CPSC 457 T01/T04

Xining Chen

Agenda

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

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)

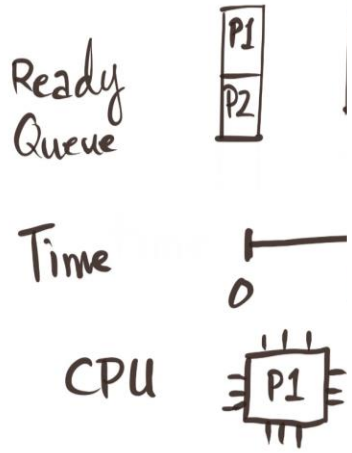
First Come First Serve (FCFS)

- 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

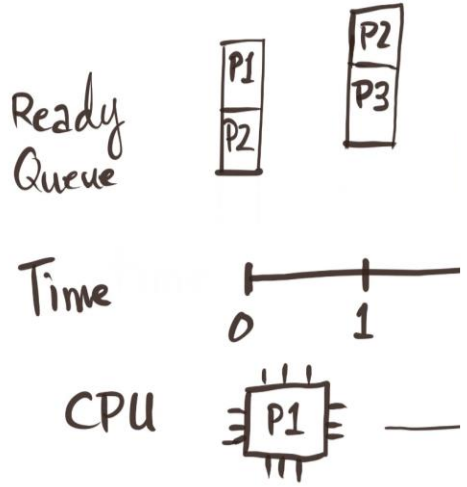
First Come First Serve (FCFS)

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



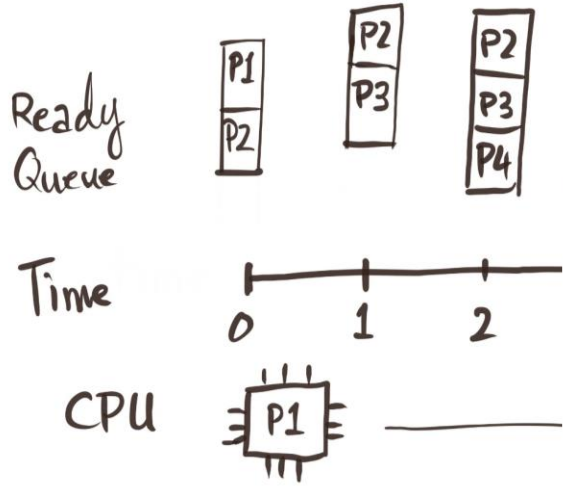
First Come First Serve (FCFS)

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



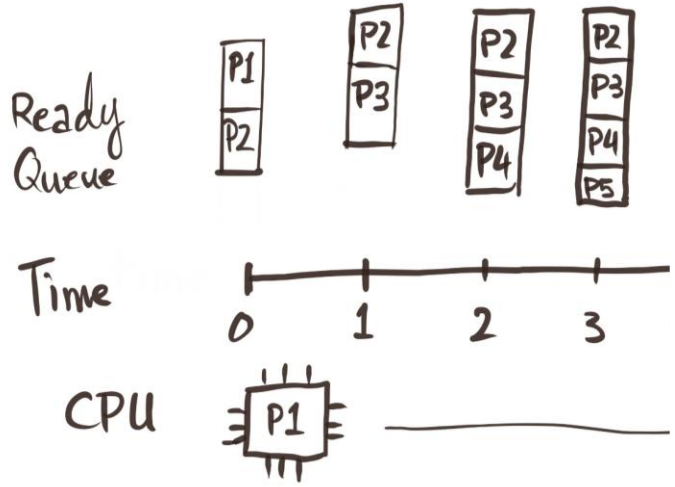
First Come First Serve (FCFS)

