CPSC 457 T03/T04

Week 5 Day 1

Xining Chen

Agenda

- CPU Scheduling
  - FCFS
  - RR
  - SJF
  - SRTN
  - Simulation loop
- Deadlock detection
- Assignment 4

CPU Scheduling

- Non-preemptive – Context switch happens only voluntarily
  - Run process until it blocks
  - Ex:// First Come First Serve (FCFS)
- Preemptive – Context switch can happen without thread cooperation
  - Direct/indirect result of some event
  - Ex:// Shortest Remaining Time Next (SRTN)
- Preemptive time-sharing – special case of preemptive
  - Periodic context switches (time-slice policy)
  - Ex:// Round Robin (RR)

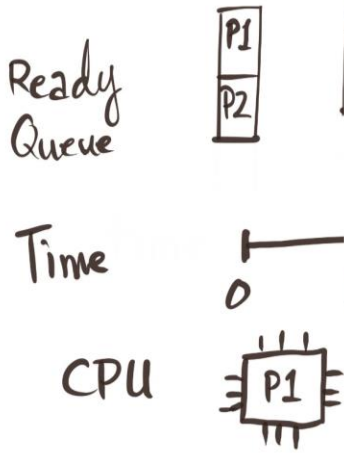
First Come First Serve (FCFS)

- Non-preemptive
- Uses a FIFO ready queue
- New jobs are appended to the ready queue
- When running process blocks, next process from ready queue starts to execute
- When process is unblocked, it’s appended to the ready queue

- Minimum number of context switches

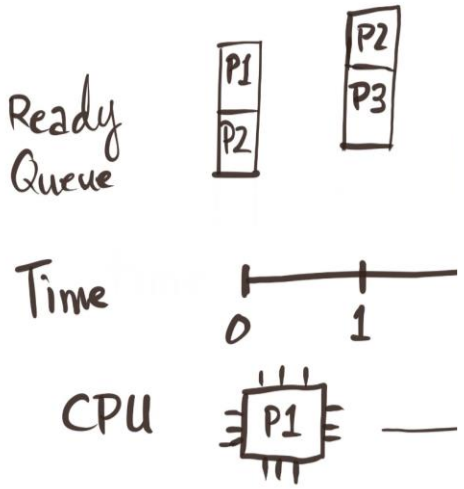
First Come First Serve (FCFS)

Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



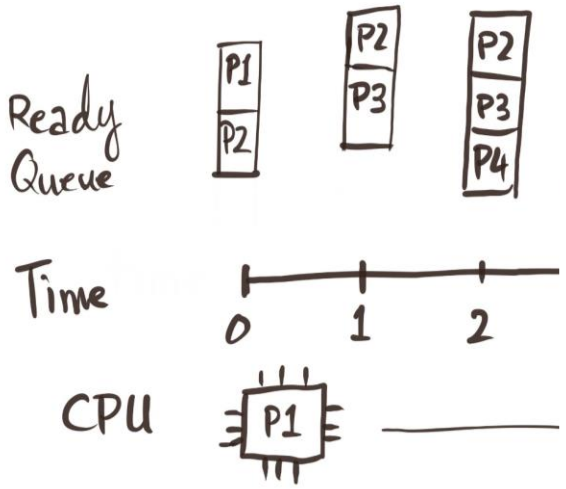
First Come First Serve (FCFS)

Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



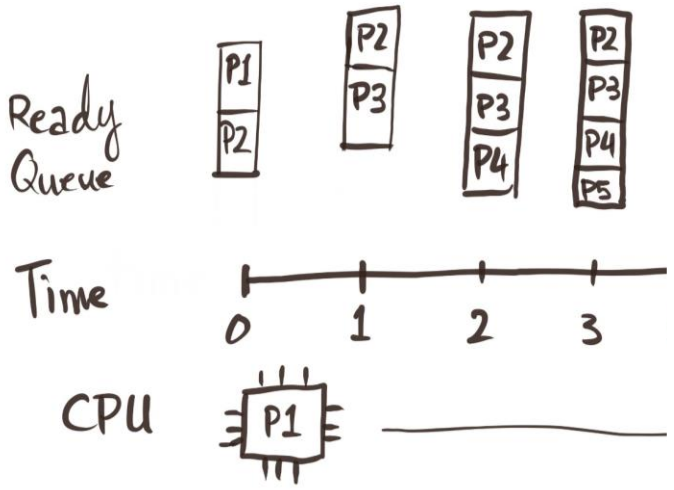
First Come First Serve (FCFS)

Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



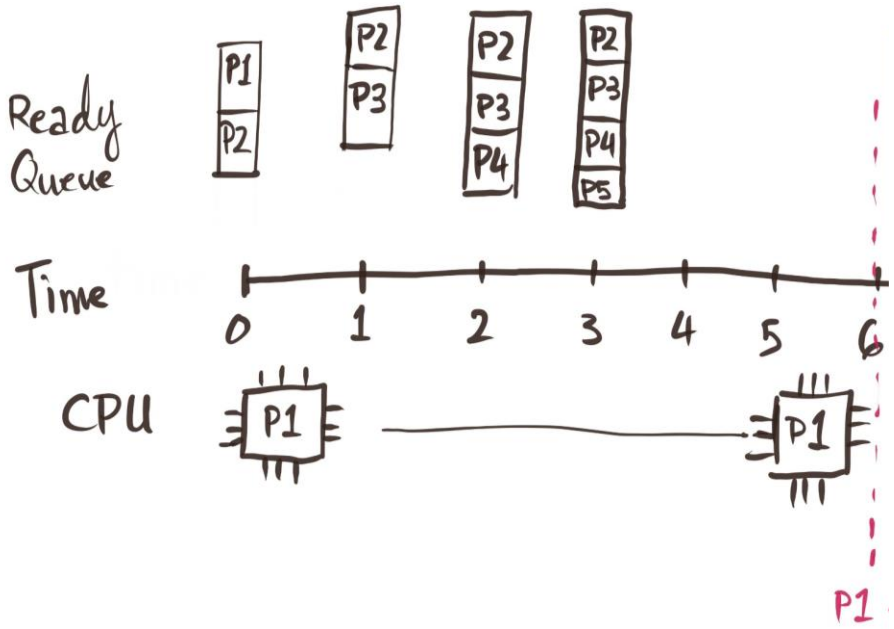
First Come First Serve (FCFS)

Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



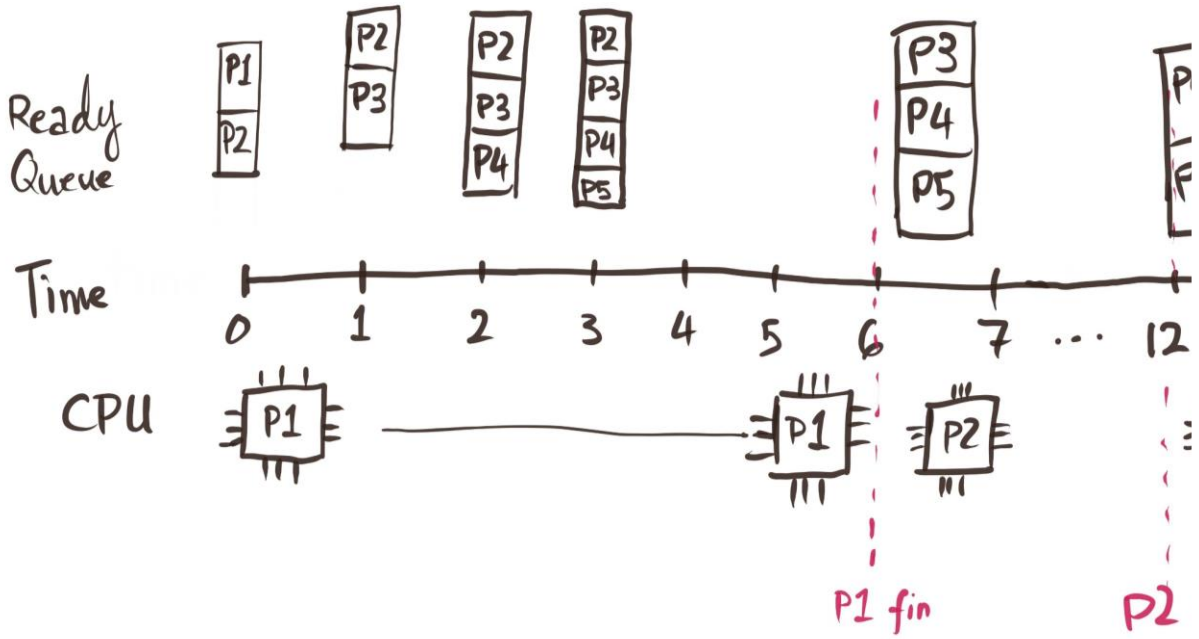
First Come First Serve (FCFS)

Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



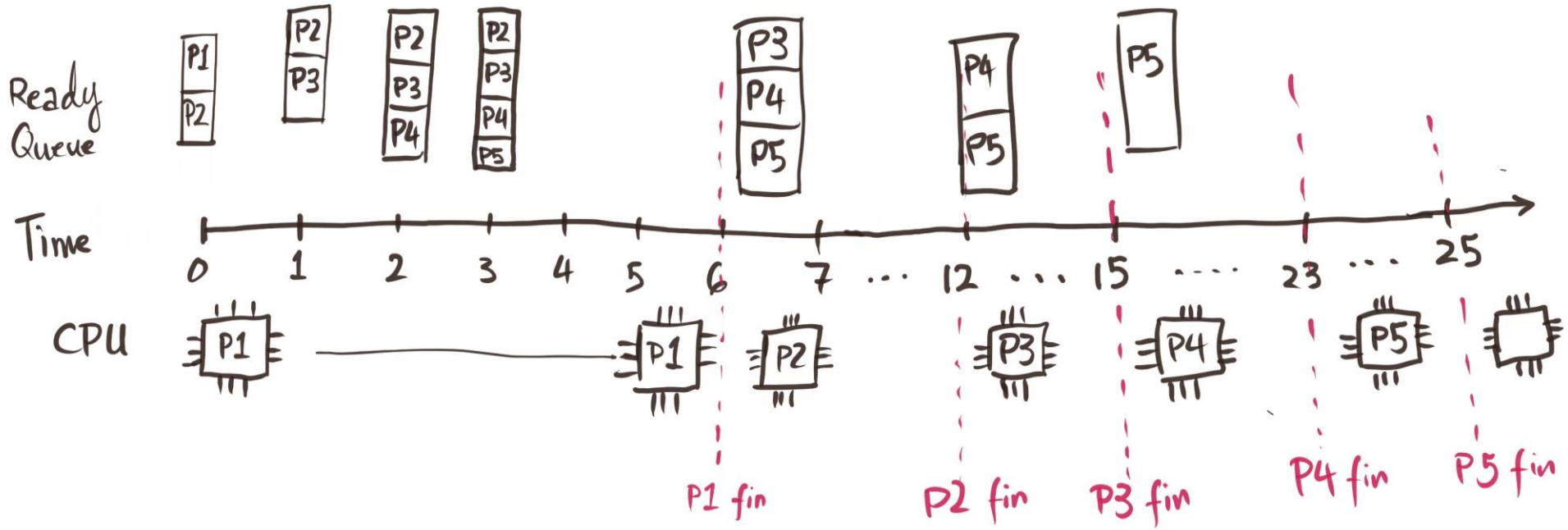
First Come First Serve (FCFS)

Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



First Come First Serve (FCFS)

Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



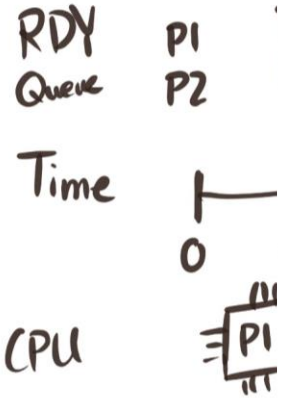
Round-Robin (RR)

- Preemptive version of FCFS
- Time slice (quantum)
- If running process exceeds the time slice, process is pre-empted (context switched)
- Preempted process goes back to ready queue
- If process completes / makes blocking call before time-slice is up, then next process in ready queue executes

Round-Robin (RR)

Suppose 3 s time slice

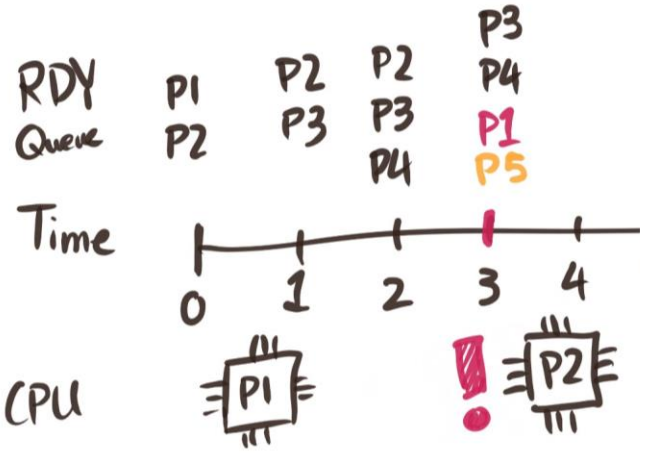
Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



Round-Robin (RR)

Suppose 3 s time slice

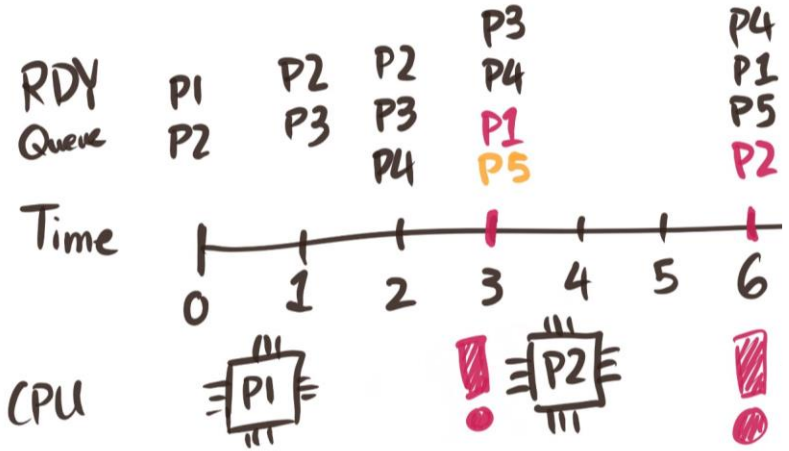
Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



