

# **Lecture 13:**

# **CPU Scheduling**

**(Threads, Dispatching, and Scheduling Policies)**

Xin Liu

xliu15@fsu.edu

COP 4610 Operating Systems

# Outline

- Threads
- Dispatcher
- Amdahl's Law
- CPU Scheduling

# Why *Concurrency*?

- Allows multiple applications to run at the same time
  - Analogy: juggling



- What is an applications?
  - A Program that runs on the OS.

# History Phase I: Hardware Expensive, Humans Cheap

- Hardware: mainframes
- OS: human operators
  - Handle one *job* (a unit of processing) at a time
  - Computer time wasted while operators walk around the machine room



IBM System/360

# OS Design Goal

- Efficient use of the hardware
  - ***Batch system***: collects a batch of jobs before processing them and printing out results
    - Job collection, job processing, and printing out results can occur concurrently
  - ***Multiprogramming***: multiple programs can run concurrently
    - Example: I/O-bound jobs and CPU-bound jobs

# History Phase II:

## Hardware Cheap, Humans Expensive

- Hardware: terminals
- OS design goal: more efficient use of human resources
  - *Timesharing systems*: each user can afford to own terminals to interact with machines
  - The operating system could support multiple users simultaneously, each with their own terminal
  - Each user had an efficient and responsive experience, without the need for dedicated machines for each person



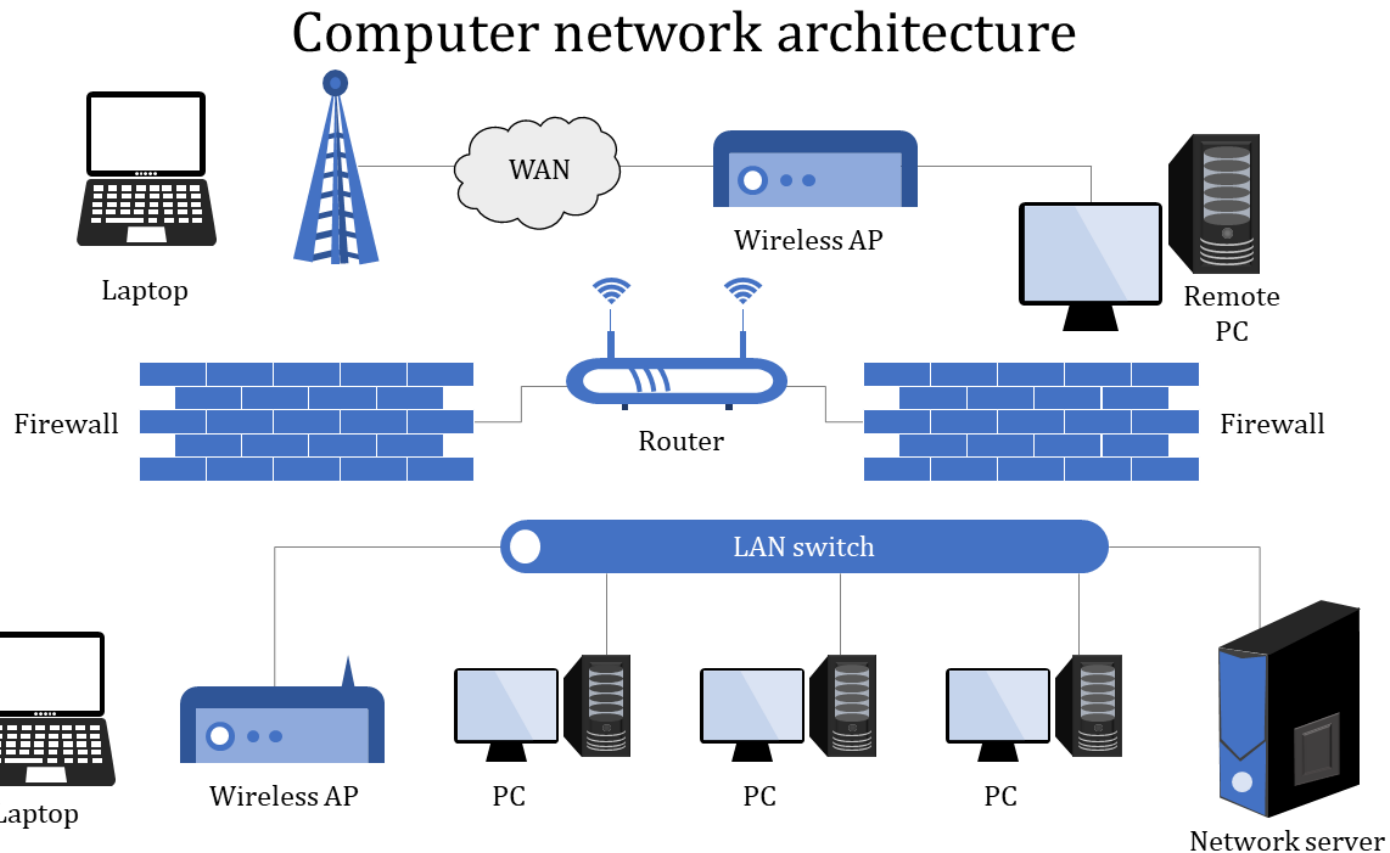
# History Phase III: Hardware Very Cheap, Humans Very Expensive

- Hardware: personal computers
- OS design goal: allowing a user to perform many tasks at the same time
  - *Multitasking*. a single user can run multiple programs on the same machine at the same time
  - *Multiprocessing*. the ability to use multiple processors on the same machine



# History Phase IV: Distributed Systems

- Hardware: computers with networks





# Why *Concurrency*?

- Allows multiple applications to run at the same time
  - Analogy: juggling
- Program is a state machine.
- What is the red ball?
  - A State.



# Benefits of Concurrency

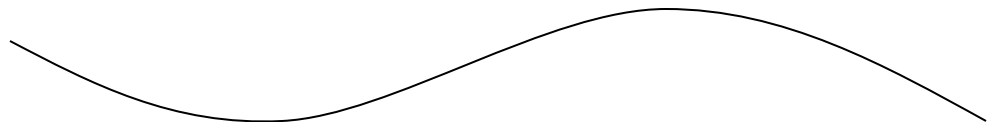
- Better performance
  - One application uses only the processor
  - One application uses only the disk drive
  - Completion time is shorter when running both concurrently than consecutively

# Drawbacks of Concurrency

- Applications need to be protected from one another
- Additional coordination mechanisms among applications
- Overhead to switch among applications
- Potential performance degradation when running too many applications

# *Thread*

- A sequential execution stream
  - The smallest CPU scheduling unit
  - Can be programmed as if it owns the entire CPU
    - Implication: an infinite loop within a thread won't halt the system
  - Illusion of multiple CPUs on a single-CPU machine



# Thread Benefits

- Simplified programming model per thread
- Example: Microsoft Word
  - One thread for grammar check; one thread for spelling check; one thread for formatting; and so on...
  - Can be programmed independently
  - Simplifies the development of large applications

# *Address Space*

- Contains all states necessary to run a program
  - Code, data, stack
  - Program counter
  - Register values
  - Resources required by the program
  - Status of the running program
- **A mechanism to protect one app from crashing another app**

# *Process*

- An address space + at least one thread of execution
  - Address space offers protection among processes
  - Threads offer concurrency
- A fundamental unit of computation

# Process =? Program

- *Program*: a collection of statements in C or any programming languages
- Process: a running instance of the program, with additional states and system resources
- Two processes can run the same program
  - The code segment of two processes are the same program
- A program can create multiple processes
  - Example: gcc, chrome

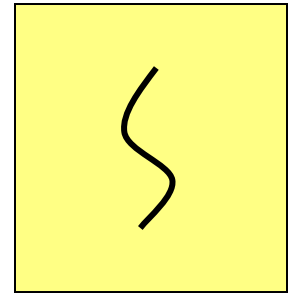


# Real-life Analogy?

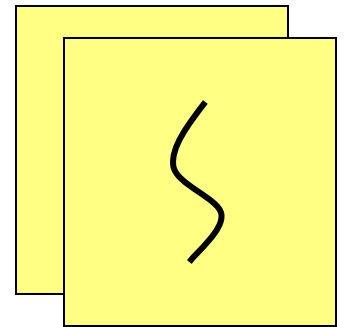
- Program: a recipe
- Process: everything needed to cook
  - e.g., kitchen
- Two chefs can cook the same recipe in different kitchens
- One complex recipe can involve several chefs

# Some Definitions

- *Uniprogramming*: running one process at a time

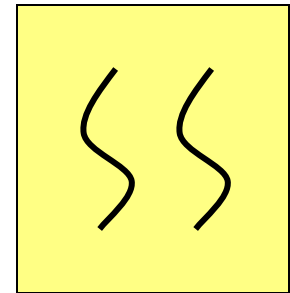


- *Multiprogramming*: running multiple processes on a machine



# Some Definitions

- *Multithreading*: having multiple threads per address space (threads share the same address space)
- *Multiprocessing*: running programs on a machine with multiple processors
- *Multitasking*: a single user can run multiple processes



# Classifications of OSes

	Single address space	Multiple address spaces
Single thread	MS DOS, Macintosh	Traditional UNIX
Multiple threads	Embedded systems	Windows, iOS

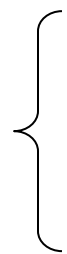
# Threads & Thread Control Block

- A thread owns a *thread control block*
  - Execution states of the thread
  - The status of the thread
    - Running or sleeping
  - Scheduling information of the thread
    - e.g., priority

# Threads & Dispatching Loop

- Threads are run from a *dispatching loop*
  - Can be thought as a per-CPU thread
  - LOOP
    - Run thread
    - Save states
    - Choose a new thread to run ← *Scheduling*
    - Load states from a different thread

*Context  
switch*



# Simple? Not quite...

- How does the dispatcher regain control after a thread starts running?
- What states should a thread save?
- How does the dispatcher choose the next thread?

# How does the dispatcher regain control?

- Two ways:
  1. Internal events (“Sleeping Beauty”)
    - Yield—a thread gives up CPU voluntarily
      - A thread is waiting for I/O
      - A thread is waiting for some other thread
  2. External events
    - Interrupts—a complete disk request
    - Timer—it’s like an alarm clock



# What states should a thread save?

- Anything that the next thread may trash before a context switch
  - Program counter
  - Registers
  - Changes in execution stack

# How does the dispatcher choose the next thread?

- The dispatcher keeps a list of threads that are ready to run
- If no threads are ready
  - Dispatcher just loops
- If one thread is ready
  - Easy

# How does the dispatcher choose the next thread?

- If more than one thread are ready
  - We choose the next thread based on the scheduling policies
  - Examples
    - FIFO (first in, first out)
    - LIFO (last in, first out)
    - Priority-based policies

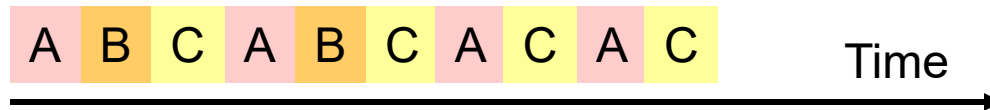
# How does the dispatcher choose the next thread?