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



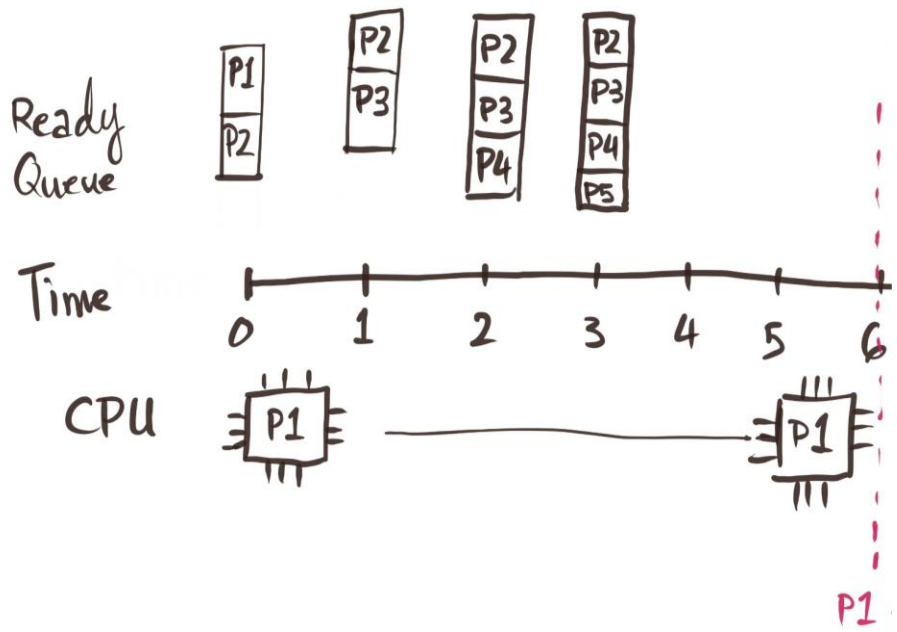
First Come First Serve (FCFS)

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



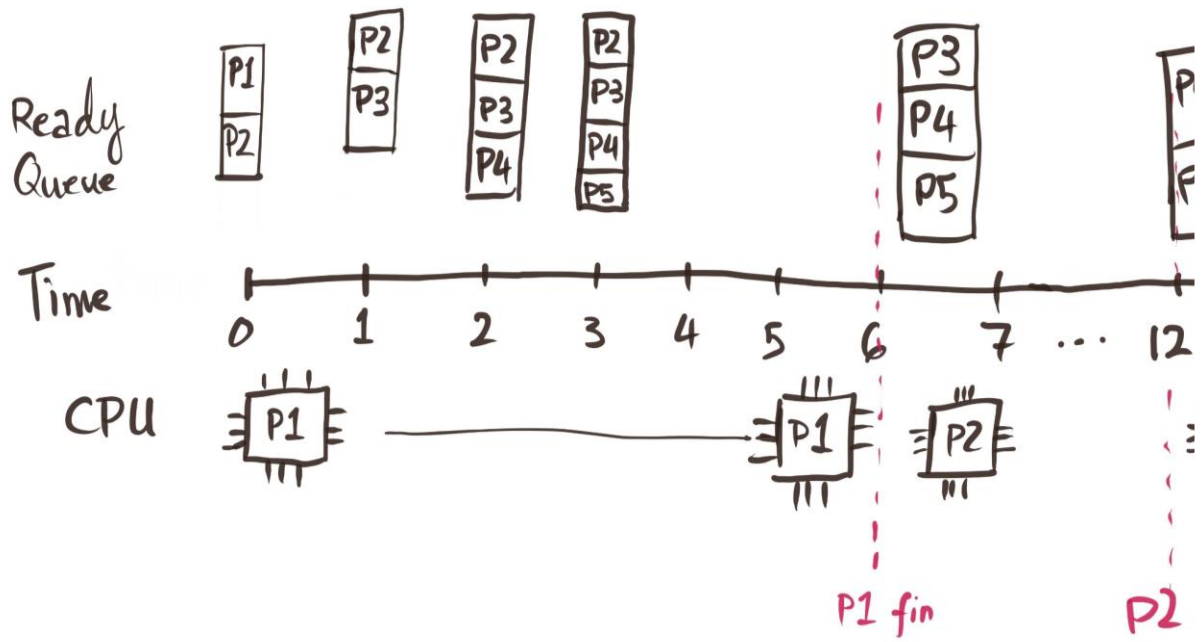
First Come First Serve (FCFS)

