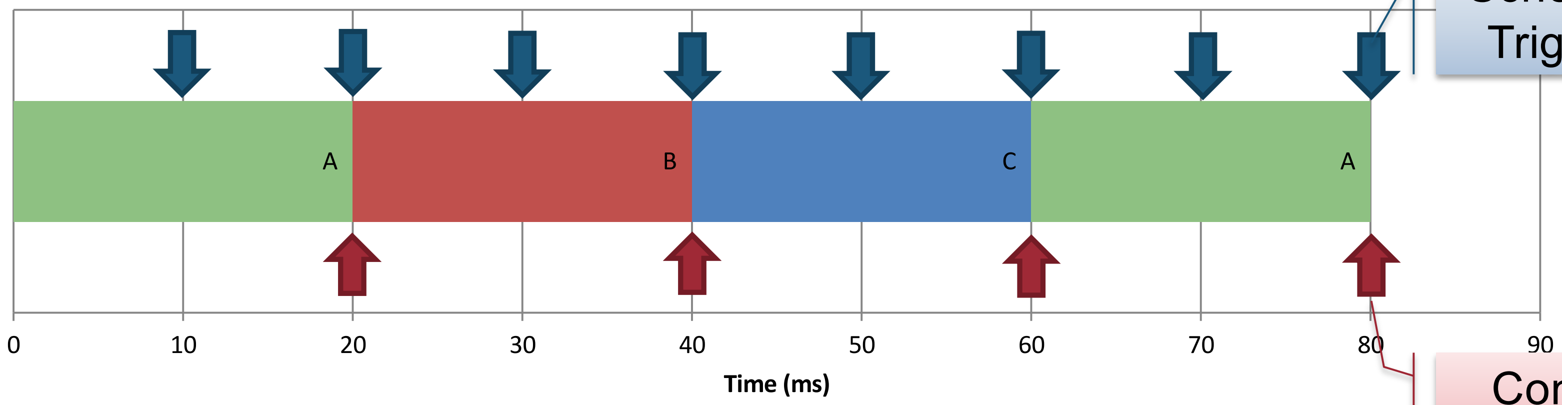
  ➔ Timer interrupt handler **invokes the scheduler**

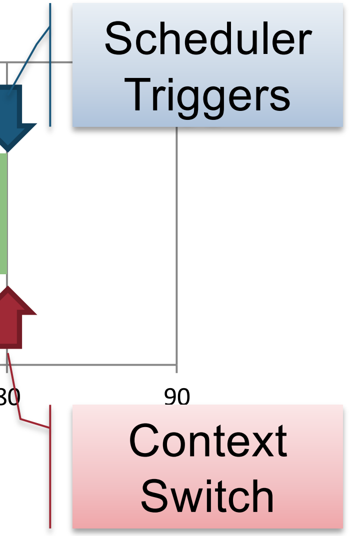
Terminology: Timer & Time Quantum

- **Interval of Timer Interrupt (ITI):**
  - i.e., the timer period
  - OS scheduler is triggered every timer interrupt
  - Typical values (**1ms to 10ms**)