Round-Robin (RR)

Suppose 3 s time slice

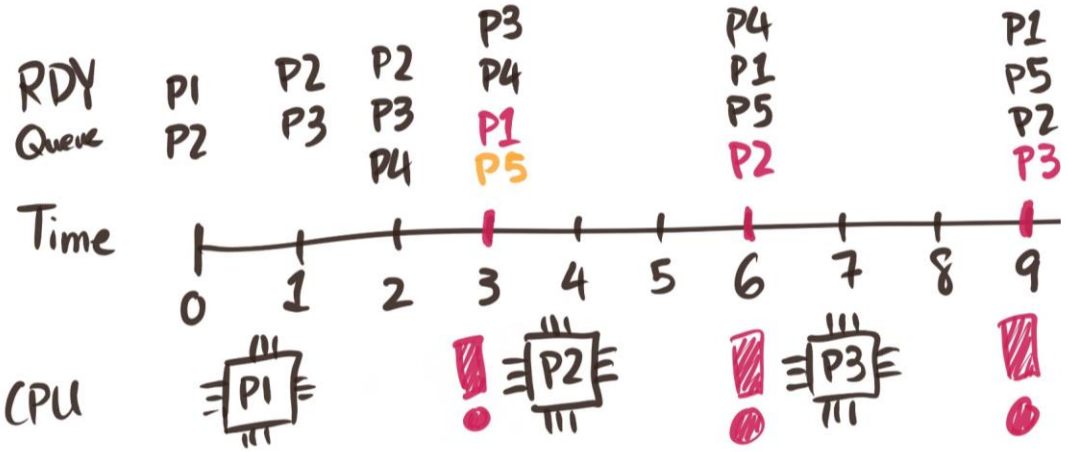
Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



Round-Robin (RR)

Suppose 3 s time slice

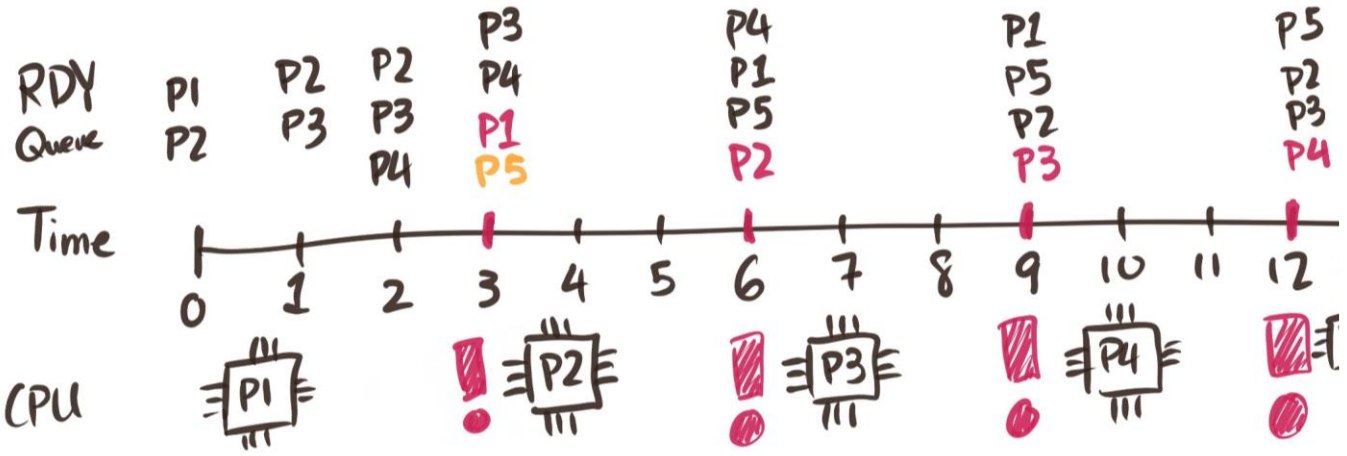
Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



Round-Robin (RR)