- Additional control by the dispatcher on how to share the CPU
  - Suppose we have three threads

Run to completion



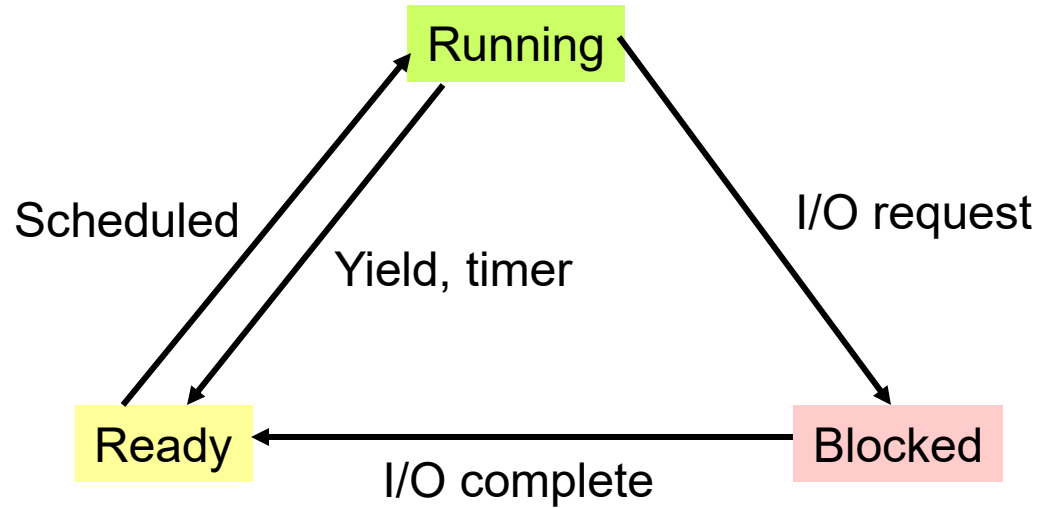
Timeshare the CPU



# Per-thread States

- Each thread can be in one of the three states
  1. *Running*: has the CPU
  2. *Blocked*: waiting for I/O or another thread
  3. *Ready to run*: on the ready list, waiting for the CPU

# Per-thread State Diagram



# For Multi-core Machines

- Each core has a dispatcher loop
  - Decide which thread will execute next
- One core has a global dispatcher loop
  - Decide which core to execute a thread

# Parallelism vs. Concurrency

- *Parallel* computations
  - Computations can happen at the same time on separate cores
- *Concurrent* computations
  - One unit of computation does not depend on another unit of computation
    - Can be done in parallel on multiple cores
    - Can time share a single core (not parallel)



# Real-life Example

- Two hands are playing piano in parallel (not concurrently)
  - Notes from left and right hands are dependent on each other
- Two separate groups singing 'row row row your boat' concurrently (and in parallel)

# Amdahl's Law

- Identifies potential performance gains from adding cores
  - $P$  = % of program that can be executed in parallel
  - $N$  = number of cores

- $speedup \leq \frac{1}{(1-P) + \frac{P}{N}}$

# Amdahl's Law

- Example

- P = 75% of program that can be executed in parallel
- N = 2 cores

- $speedup \leq \frac{1}{(1-0.75) + \frac{0.75}{2}} = 1.6$

# CPU Scheduler

- A *CPU scheduler* is responsible for
  - Removal of running process from the CPU
  - Selection of the next running process
    - Based on a particular strategy

# Goals for a Scheduler

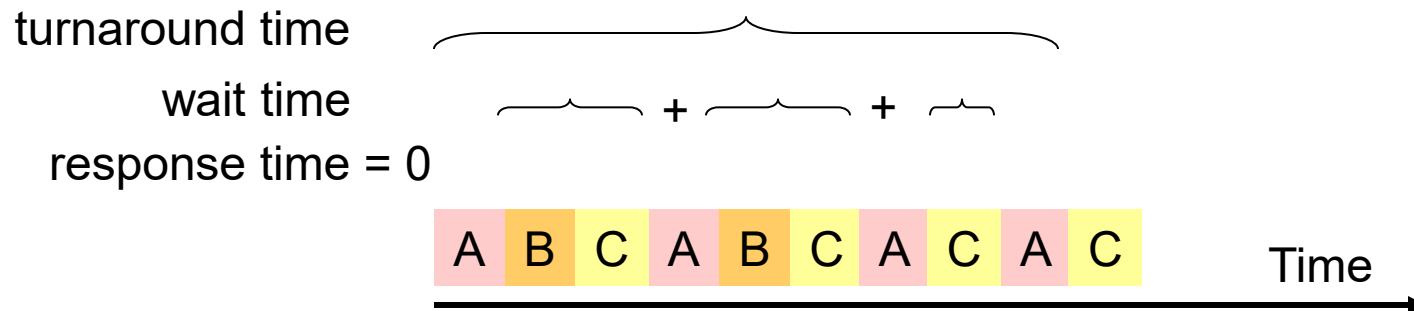
- Maximize
  - *CPU utilization*: keep the CPU as busy as possible
  - *Throughput*: the number of processes completed per unit time

# Goals for a Scheduler

- Minimize
  - *Response time*: the time of submission to the time the first response is produced
  - *Wait time*: total time spent waiting in the ready queue
  - *Turnaround time*: the time of submission to the time of completion

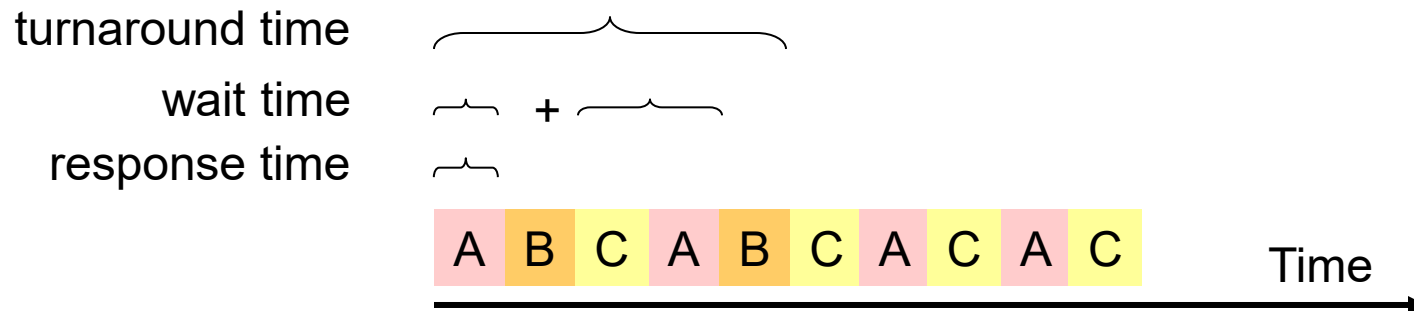
# Goals for a Scheduler

- Suppose we have processes A, B, and C, submitted at time 0
- We want to know the response time, wait time, and turnaround time of process A



# Goals for a Scheduler

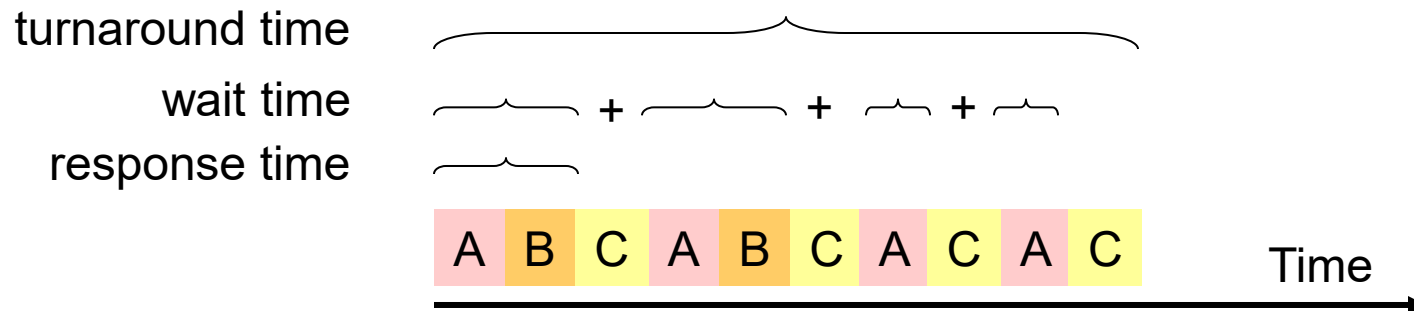
- Suppose we have processes A, B, and C, submitted at time 0
- We want to know the response time, wait time, and turnaround time of process B





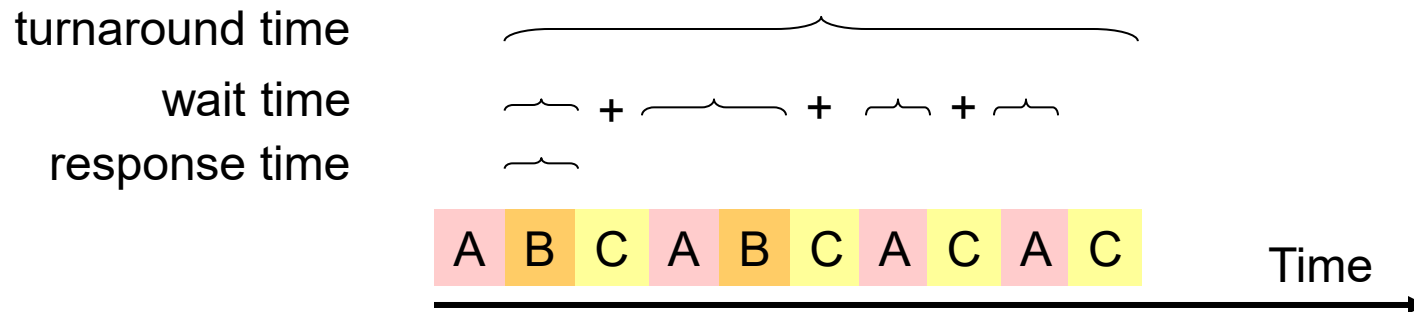
# Goals for a Scheduler

- Suppose we have processes A, B, and C, submitted at time 0
- We want to know the response time, wait time, and turnaround time of process C



# Goals for a Scheduler

- Suppose we have processes A and B submitted at time 0; process C, time 1
- We want to know the response time, wait time, and turnaround time of process C



# Goals for a Scheduler

- Achieve *fairness*
  - What is fair?
    - Guaranteed to have at least  $1/n$  share
    - How do two people divide a cake in a fair way?
- There are tensions among these goals

# Assumptions

- Each user runs one process
- Each process is single threaded
- Processes are independent
- They are not realistic assumptions; they serve to simplify analyses

# Scheduling Policies

- FIFO (first in, first out)
- Round robin
- SJF (shortest job first)
- Multilevel feedback queues
- Lottery scheduling

# FIFO

- *FIFO*: assigns the CPU based on the order of requests
  - *Nonpreemptive*: A process keeps running on a CPU until it is blocked or terminated
  - Also known as FCFS (first come, first serve)
  - + Simple
  - Short jobs can get stuck behind long jobs

# Round Robin

- *Round Robin (RR)* periodically releases the CPU from long-running jobs
  - Based on timer interrupts so short jobs can get a fair share of CPU time
  - *Preemptive*: a process can be forced to leave its running state and replaced by another running process
  - *Time slice*: interval between timer interrupts

# More on Round Robin

- If time slice is too long
  - Scheduling degrades to FIFO
- If time slice is too short
  - Throughput suffers
  - Context switching cost dominates



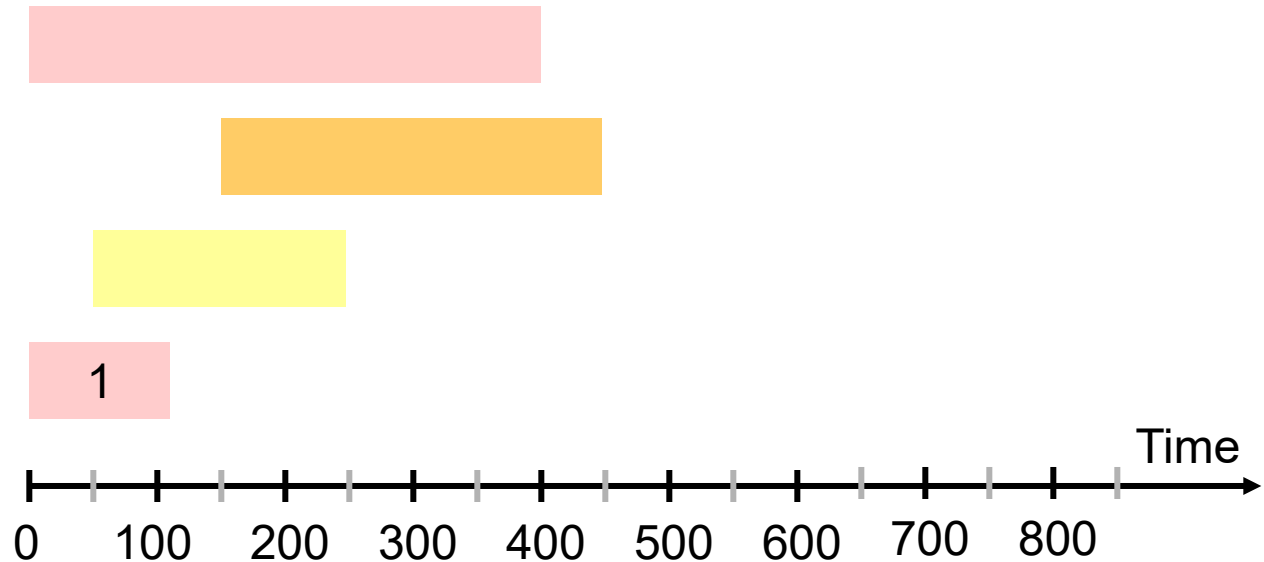
# Round Robin (Time slice = 100)