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



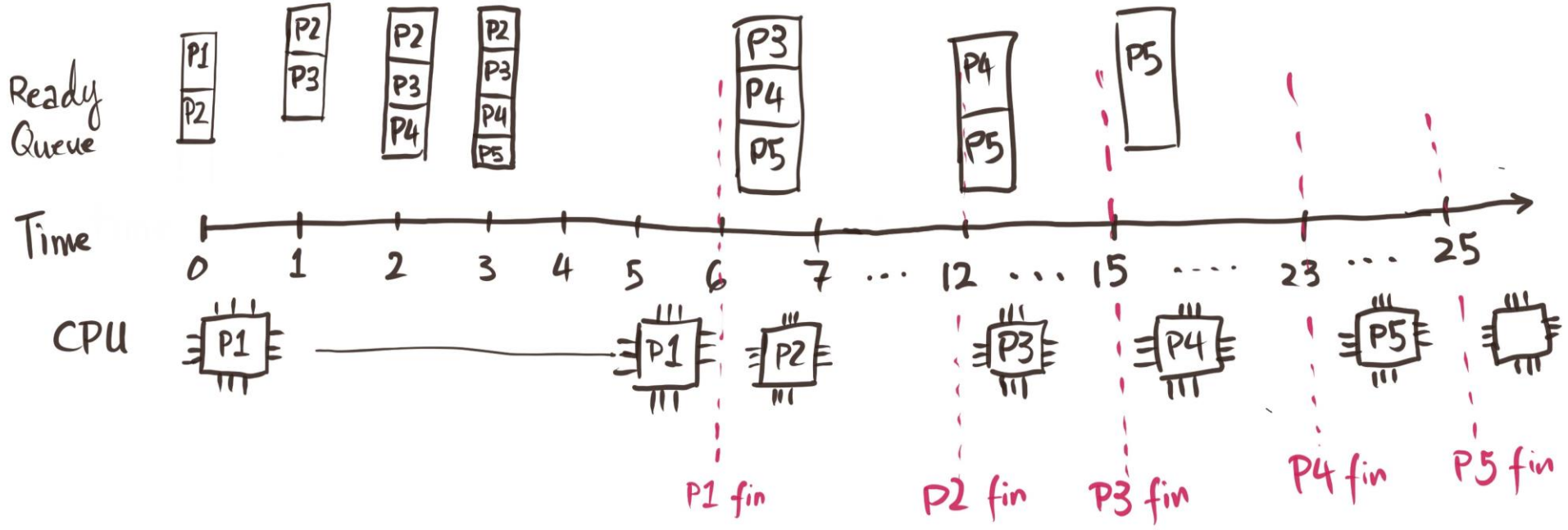
First Come First Serve (FCFS)

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



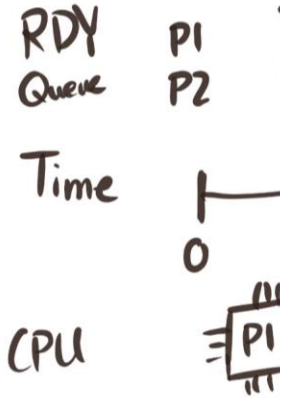
Round-Robin (RR)

- 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

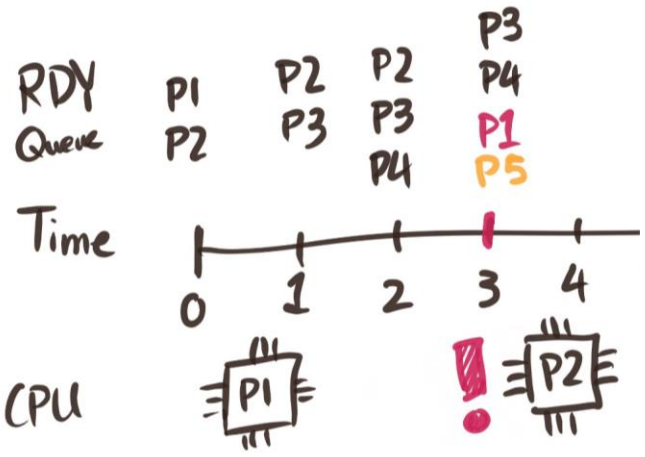
Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



Round-Robin (RR)