Suppose 3 s time slice

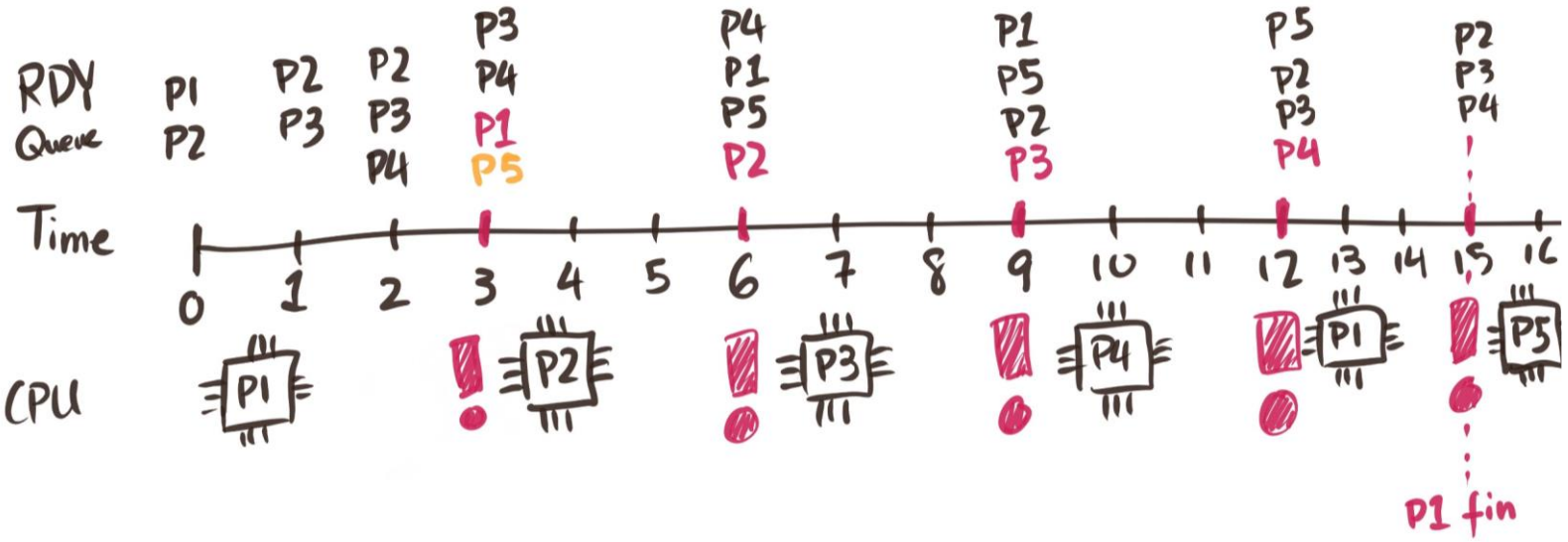
Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



Round-Robin (RR)

Suppose 3 s time slice

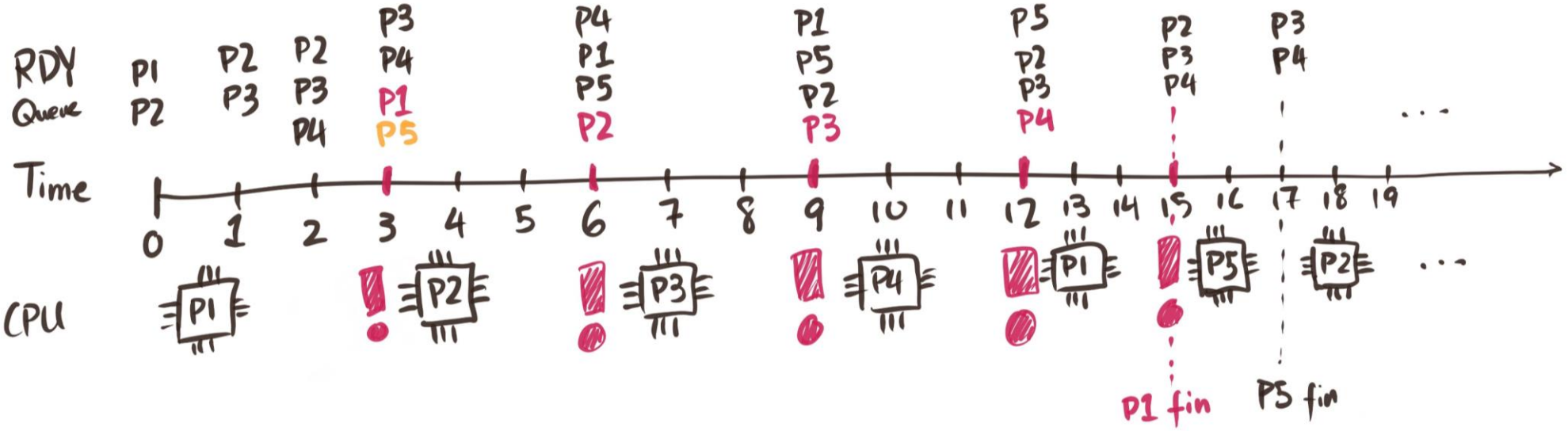
Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



Round-Robin (RR)