Process 1

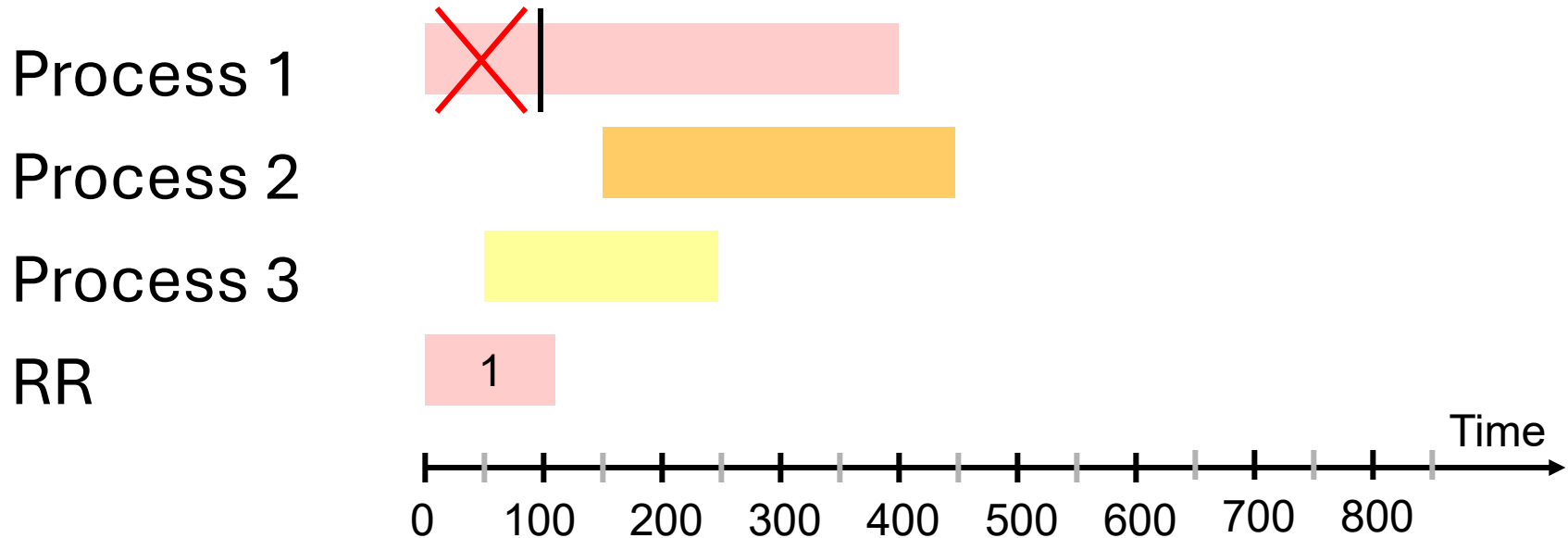
Process 2

Process 3

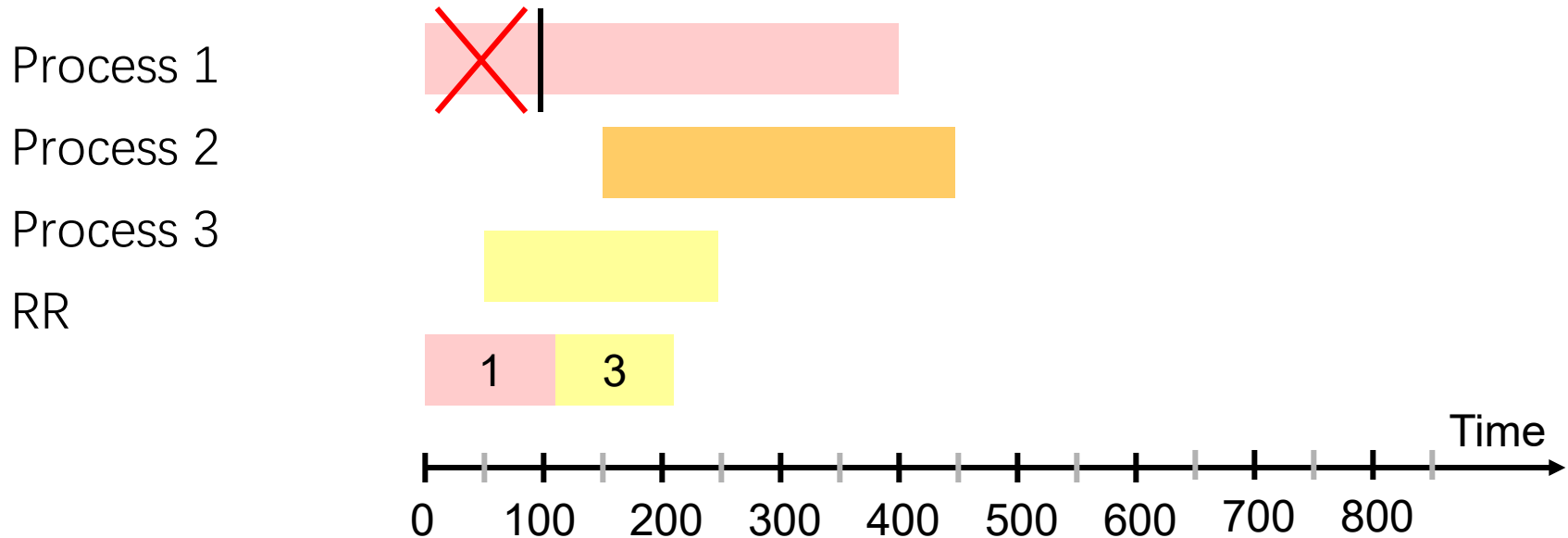
RR



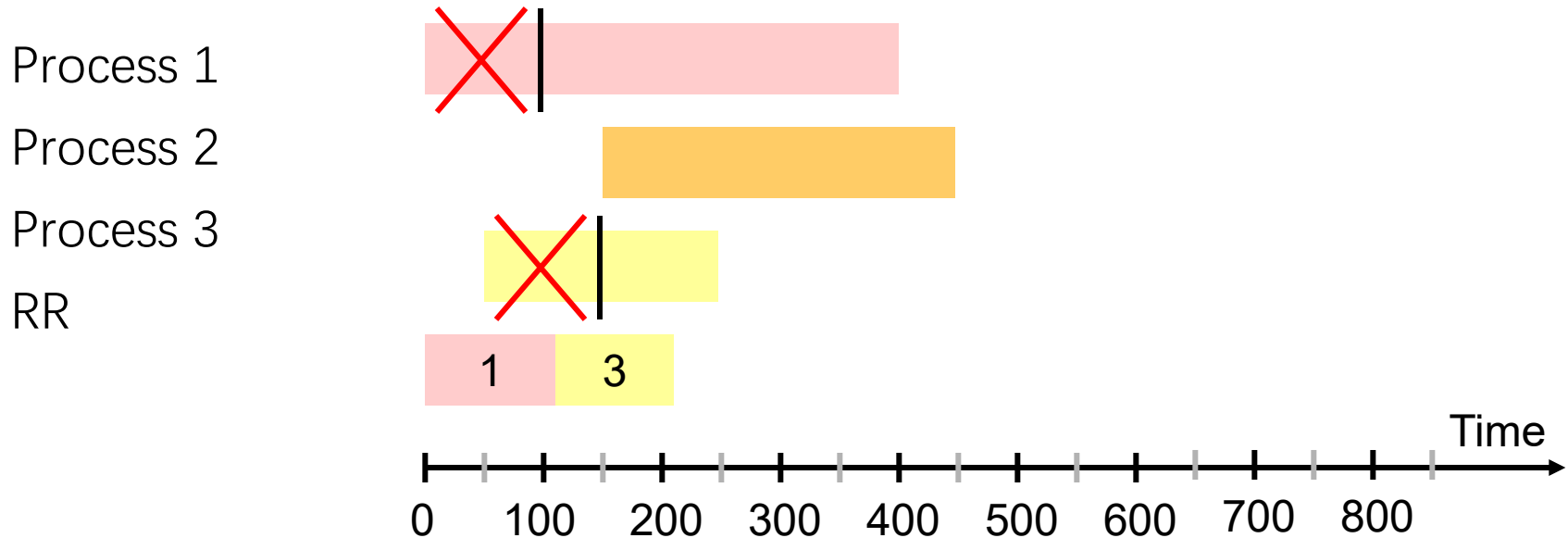
# Round Robin (Time slice = 100)



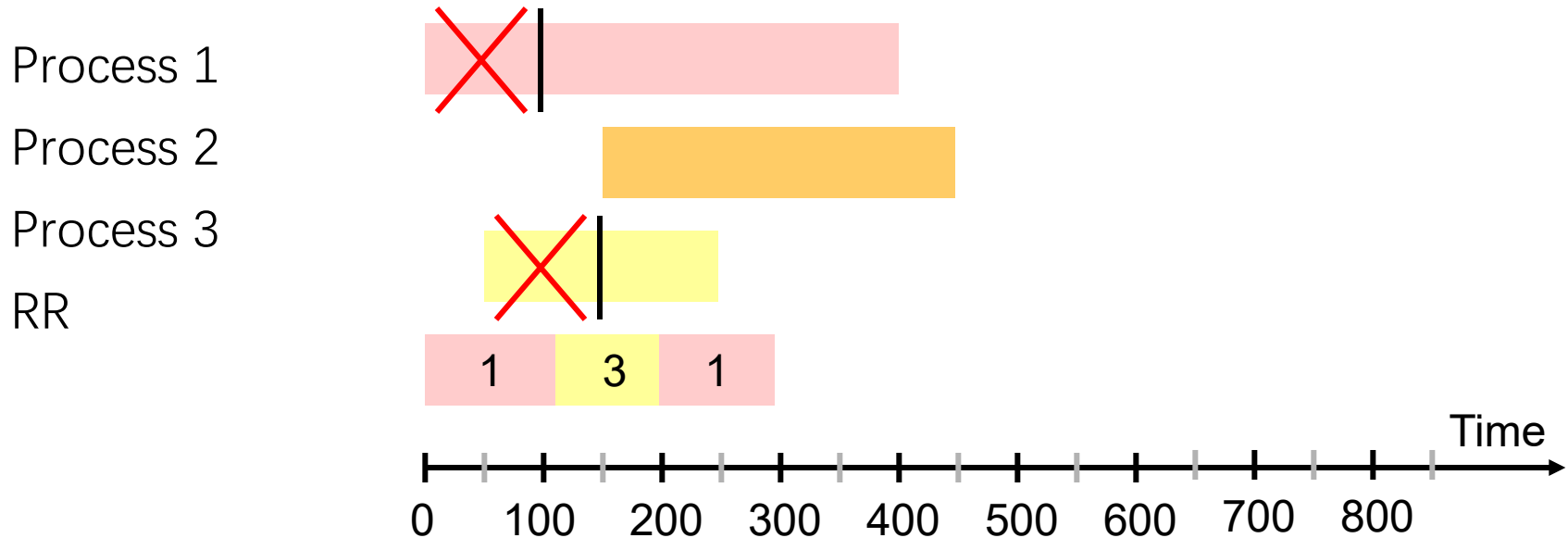
# Round Robin (Time slice = 100)



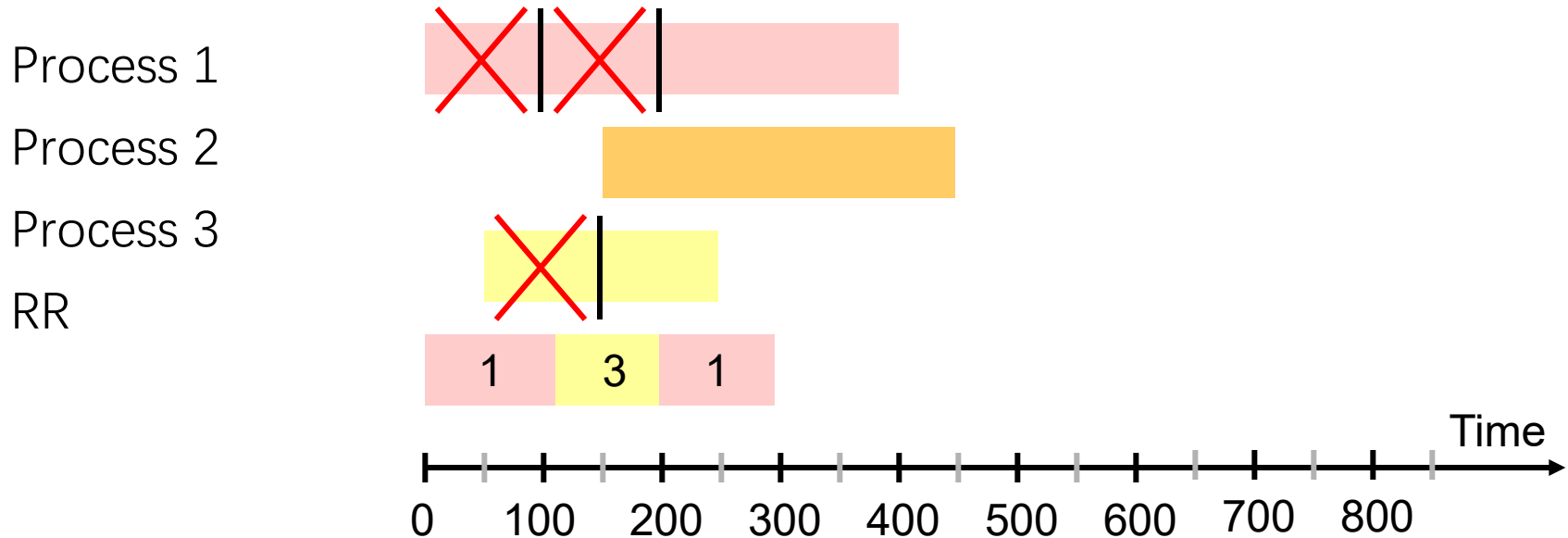
# Round Robin (Time slice = 100)