Suppose 3 s time slice

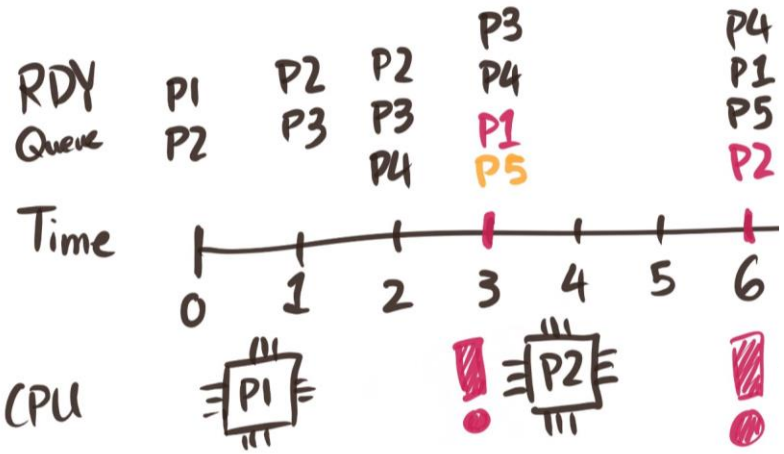
Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



Round-Robin (RR)

Suppose 3 s time slice

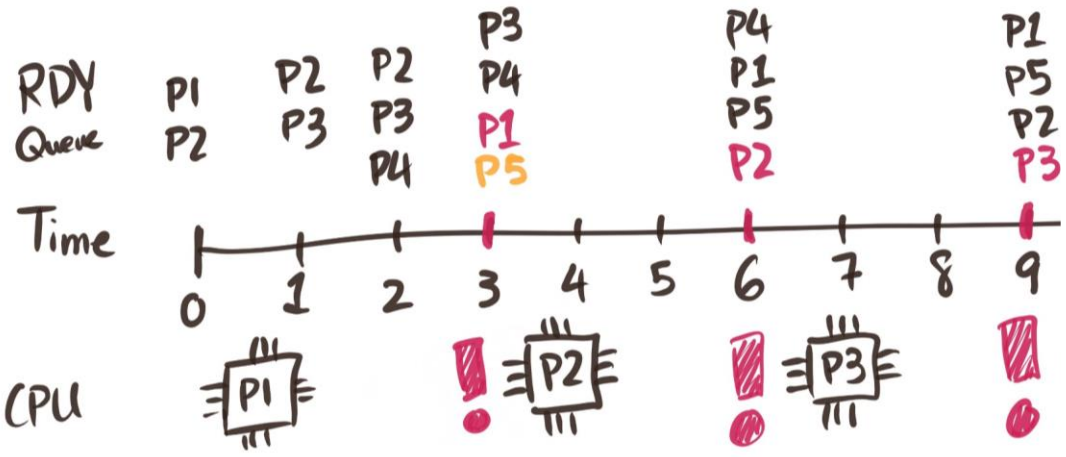
Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



Round-Robin (RR)

Suppose 3 s time slice

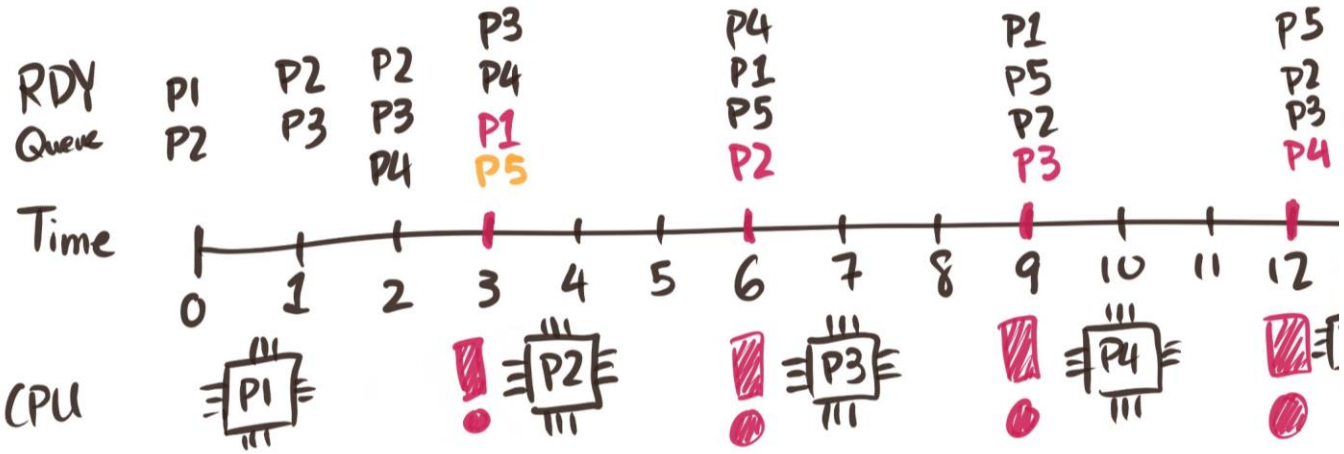
Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