Suppose 3 s time slice

Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2

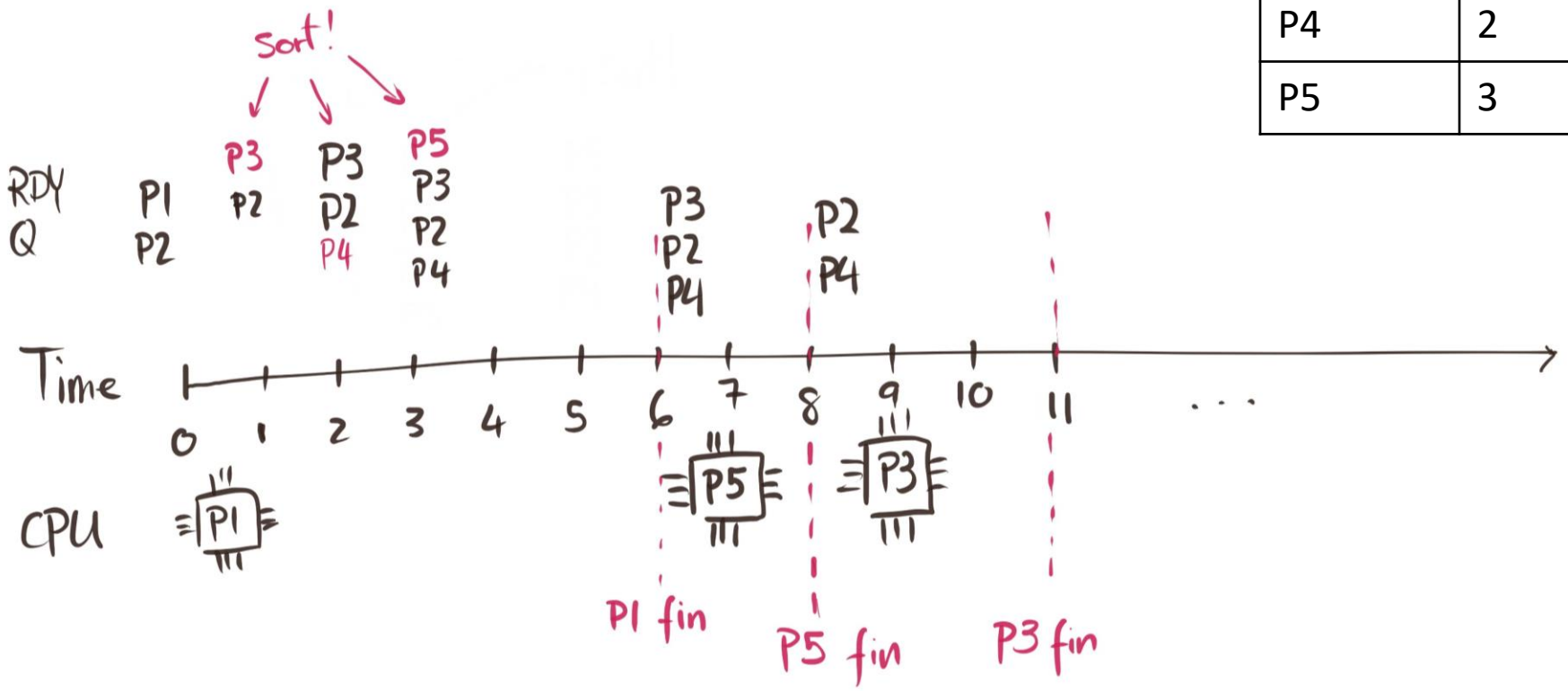


Shortest-Job-First (SJF)

- Non-preemptive
- Similar to FCFS, but sort ready queue by execution time
- Ties – resolve using FCFS

Shortest-Job-First (SJF)

Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



Shortest-Remaining-Time-Next (SRTN)

- Preemptive version of SJF
- Similar to SJF – except ready queue is sorted by remaining time
- pre-emption happens as a result of adding a job
- Optimal turnaround time

Simulation loop

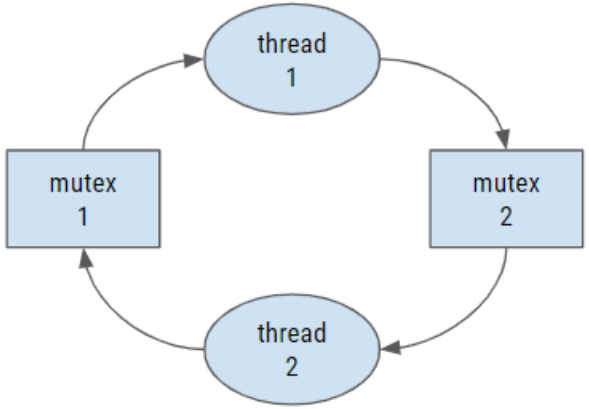
```
curr_time = 0
while(1) {
    if simulation done break
    …
    curr_time ++
}
```

```
curr_time = 0
jobs_remaining = size of job queue
while(1) {
    if jobs_remaining == 0 break
    if process in cpu is done
        mark process done
        set CPU idle
        jobs_remaining --
        continue
    if a new process arriving
        add new process to RQ
        continue
    if cpu is idle and RQ not empty
        move process from RQ to CPU
        continue
    execute one burst of job on CPU
    curr_time ++
}
```