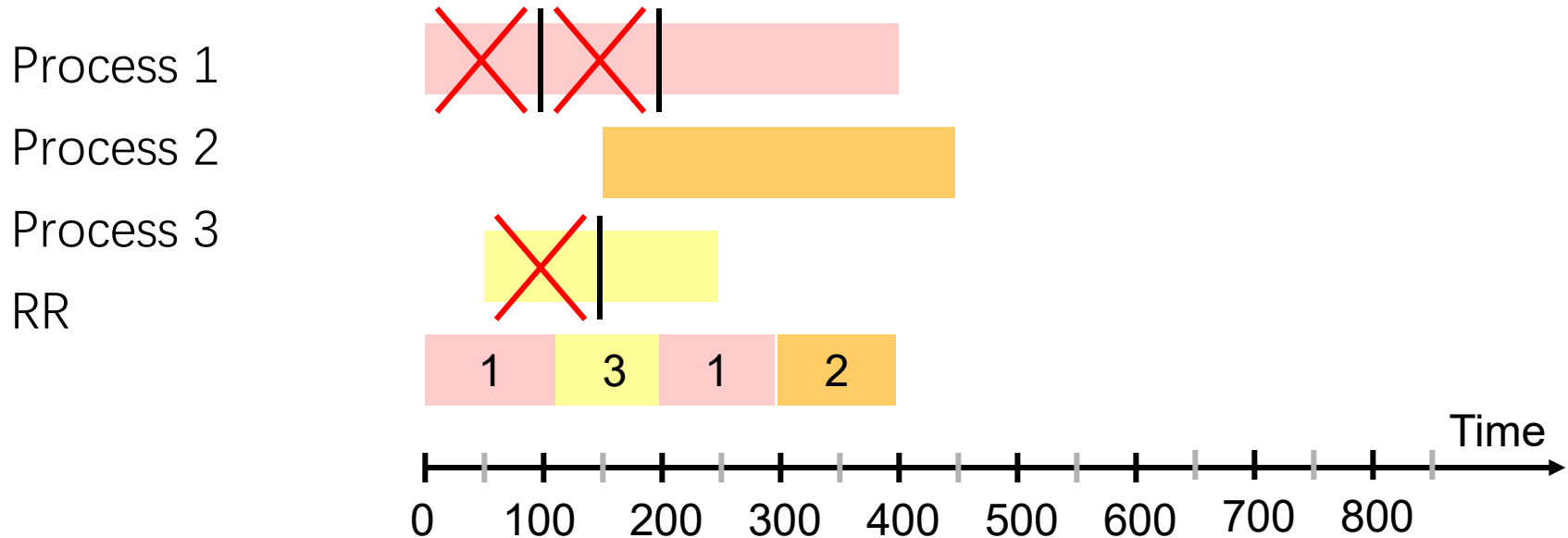
# Round Robin (Time slice = 100)



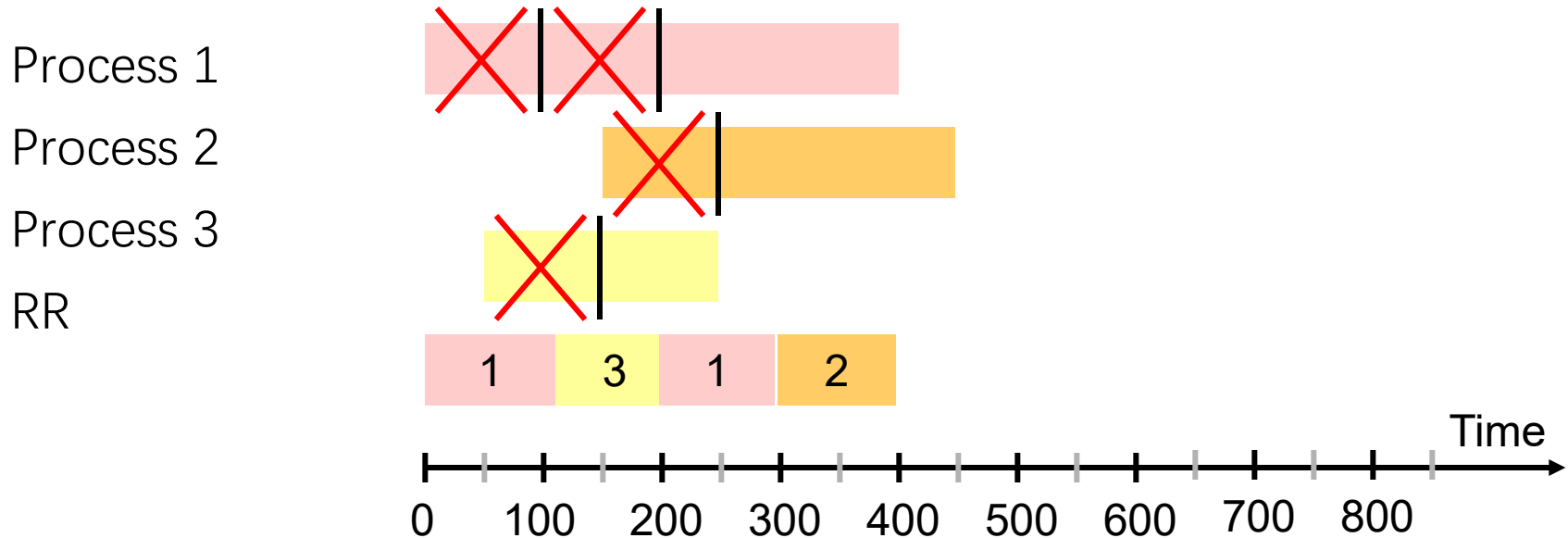
# Round Robin (Time slice = 100)