Round-Robin (RR)

Suppose 3 s time slice

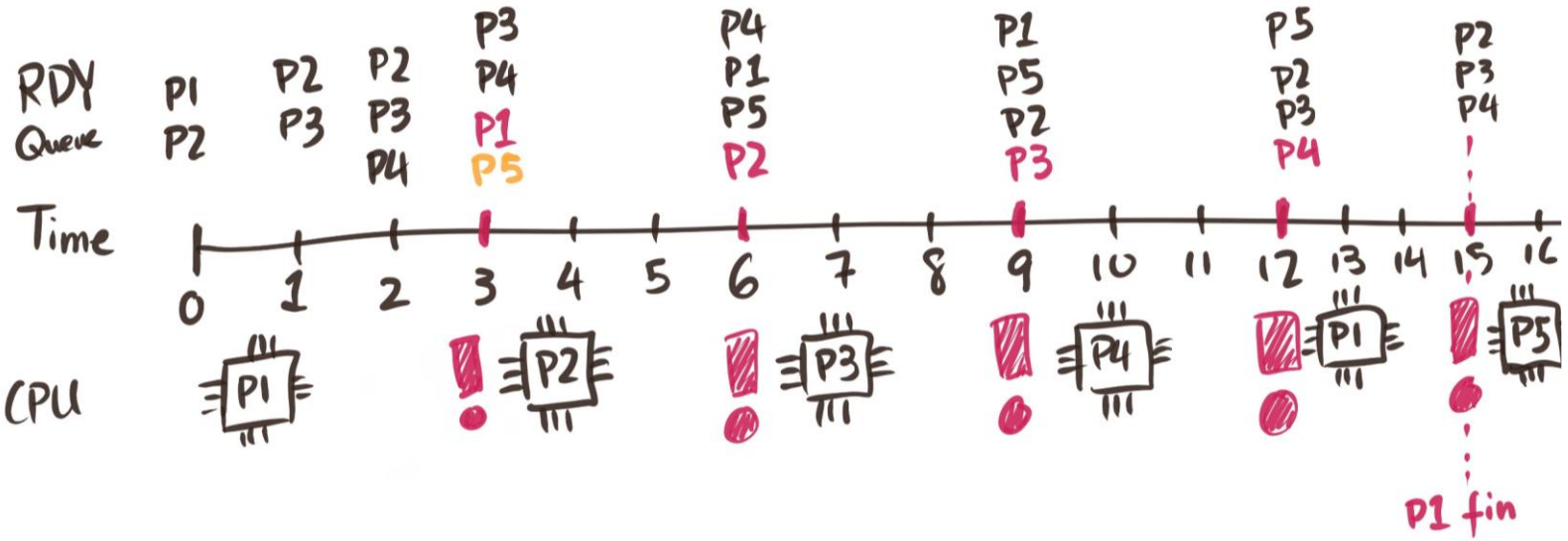
Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



Round-Robin (RR)