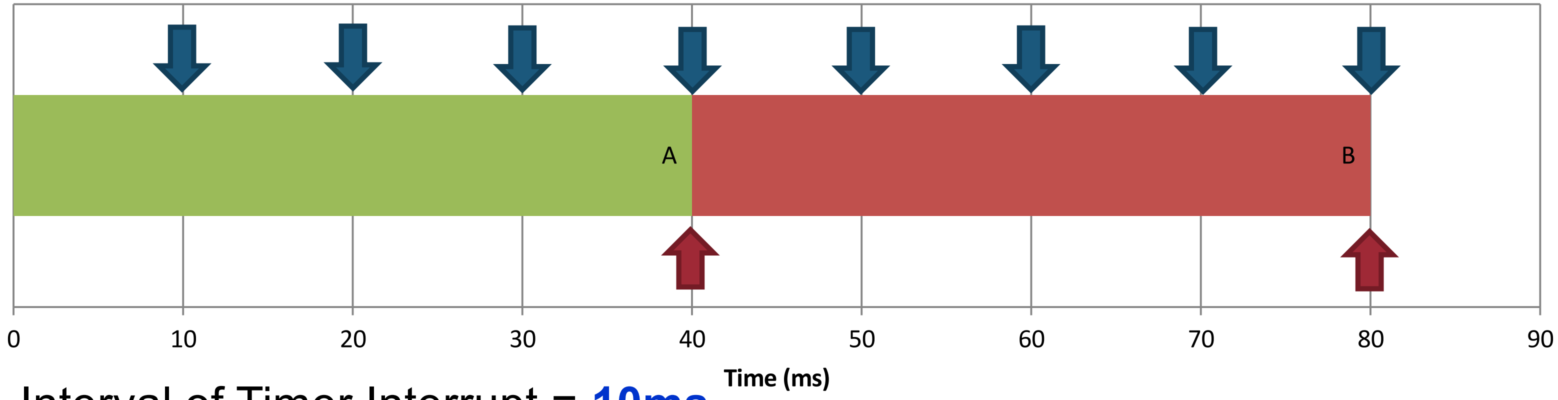
- **Time Quantum:**
  - Execution duration given to a process
  - Could be ***constant*** or ***variable*** among the processes
  - Must be multiples of interval of timer interrupt
  - Large range of values (commonly **5ms to 100ms**)

Illustration: ITI vs Time Quantum





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



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

Scheduling Algorithms:

- Algorithms covered:
  1. **R**ound **R**obin (**RR**)
  2. **Priority** Based
  3. **M**ulti-**L**evel **F**eedback **Q**ueue (**MLFQ**)
  4. **Lottery** Scheduling