# Round Robin (Time slice = 100)

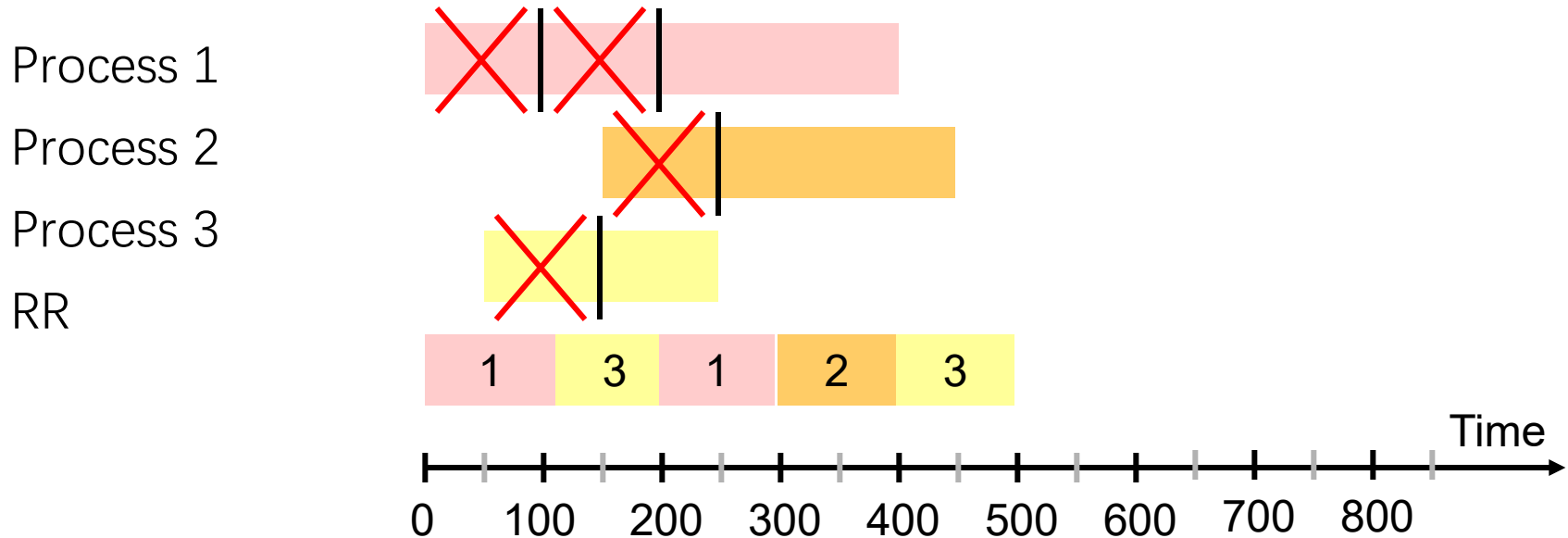


# Round Robin (Time slice = 100)

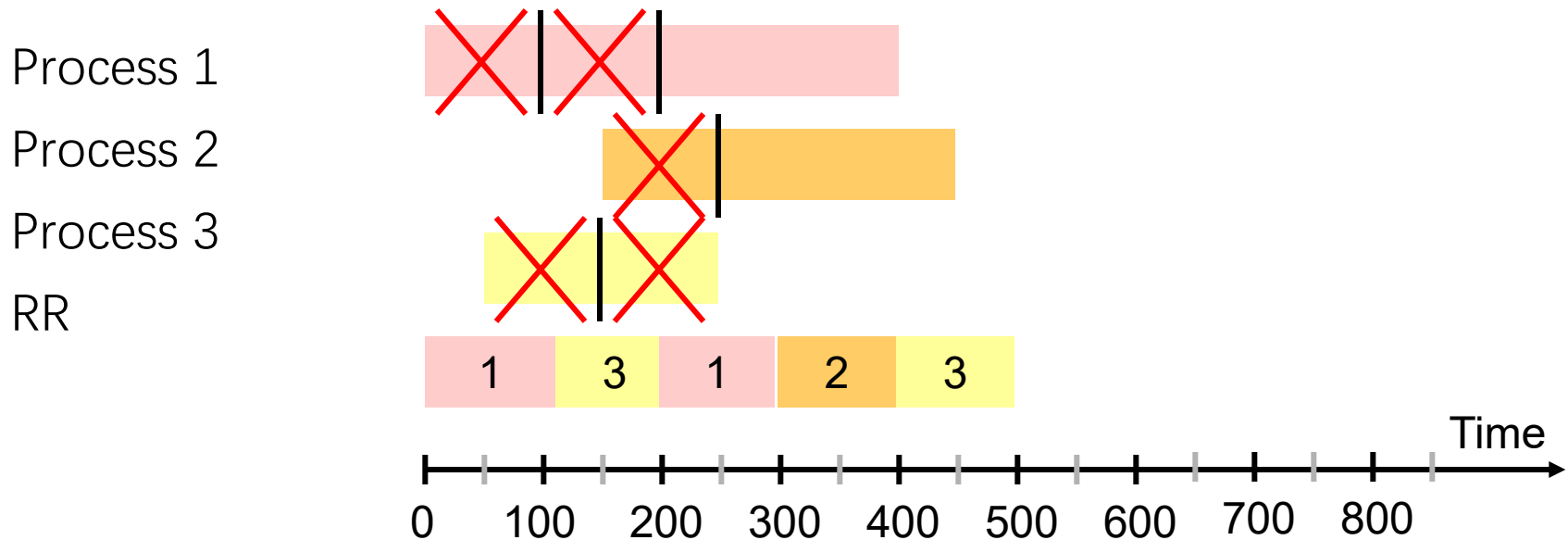




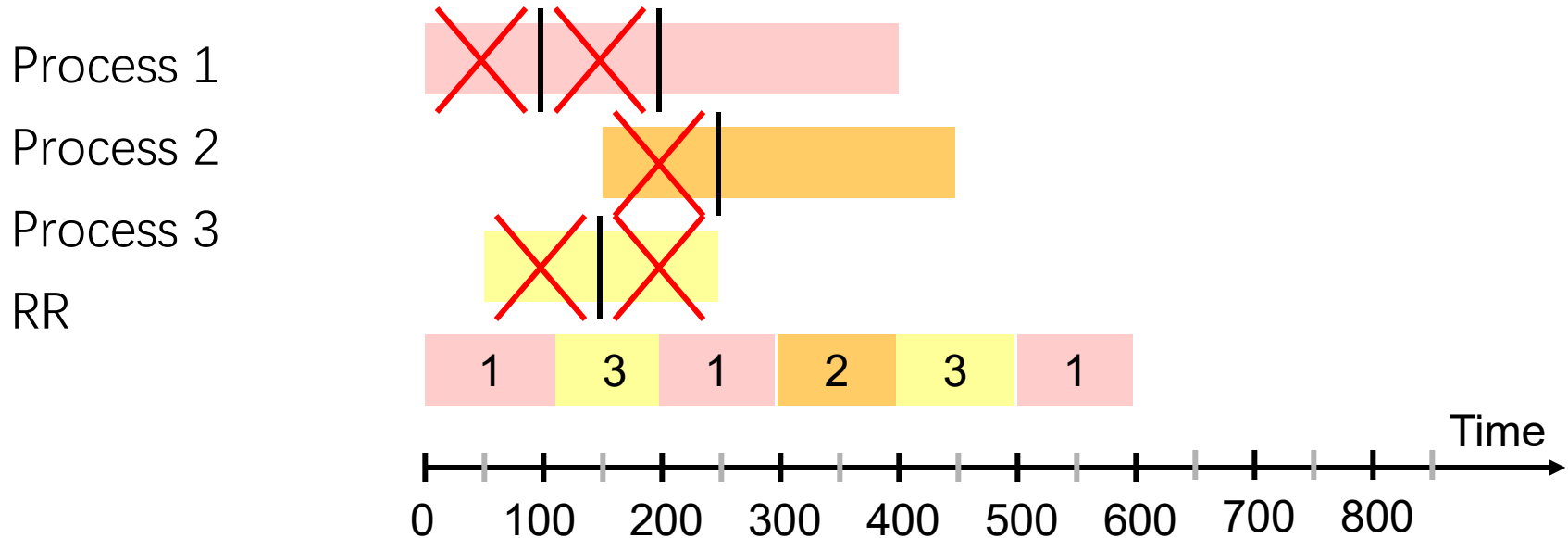
# Round Robin (Time slice = 100)



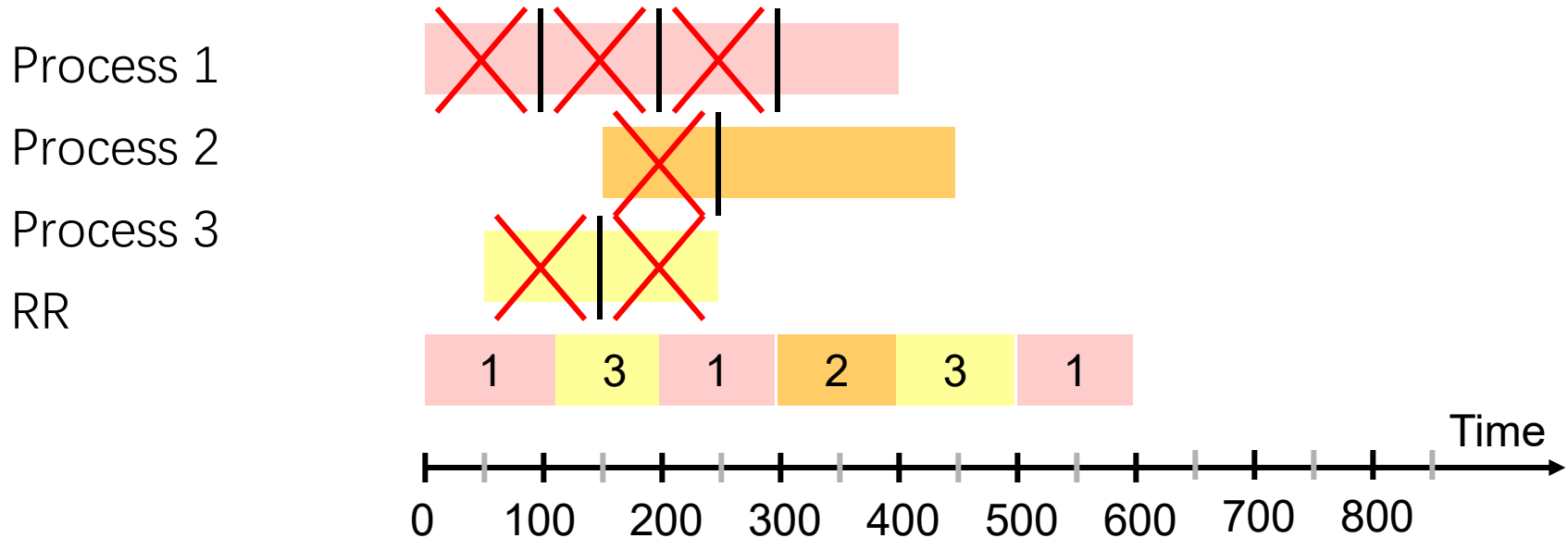
# Round Robin (Time slice = 100)



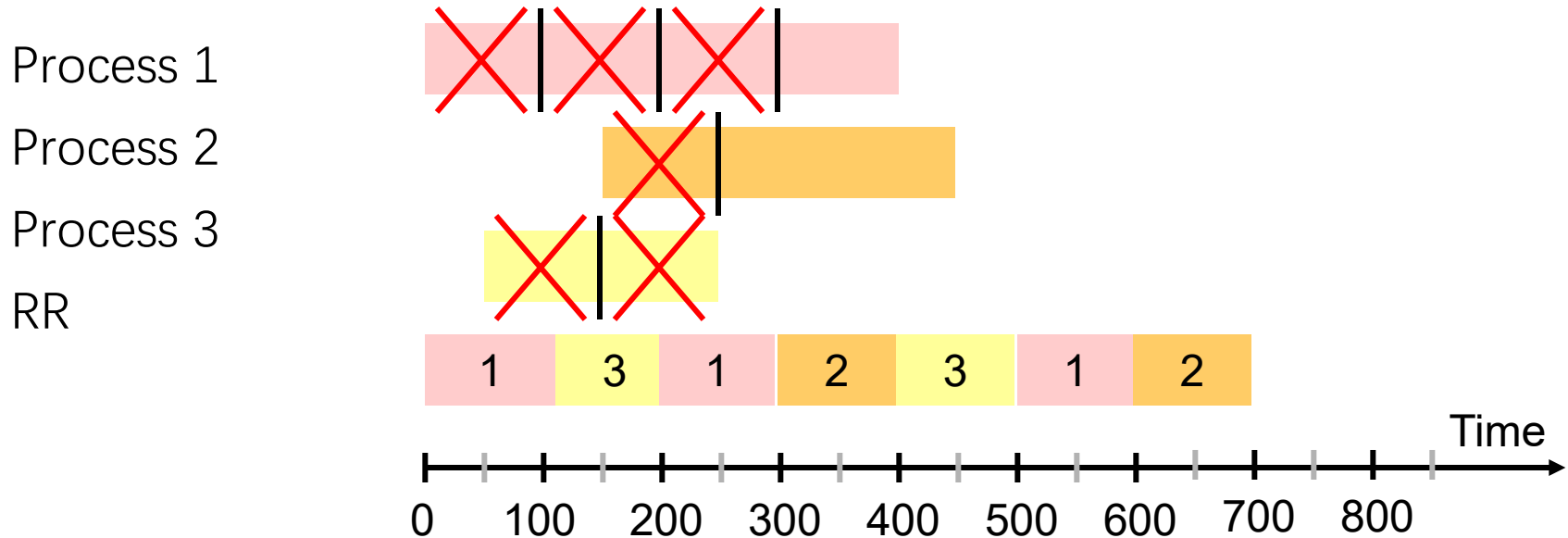
# Round Robin (Time slice = 100)



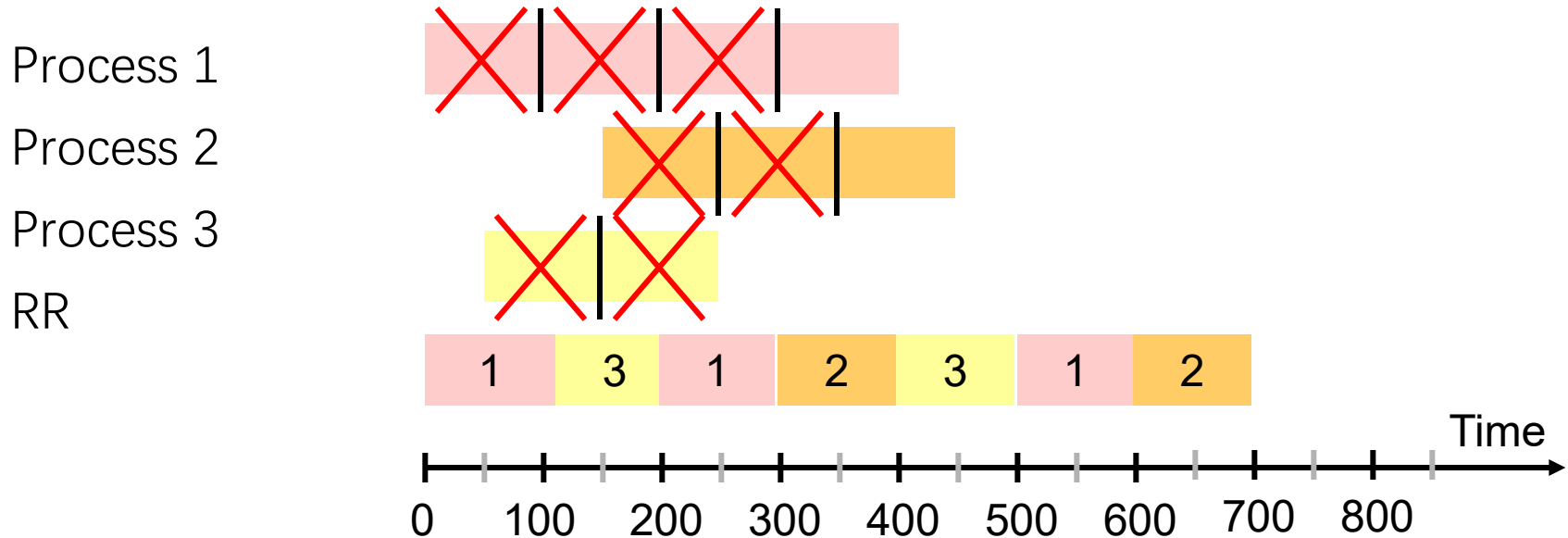
# Round Robin (Time slice = 100)



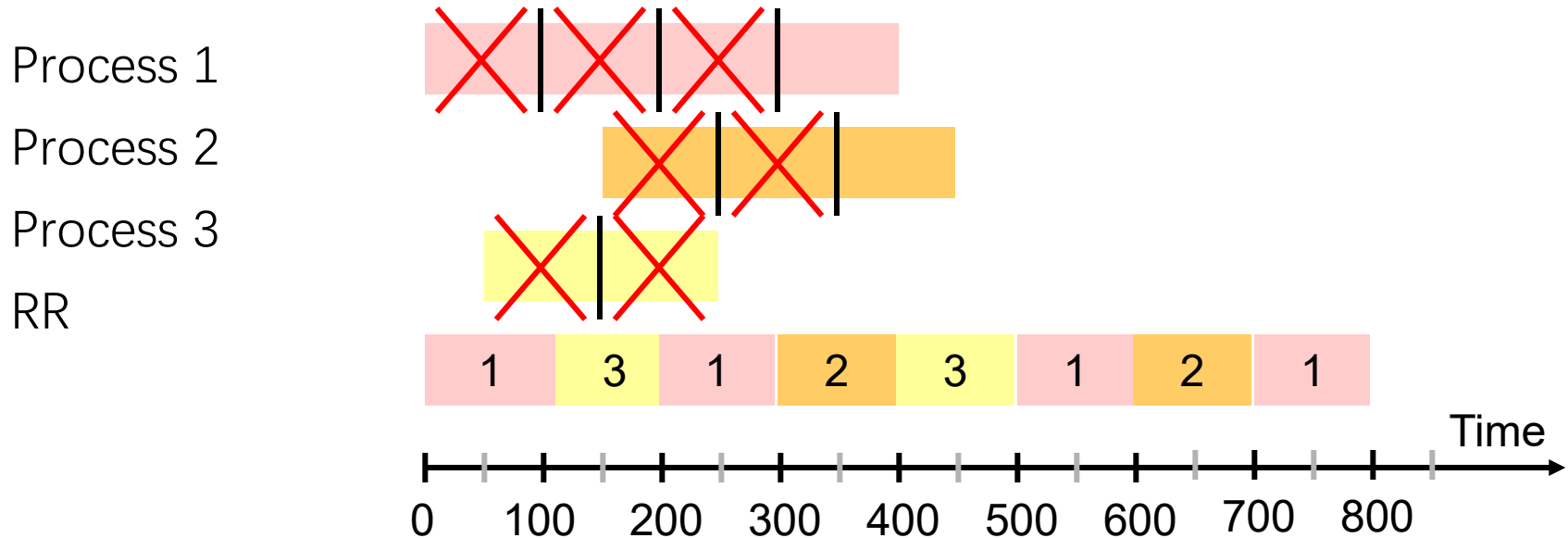
# Round Robin (Time slice = 100)