Suppose 3 s time slice

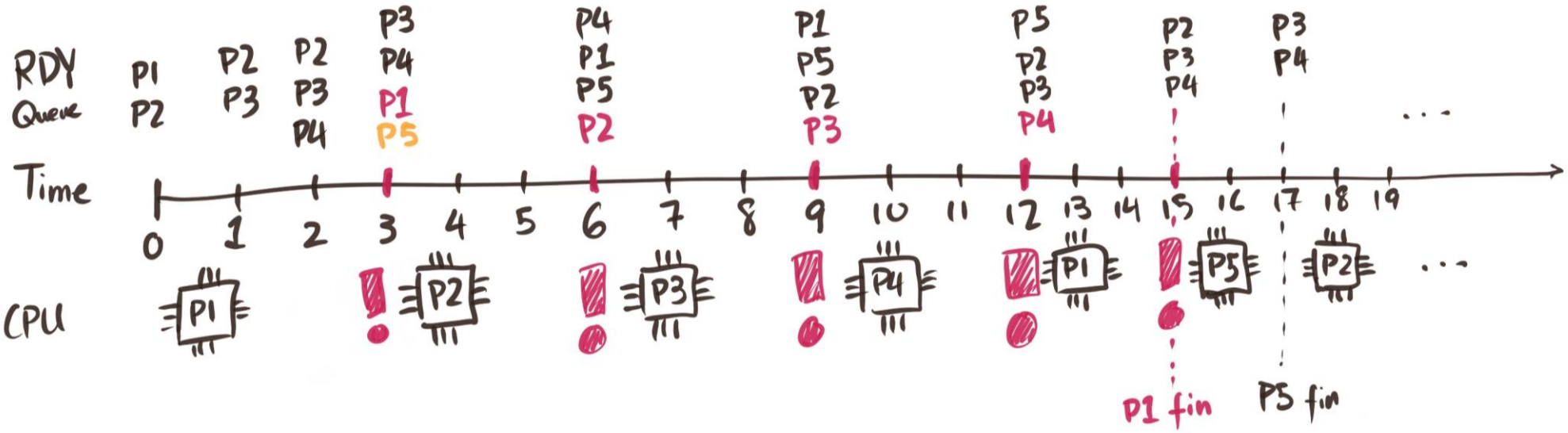
Process	Arrival	Burst
P1	0	6
P2	0	6
P3	1	3
P4	2	8
P5	3	2