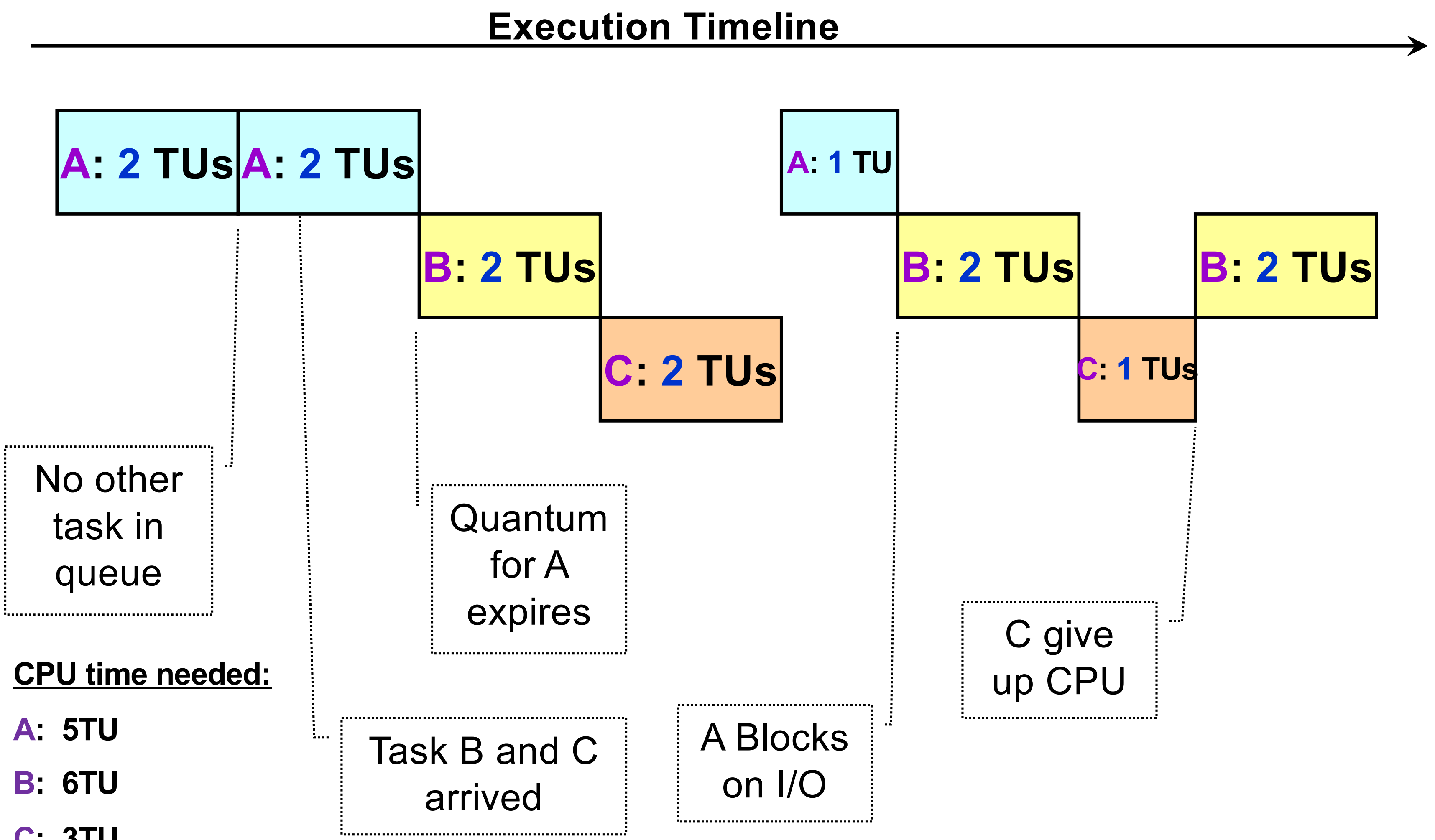
Round Robin: **RR**

- General Idea:
  - Tasks are stored in a **FIFO queue**
  - Pick the **first task from queue** front to run until:
    - The task gives up the CPU voluntarily, or
    - The task blocks, or
    - A fixed **time slice** (**quantum**) elapsed // key difference from FCFC
  - The task is then **placed at the end of queue** to wait for another turn
    - Blocked task will be moved to another queue to wait for its request
    - When blocked task is ready again, it is placed at the end of queue

Round Robin: **RR** (cont.)

- Notes:
  - Basically a preemptive version of FCFS
  - **Response time guarantee**:
    - Given *n* tasks and quantum *q*
    - Time before a task get CPU is bounded by $(n-1)q$
  - **Timer interrupt needed:**
    - For scheduler to check on quantum expiry
  - The **choice of time quantum** duration is important:
    - **Big quantum:** Better CPU utilization but longer waiting time
    - **Small quantum:** Bigger overhead (worse CPU utilization) but shorter waiting time

Round Robin: **Illustration**



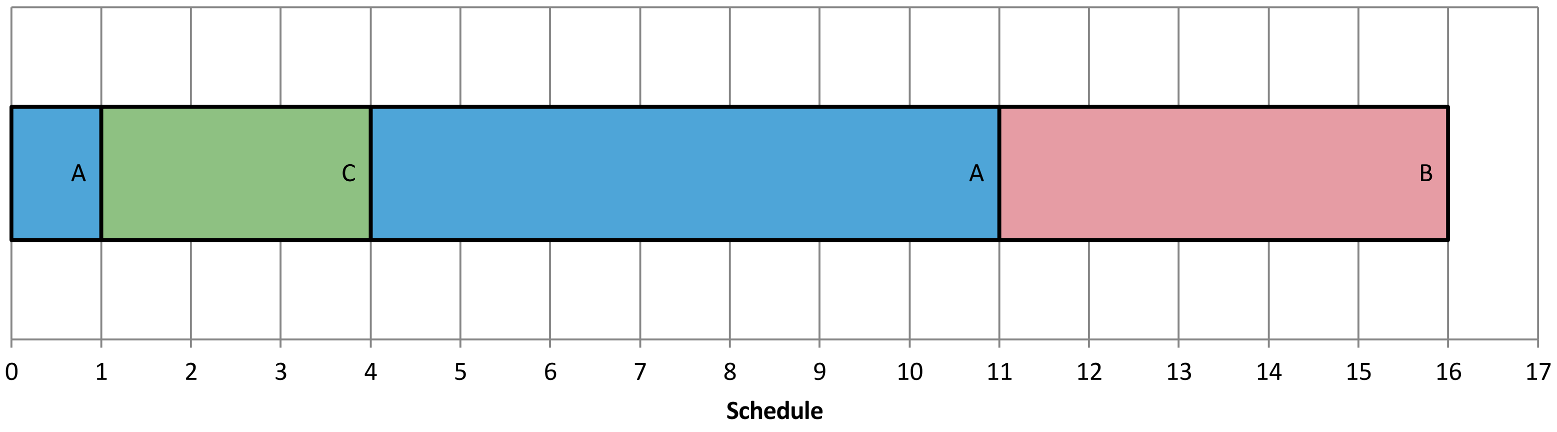
<u>Arrival Time:</u>	<u>CPU time needed:</u>
A: 0	A: 5TU
B: at 2.8TU	B: 6TU
C: at 2.9TU	C: 3TU