Deadlocks

- graph with a set of vertices V and a set of edges E
- set of vertices V is partitioned into two subsets:
  - $P = \{P_1, P_2 \ldots P_n\}$, the set of all processes in the system, represented as ellipsoids 
  - $R = \{R_1, R_2 \ldots R_m\}$, the set of all resources in the system, represented as rectangles 
- request edge – directed edge $P_i \rightarrow R_j$

- assignment edge – directed edge $R_j \rightarrow P_i$



Deadlock Detection

- Cycle detection in Resource-Allocation graph
- Topological sort

Deadlock Detection

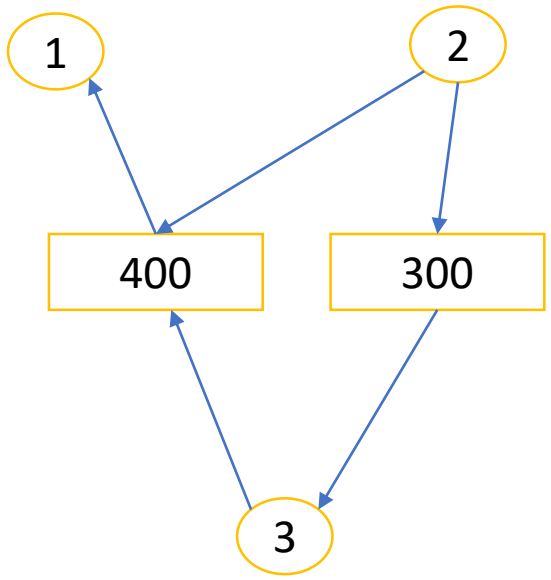
- Topological sort:
  - Need to keep track of “Need” / Request (Out-degree)
  - Need to keep track of “Have” (incoming nodes)

  “If I don’t need anything, I can execute and release my acquired resources”

  If I don’t have any outgoing edges, then I can be removed from adjacency list

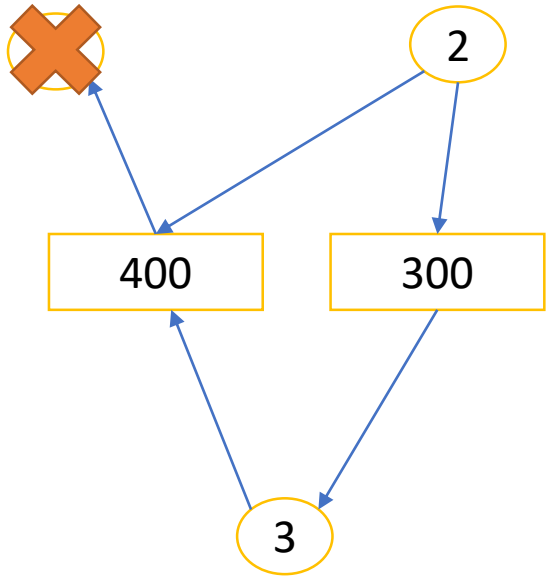
  Need to update out degree of all dependents (incoming nodes) every time something gets removed from the adjacency list

Deadlock Detection



Nodes	Incoming nodes	Outgoing degree
1	[400]	0
2	[]	2
400	[2,3]	1
300	[2]	1
3	[300]	1

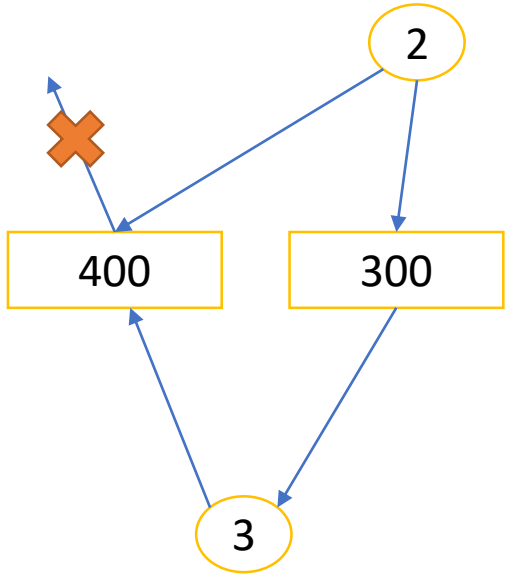
Deadlock Detection



Nodes	Incoming nodes	Outgoing degree
1	[400]	0
2	[]	2
400	[2,3]	1
300	[2]	1
3	[300]	1

Remove!