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

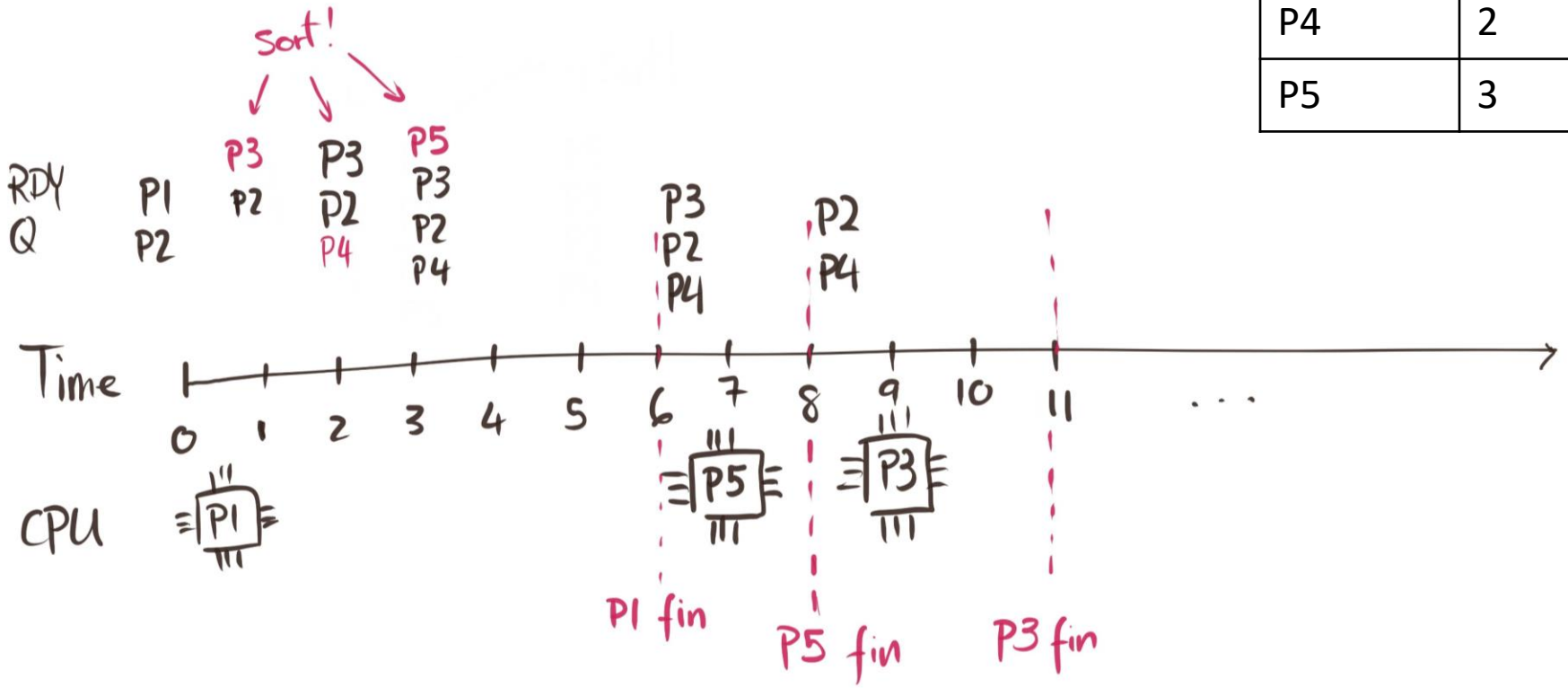


Shortest-Job-First (SJF)

- 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

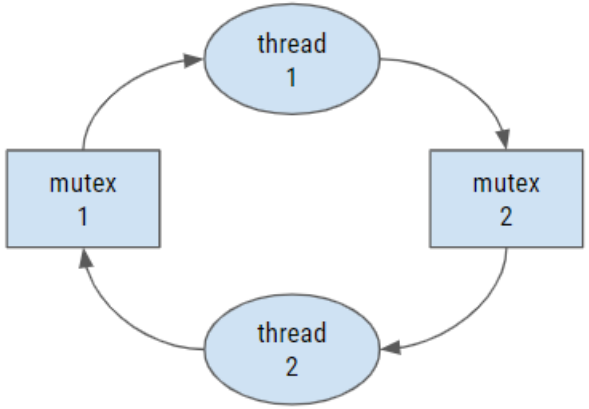
```
curr_time = 0
while(1) {
    if simulation done break
    …
    curr_time ++
}
```

Demo (fcfsSimulationLoop.cpp)

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
- Depth First Search

Deadlock Detection

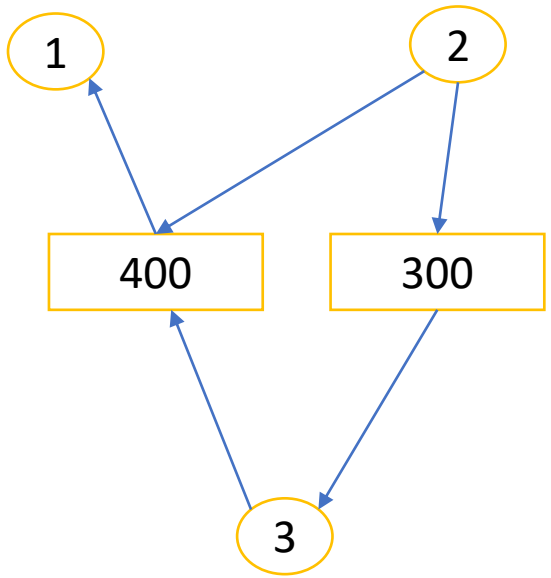
- Topological sort:
  - Need to keep track of “Need” / Request (Out-degree)
  - Need to keep track of “Have” (incoming nodes)

“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

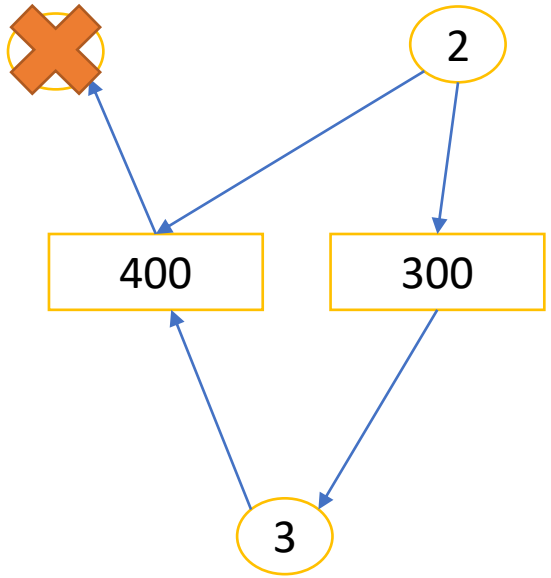
Need to update out degree of all dependents (incoming nodes) every time something gets removed from the adjacency list