Priority Scheduling

- General Idea:
  - Some processes are more important than others
    - Cannot treat all process as equal
  - Assign a **priority value** to all tasks
  - Select task with **highest priority value**
- Variants:
  - **Preemptive version:**
    - Higher priority process can preempts running process with lower priority
  - **Non-preemptive version:**
    - Late coming high-priority process has to wait for next round of scheduling

Priority Scheduling: Illustration

Tasks	Arrival Time	Priority (1=highest)
A: 8 TUs	Time 0	3
C: 3 TUs	Time 1	1
B: 5 TUs	Time 1	5



Priority Scheduling: **Shortcomings**

- Low priority process can **starve**:
  - High priority process keep hogging the CPU
  - Even worse in preemptive variant. Why?
- Possible solutions:
  - Decrease the priority of currently running process after every time quantum
    - Eventually dropped below the next highest priority
  - Give each process a **minimum time quantum**
    - Ensures that low-priority processes run for a while when they eventually get a chance
- Hard to guarantee/control the exact amount of CPU time given to a process using priority

Multi-Level Feedback Queue (MLFQ)

- Designed to solve one BIG + HARD issue:
  - How do we schedule without perfect knowledge?
  - Most algorithms require certain information (*process behavior*, *running time*, etc)

- **MLFQ** is:
  - **Adaptive**:
    - "Learn the process behavior automatically"
  - Seeks to minimize both:
    - Response time for interactive and IO-bound processes
    - Turnaround time for CPU-bound processes

MLFQ: Rules

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

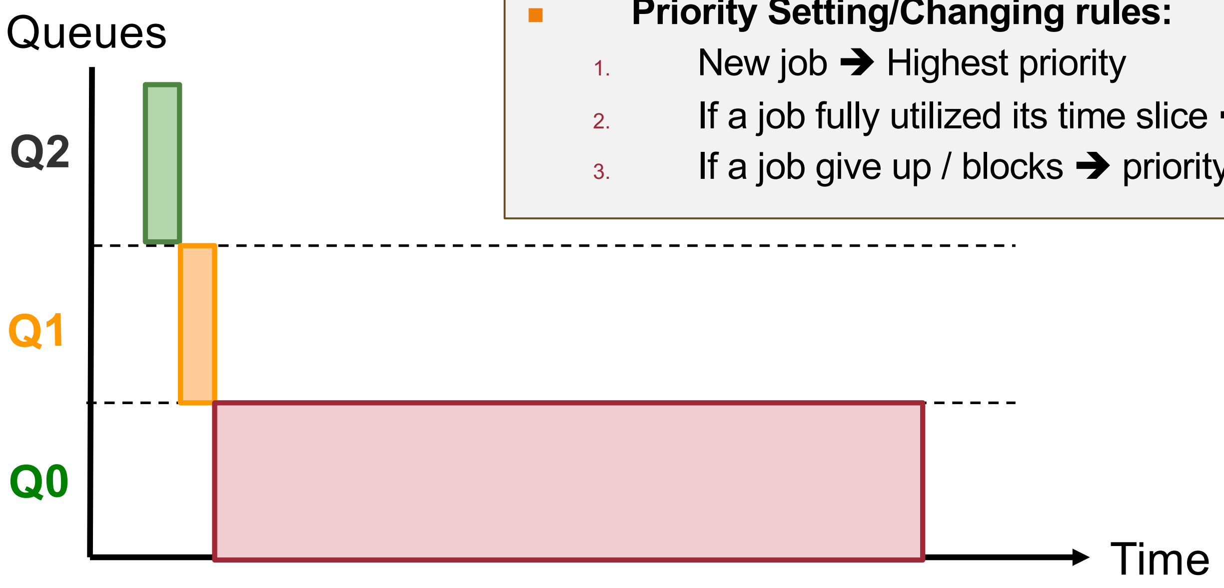
- **Priority Setting/Changing rules:**
  1. New job ➔ Highest priority
  2. If a job fully utilized its time slice ➔ priority reduced
  3. If a job give up / blocks before it finishes the time slice ➔ priority retained