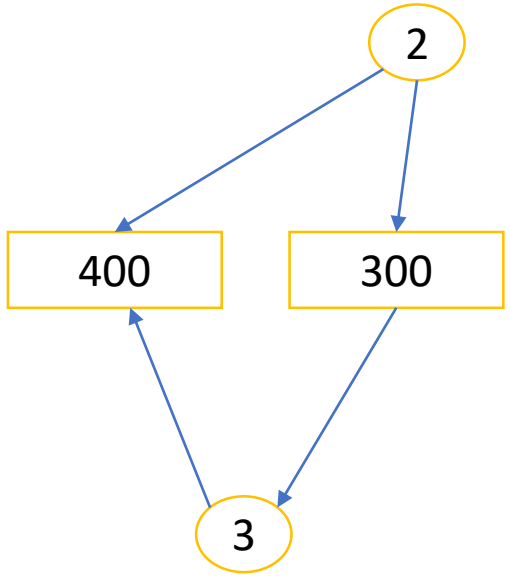
Deadlock Detection



Nodes	Incoming nodes	Outgoing degree
1	[400]	0
2	[]	2
400	[2,3]	0
300	[2]	1
3	[300]	1

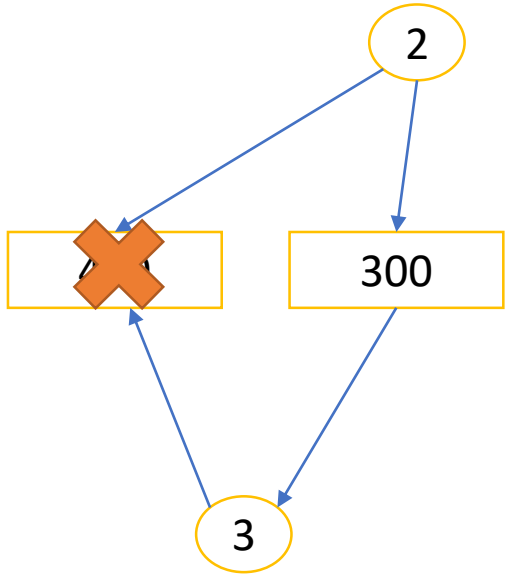
-1

Deadlock Detection



Nodes	Incoming nodes	Outgoing degree
2	[]	2
400	[2,3]	0
300	[2]	1
3	[300]	1

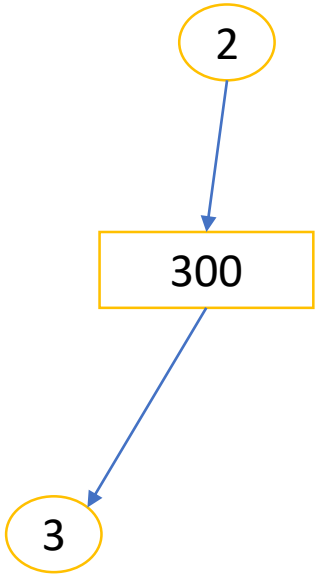
Deadlock Detection



Nodes	Incoming nodes	Outgoing degree
2	[]	2
400	[2,3]	0
300	[2]	1
3	[300]	1

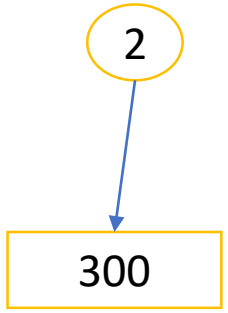
Remove!

Deadlock Detection



Nodes	Incoming nodes	Outgoing degree	
2	[]	1	-1
300	[2]	1	
3	[300]	0	-1

Deadlock Detection



Nodes	Incoming nodes	Outgoing degree
2	[]	1
300	[2]	0

Deadlock Detection

Nodes	Incoming nodes	Outgoing degree
2	[]	0

Deadlock Detection

No Deadlock! ☺

Nodes	Incoming nodes	Outgoing degree

Assignment 4 – Question1

- Detect deadlock using topological sort
  1. Process each line of the file correctly → create adjacency list
  2. Run topological sort
  3. After topological sort finishes
     - if adjacency list is empty: no cycle (no deadlock)
     - if adjacency list is not empty: cycle exists (deadlock)

Assignment 4 – Question1

One possible implementation:

- Create a class called Node
  - It will contain its “id” (name), out degree, and a vector of incoming nodes.
  - It should have at least 2 methods
    1. A method for adding incoming node to it’s incoming nodes vector
    2. A method for removing nodes from the incoming nodes vector
    3. Optional: a print function for printing a node

- Use a vector for the adjacency list to store the node objects.

Another possible implementation:

- Use a 2D vector

And another possible implementation:

- Use map or unordered_map in C++

Assignment 4 – Question1

- Things to watch out for:
    1. How to distinguish between a process and a resource?

Next time

- An optimization for topological sort
- Open tutorial for Assignment 4 help