# Round Robin (Time slice = 100)



# Round Robin (Time slice = 100)



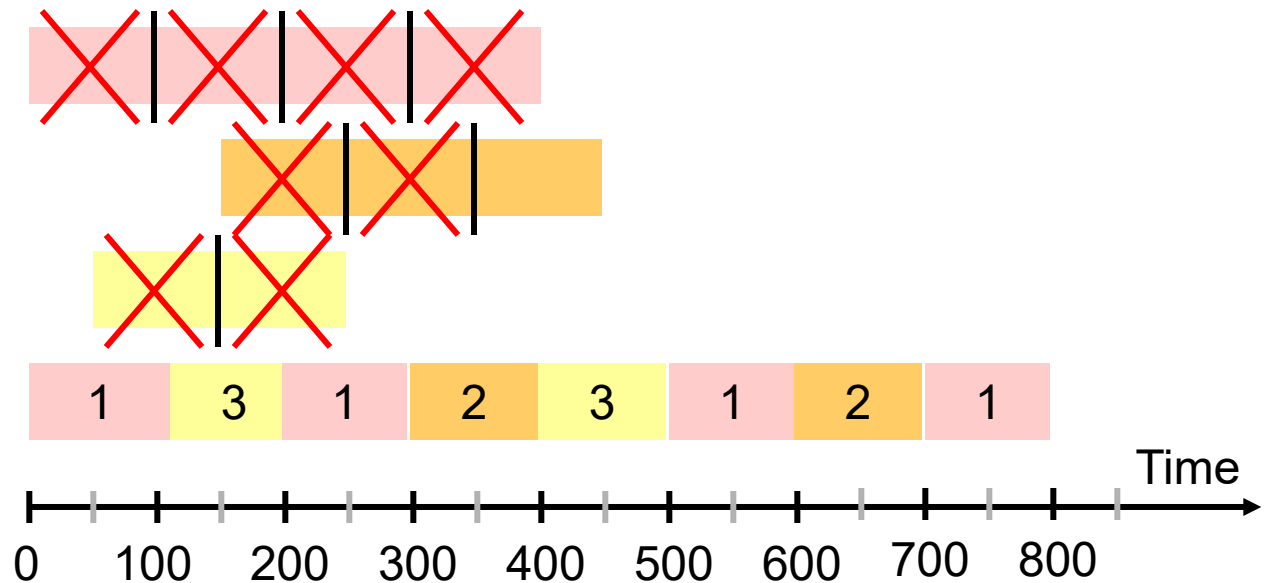
# Round Robin (Time slice = 100)

Process 1

Process 2

Process 3

RR





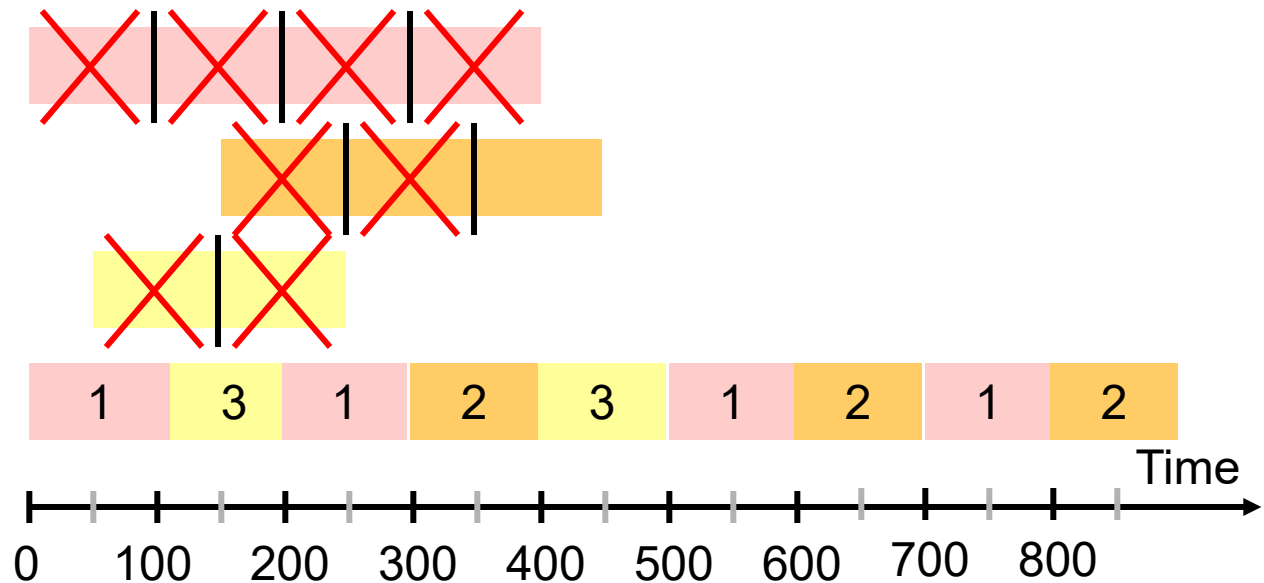
# Round Robin (Time slice = 100)

Process 1

Process 2

Process 3

RR



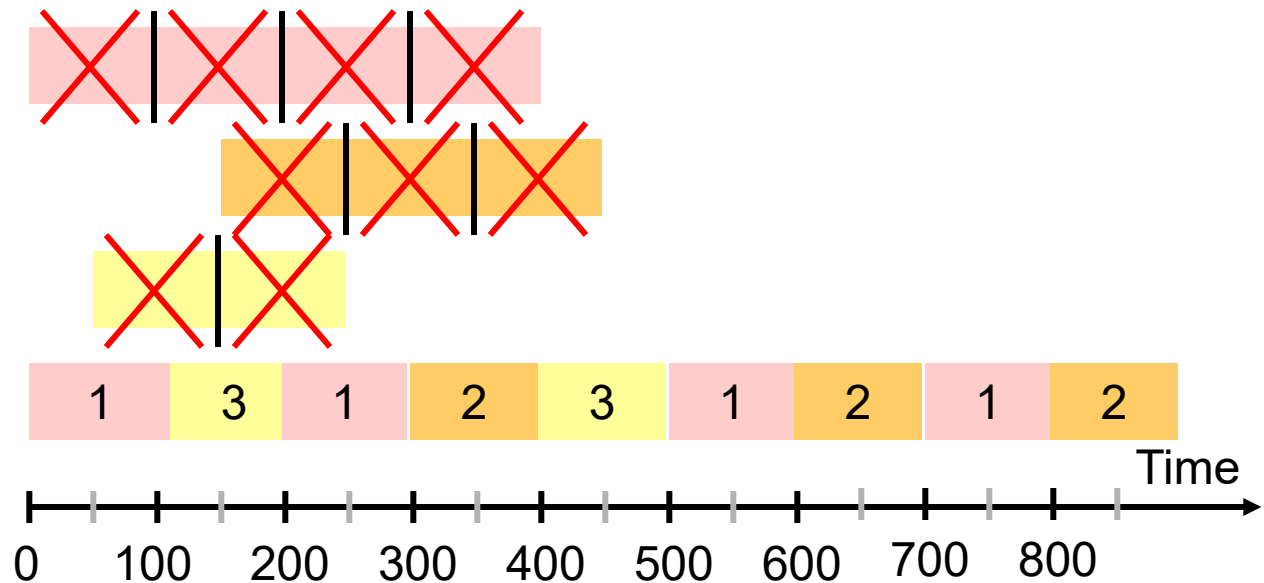
# Round Robin (Time slice = 100)