MLFQ: Example 1

- 3 Queues: Q2 (highest priority), Q1, Q0
- A single long running job

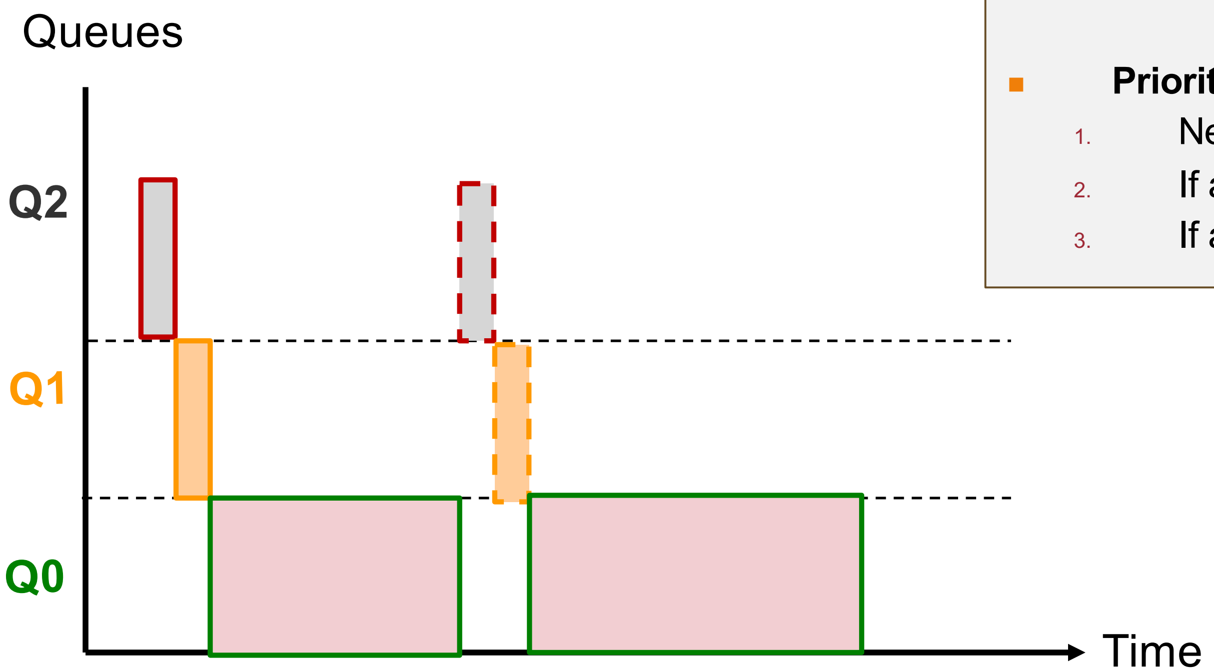
- **Basic rules:**
  1. If Priority(A) > Priority(B) ➔ A runs
  2. If Priority(A) == Priority(B) ➔ A and B runs in RR