Process 1

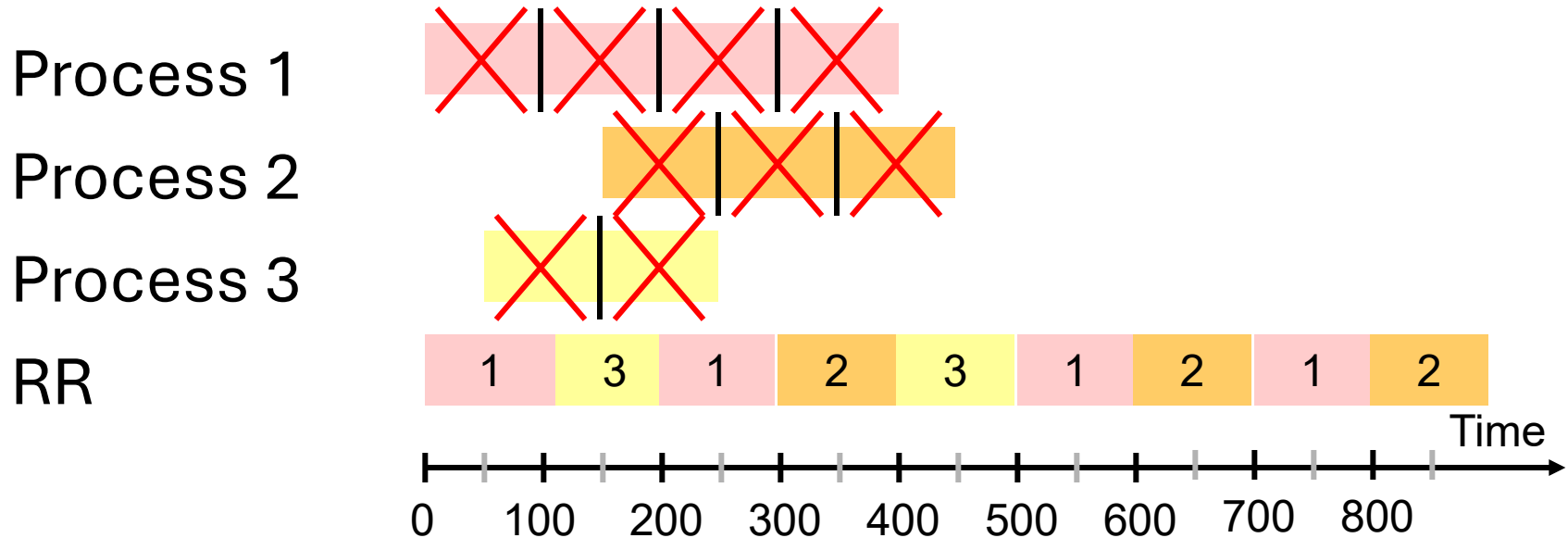
Process 2

Process 3

RR



# Round Robin (Time slice = 100)

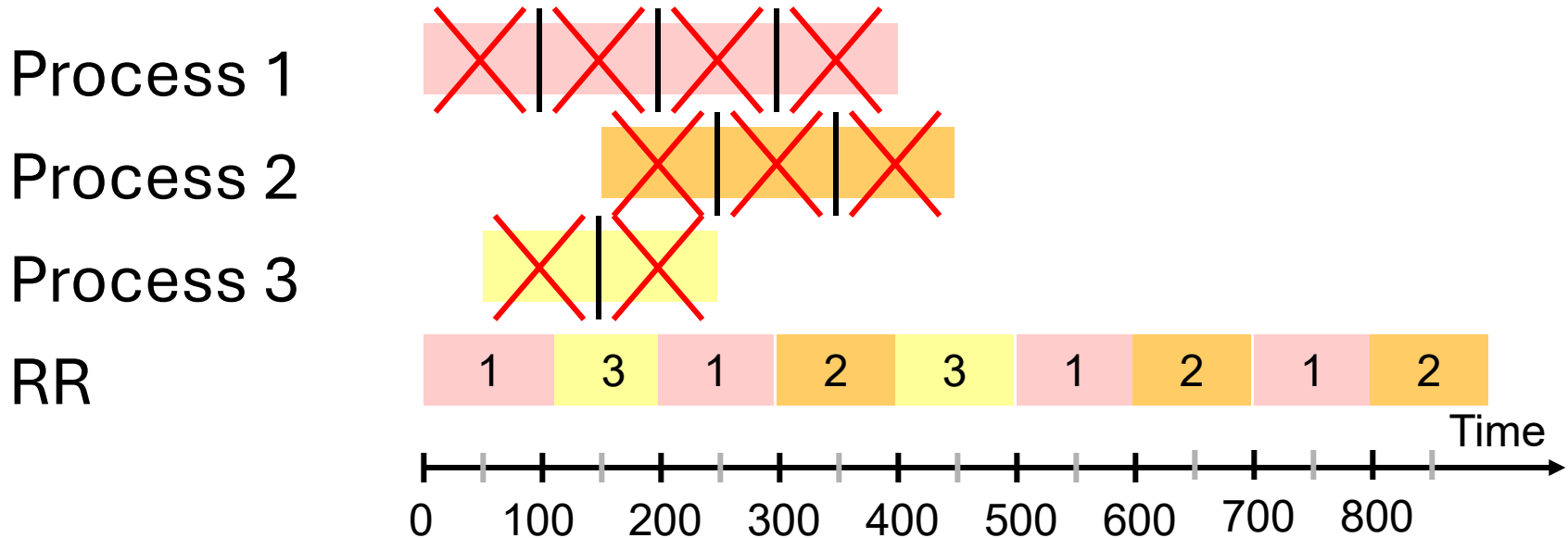


Response time for process 1: 0

Response time for process 2:  $300 - 150 = 150$

Response time for process 3:  $100 - 50 = 50$

# Round Robin (Time slice = 100)

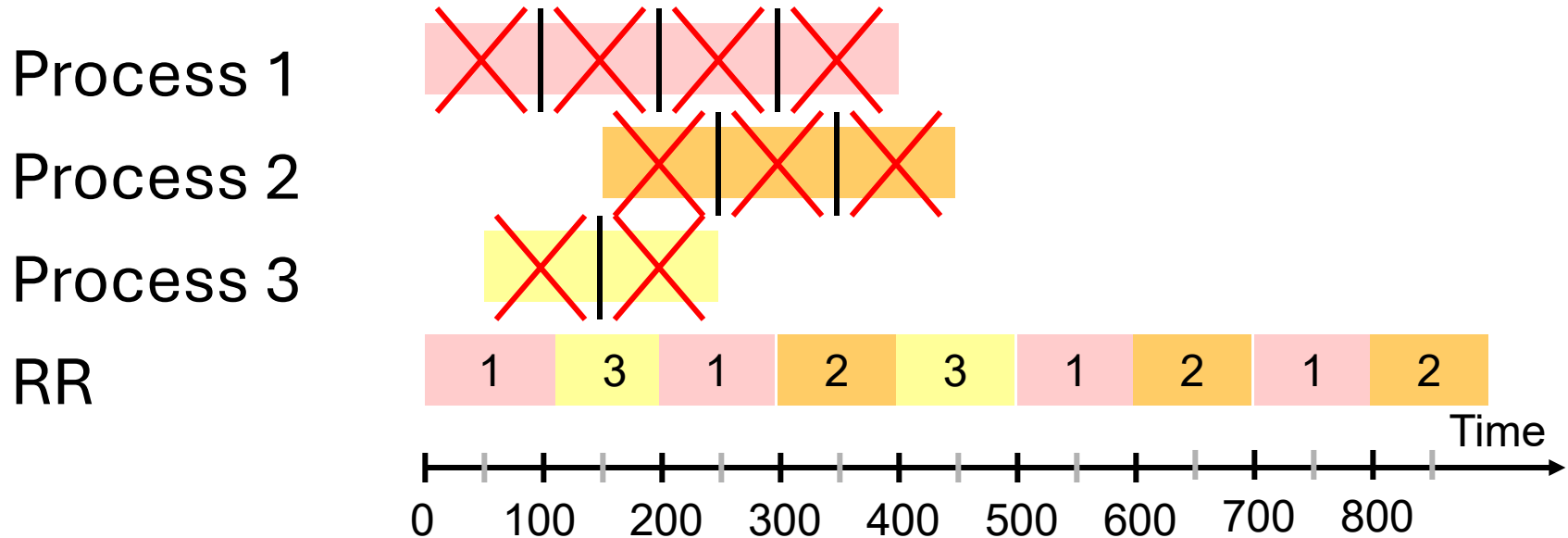


Wait time for process 1:  $0 + (200 - 100) + (500 - 300) + (700 - 600) = 400$

Wait time for process 2:  $(300 - 150) + (600 - 400) + (800 - 700) = 450$

Wait time for process 3:  $(100 - 50) + (400 - 200) = 250$

# Round Robin (Time slice = 100)



Turnaround time for process 1:  $800 - 0 = 800$

Turnaround time for process 2:  $900 - 150 = 750$

Turnaround time for process 3:  $500 - 50 = 450$

# FIFO vs. Round Robin

- With zero-cost context switch, is RR always better than FIFO?

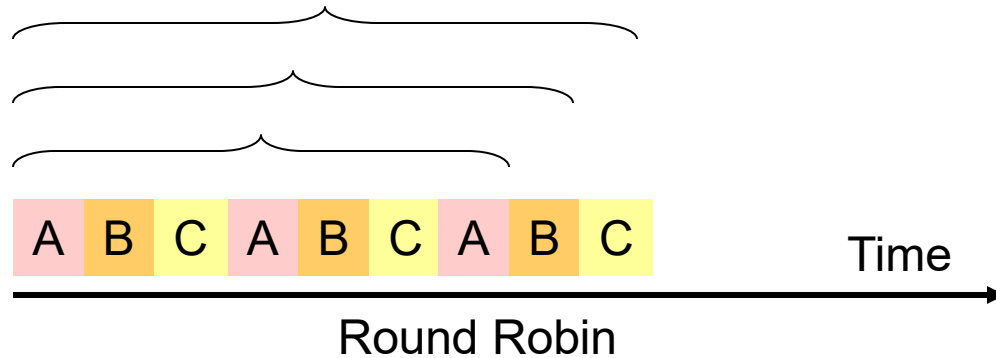
# FIFO vs. Round Robin

- Suppose we have three jobs of equal length

turnaround time of C

turnaround time of B

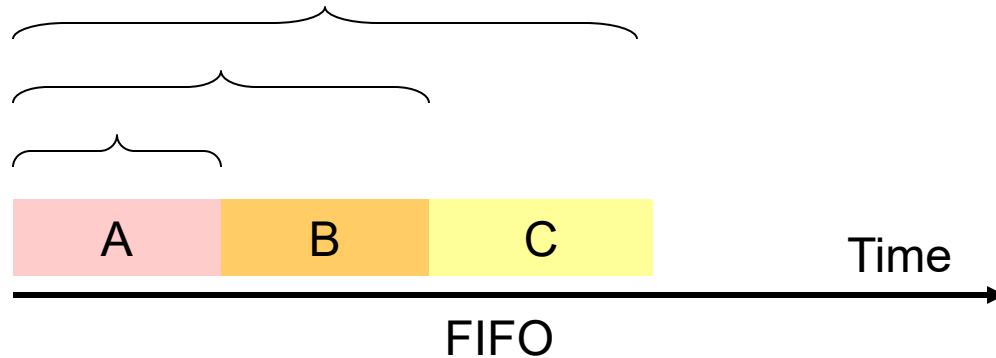
turnaround time of A



turnaround time of C

turnaround time of B

turnaround time of A



# FIFO vs. Round Robin

- Round Robin
  - + Shorter response time
  - + Fair sharing of CPU
  - Not all jobs are preemptive
  - Not good for jobs of the same length



# Shortest Job First (SJF)

- *SJF* runs whatever job puts the least demand on the CPU, also known as *STCF (shortest time to completion first)*
  - + Provably optimal
  - + Great for short jobs
  - + Small degradation for long jobs
- Real life example: supermarket express checkouts

# SJF Illustrated

turnaround time of C

turnaround time of B

turnaround time of A

wait time of C

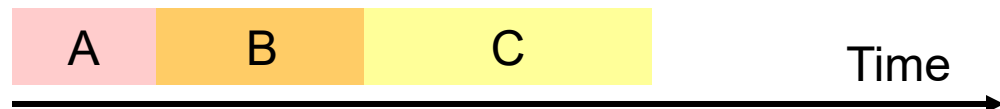
wait time of B

wait time of A = 0

response time of C

response time of B

response time of A = 0



Shortest Job First

# Shortest Remaining Time First (SRTF)

- *SRTF*: a preemptive version of SJF
  - If a job arrives with a shorter time to completion, SRTF preempts the CPU for the new job
  - Also known as *SRTCF (shortest remaining time to completion first)*
  - Generally used as the base case for comparisons

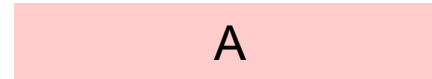
# SJF and SRTF vs. FIFO and Round Robin

- If all jobs are the same length, SJF → FIFO
  - FIFO is the best you can do
- If jobs have varying length
  - Short jobs do not get stuck behind long jobs under SRTF

# A More Complicated Scenario (Arrival Times = 0)

- Process A (6 units of CPU request)

- 100% CPU
- 0% I/O



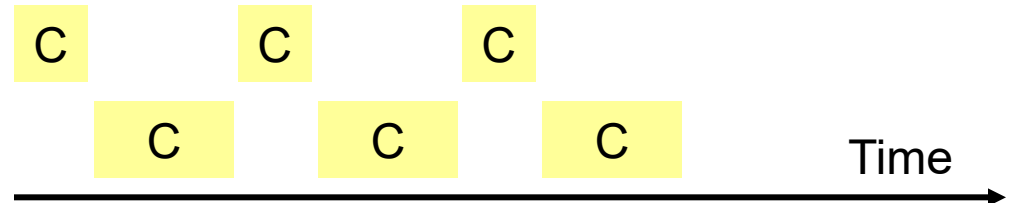
- Process B (6 units of CPU request)

- 100% CPU
- 0% I/O



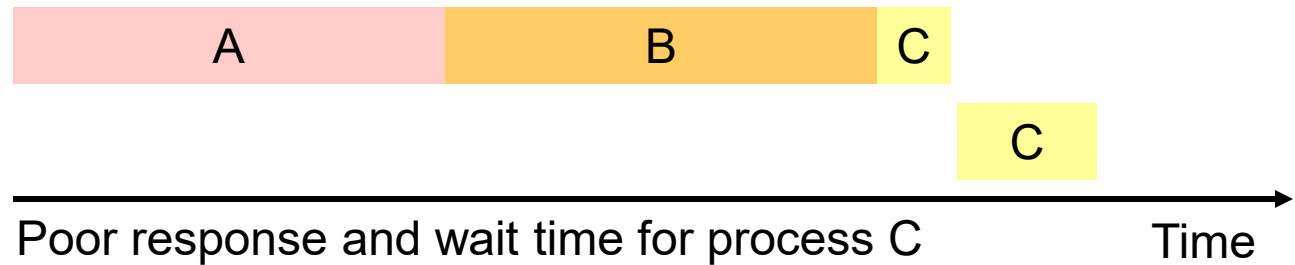
- Process C (infinite loop)