- **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 ➔ priority retained



MLFQ: Example 2

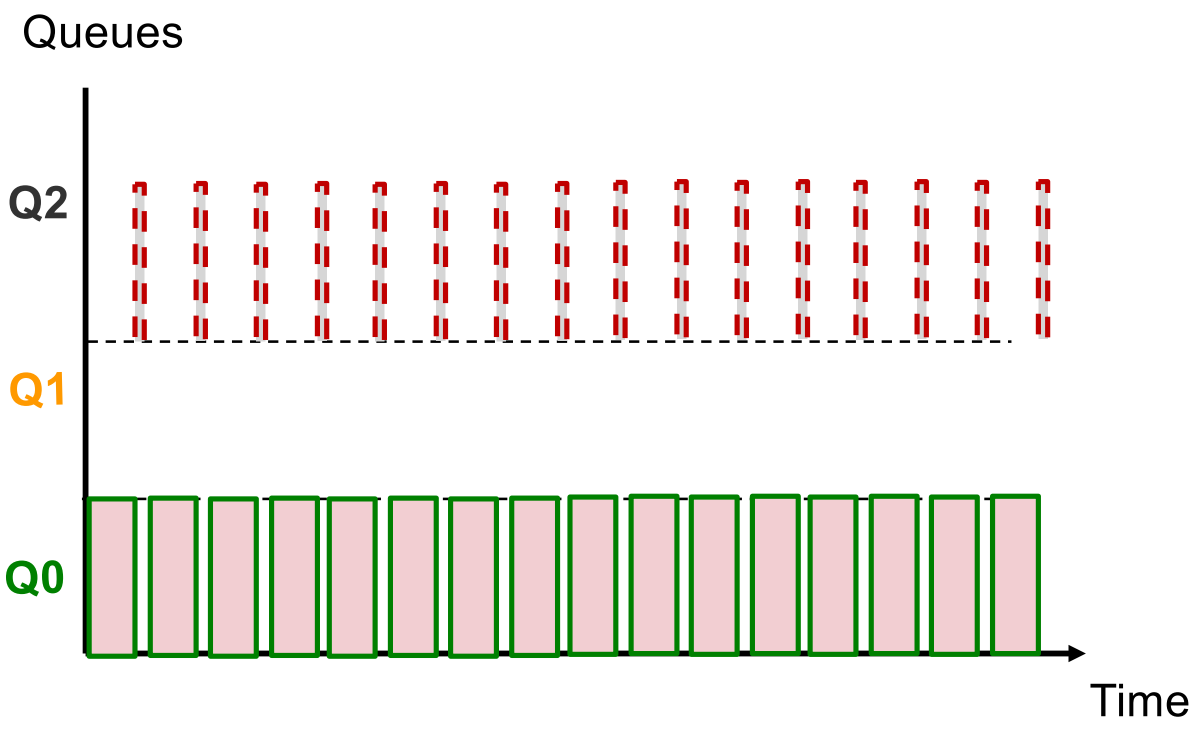
- Example 1 + a short job in the middle
  - A short job appears sometime in the middle



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

MLFQ: Example 3

- Two jobs:
  - A = CPU bound (already in the system for quite some time)
  - B = I/O bound



MLFQ: Questions to ponder

- Can you think of a way to abuse the algorithm? ☺
  - Equivalent question: MLFQ does not work well for what kind combination of jobs?

- What are the ways to rectify the above?

Lottery Scheduling

General Idea:

- Scheduling is done in rounds. In every round:
  - Give out "*lottery tickets*" to processes for various system resources
    - E.g., CPU time, I/O device etc
  - When a scheduling decision is needed:
    - A lottery ticket is **chosen randomly among eligible tickets**
    - The winner is **granted the resource**
- In every round, a process holding **X%** of tickets
  - Can win **X%** of the lottery held
  - Use the resource **X%** of the time

Lottery Scheduling: Properties

- **Responsive:**
  - Every participating process gets to run in every round
  - A newly created process can participate in the next lottery round
- Provides **good level of control**:
  - A process can be given **Y** lottery tickets
    - It can then distribute to its child process
  - An important process can be given more lottery tickets
    - Can control the proportion of usage
  - Each resource can have its own set of tickets
    - Different proportion of usage **per resource** **per task**
  - Simple implementation

Summary

- Scheduling in OS:
  - Basic definition
  - Factors that affect scheduling
    - Process, Environment
  - Criteria of good scheduling

- Scheduling Algorithms:
  - FCFS, SJF, SRT for Batch Processing System
  - RR, Priority, Multi-Level Queues and Lottery scheduling for Interactive System

References

- Modern Operating System (4^{th} Edition)
  - **Chapter 2.4**

- Operating System Concepts (9^{th} Edition)
  - **Chapter 6**

- Operating Systems: Three Easy Pieces
  - http://pages.cs.wisc.edu/~remzi/OSTEP
  - **Chapters 7, 8, 9**
  - Advanced (optional): chapter 10