- 33% CPU
- 67% I/O

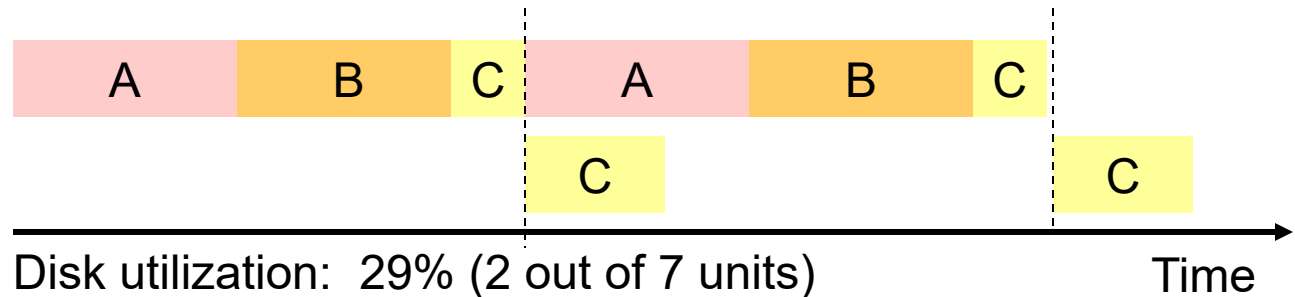


# A More Complicated Scenario

- FIFO
  - CPU
  - I/O



- Round Robin with time slice = 3 units
  - CPU
  - I/O

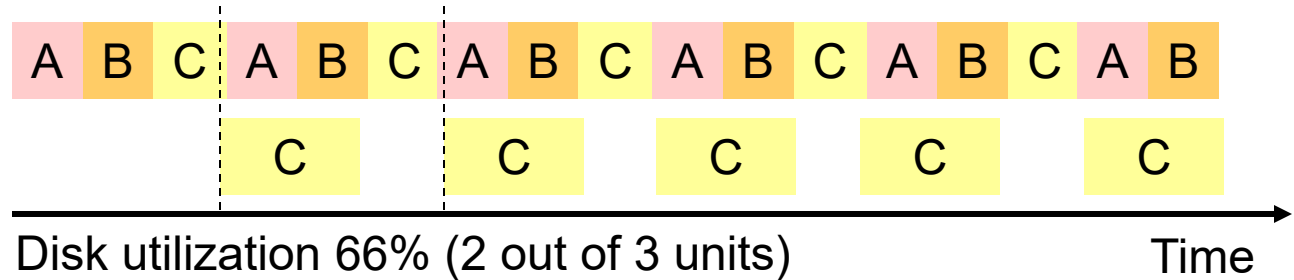


**Important**

# A More Complicated Scenario

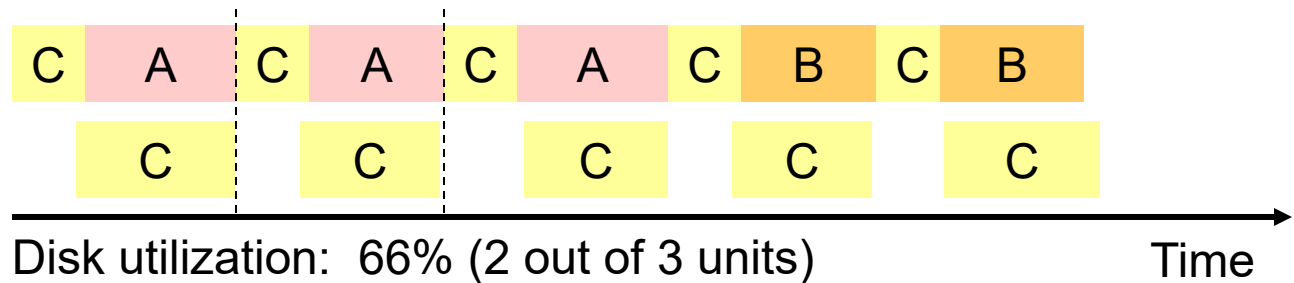
- Round Robin with time slice = 1 unit

- CPU
- I/O



- SRTCF

- CPU
- I/O



# Drawbacks of Shortest Job First

- *Starvation*: constant arrivals of short jobs can keep long ones from running
- There is no way to know the completion time of jobs (most of the time)
  - Some solutions
    - Ask the user, who may not know any better
    - If a user cheats, the job is killed



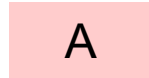
# Priority Scheduling (Multilevel Queues)

- *Priority scheduling*: The process with the highest priority runs first

- Priority 0:



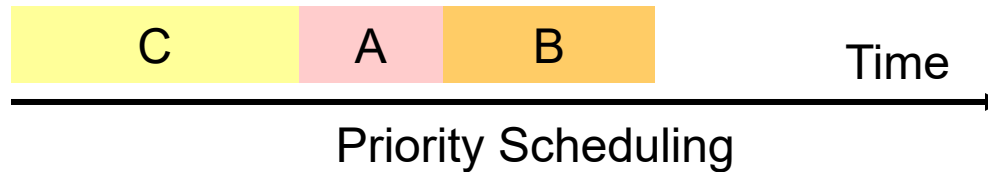
- Priority 1:



- Priority 2:



- Assume that low numbers represent high priority



# Priority Scheduling

## + Generalization of SJF

- With SJF, higher priority is inversely proportionally to requested\_CPU\_time

## - Starvation

- To prevent starvation, mechanisms like aging can be used, where the priority of a process increases the longer it waits.

# Multilevel Feedback Queues

- *Multilevel feedback queues* use multiple queues with different priorities
  - Round robin at each priority level
  - Run highest priority jobs first
  - Once those finish, run next highest priority, etc
  - Jobs start in the highest priority queue
  - If time slice expires, drop the job by one level
  - If time slice does not expire, push the job up by one level

# Multilevel Feedback Queues

time = 0

- Priority 0 (time slice = 1):



- Priority 1 (time slice = 2):

- Priority 2 (time slice = 4):



# Multilevel Feedback Queues

time = 1

- Priority 0 (time slice = 1):



- Priority 1 (time slice = 2):





- Priority 2 (time slice = 4):



# Multilevel Feedback Queues

time = 2

- Priority 0 (time slice = 1): 
- Priority 1 (time slice = 2): 
- Priority 2 (time slice = 4):



# Multilevel Feedback Queues

time = 3

- Priority 0 (time slice = 1):
- Priority 1 (time slice = 2):
- Priority 2 (time slice = 4):



# Multilevel Feedback Queues

time = 3

- Priority 0 (time slice = 1):
- Priority 1 (time slice = 2):
- Priority 2 (time slice = 4):





suppose process A is blocked on an I/O





# Multilevel Feedback Queues

time = 3



- Priority 0 (time slice = 1): 
- Priority 1 (time slice = 2): 
- Priority 2 (time slice = 4):

suppose process A is blocked on an I/O



# Multilevel Feedback Queues

time = 5

- Priority 0 (time slice = 1): 
- Priority 1 (time slice = 2): 
- Priority 2 (time slice = 4):

suppose process A is returned from an I/O



# Multilevel Feedback Queues

time = 6

- Priority 0 (time slice = 1):
- Priority 1 (time slice = 2):
- Priority 2 (time slice = 4):

C

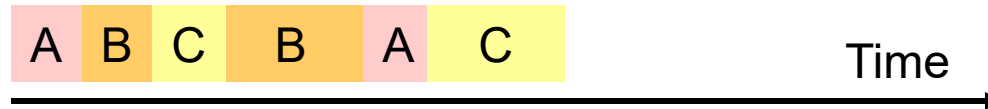


# Multilevel Feedback Queues

time = 8

- Priority 0 (time slice = 1):
- Priority 1 (time slice = 2):
- Priority 2 (time slice = 4):

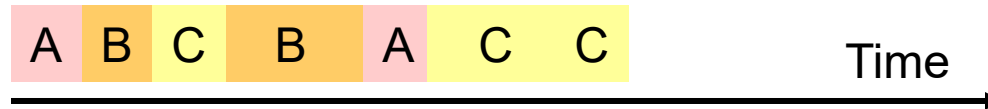
C



# Multilevel Feedback Queues

time = 9

- Priority 0 (time slice = 1):
- Priority 1 (time slice = 2):
- Priority 2 (time slice = 4):



# Multilevel Feedback Queues

- Approximates SRTF
  - A CPU-bound job drops like a rock
  - I/O-bound jobs stay near the top
  - Still unfair for long running jobs
  - Counter-measure: *Aging*
    - Increase the priority of long running jobs if they are not serviced for a period of time
    - Tricky to tune aging

# Lottery Scheduling

- *Lottery scheduling* is an adaptive scheduling approach to address the fairness problem
  - Each process owns some tickets
  - On each time slice, a ticket is randomly picked
  - On average, the allocated CPU time is proportional to the number of tickets given to each job

# Lottery Scheduling

- To approximate SJF, short jobs get more tickets
- To avoid starvation, each job gets at least one ticket



# Lottery Scheduling Example

- short jobs: 10 tickets each
- long jobs: 1 ticket each

# short jobs/# long jobs	% of CPU for each short job	% of CPU for each long job
1/1	91% (10/11)	9% (1/11)
0/2	0%	50%
2/0	50%	0%
10/1	10% (10/101)	1% (1/101)
1/10	50% (10/20)	5% (1/20)

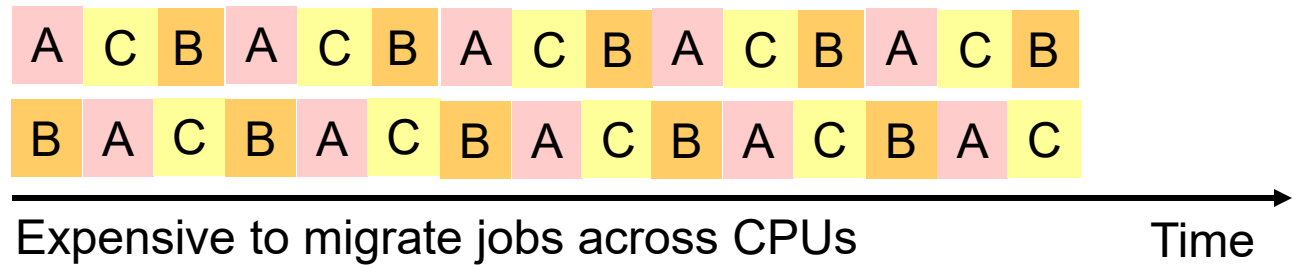
# Pros and Cons of Lottery Scheduling

- + Good for coordinating computers with different computing power
- + Good for controlling the schedules for child processes
- Not as good for real-time systems

# Multicore Scheduling

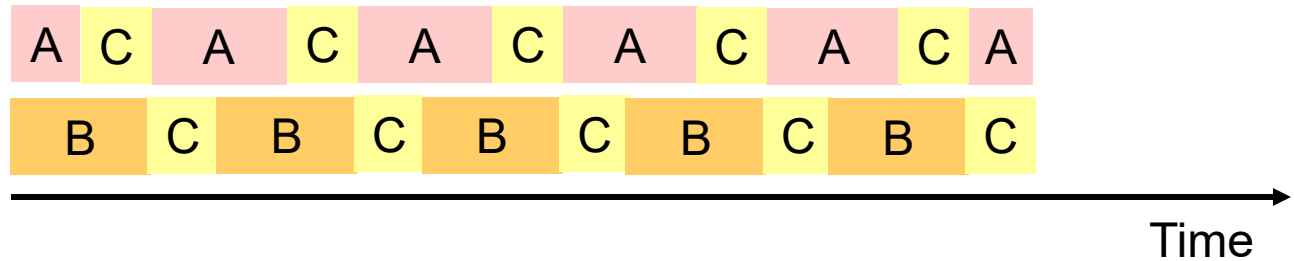
- Single-queue multiprocessor scheduling (SQMS)

- CPU1
- CPU2



- Another SQMS (Poor *CPU affinity*)

- CPU1
- CPU2

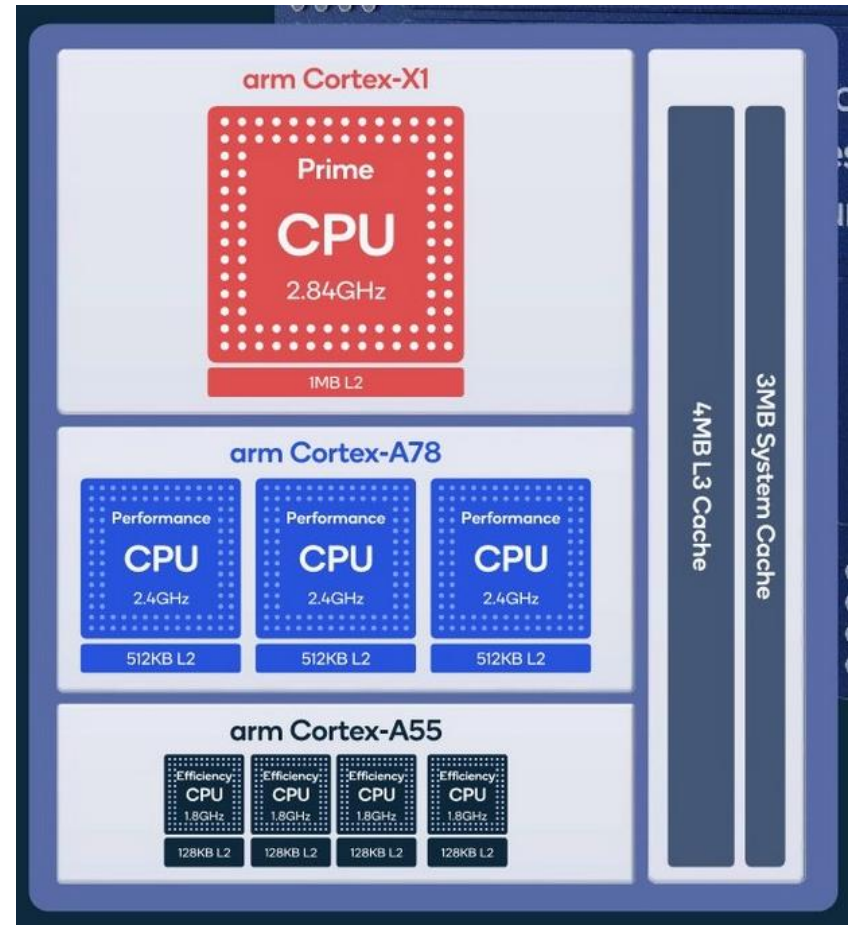


# More Schedulers

- Multi-queue scheduling
- $O(1)$  scheduler
- Completely Fair Scheduler (CFS)

# Real World

- Big.LITTLE
  - Snapdragon 888
- Others
  - The physical distance between CPUs on the circuit board
  - Power
  - ...



# Takeaways

- OS is a state machine.
- Process, Thread, Address Space
- Thread Dispatch Loop
- Amdahl's Law
- CPU Scheduling