Deadlock Detection



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

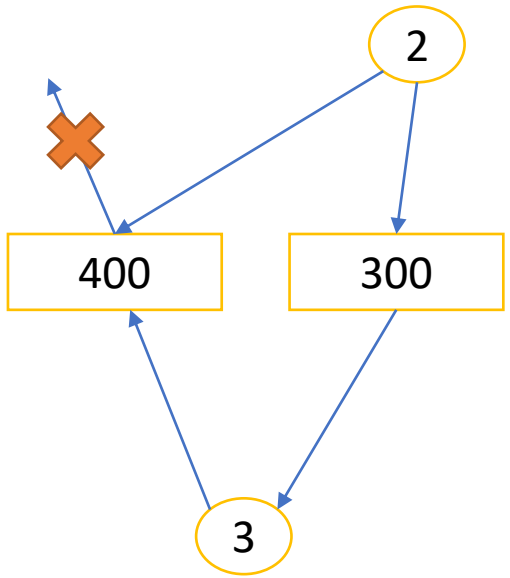
Deadlock Detection



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

Remove!

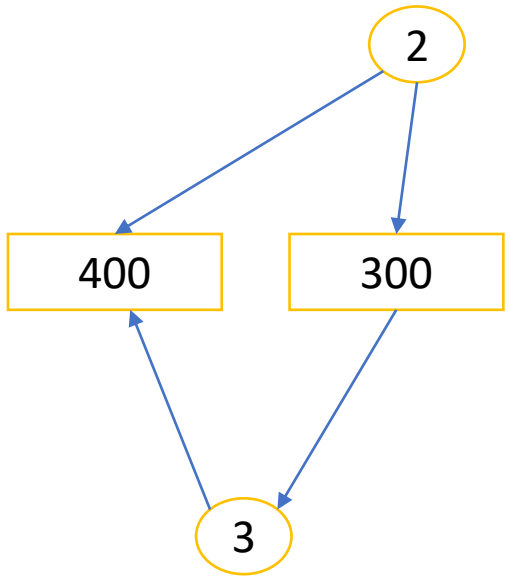
Deadlock Detection



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

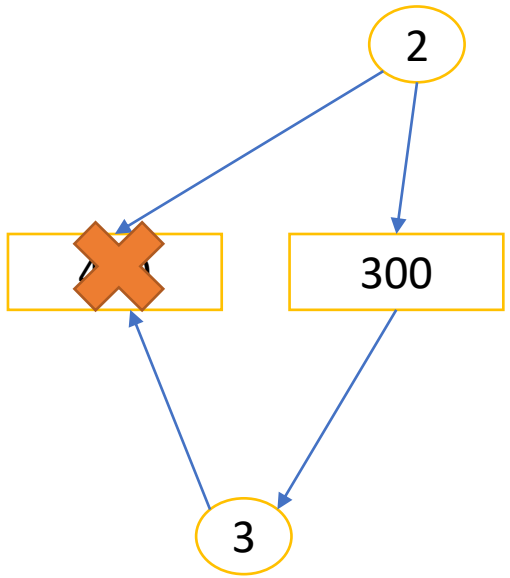
-1

Deadlock Detection



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

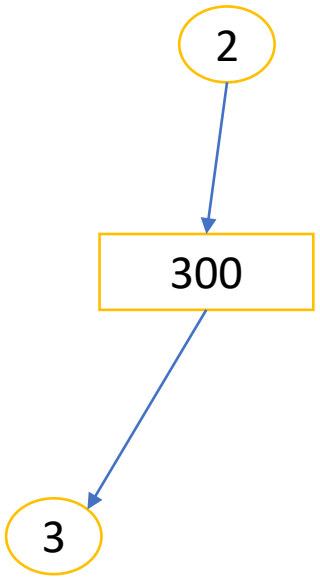
Deadlock Detection



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

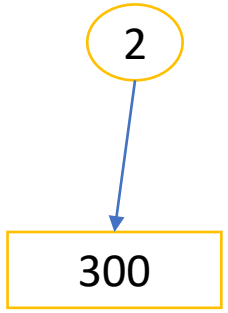
Remove!

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

2

Nodes	Incoming nodes	Outgoing degree
2	[]	0

Deadlock Detection

No Deadlock! ☺

Nodes	Incoming nodes	Outgoing degree

Optimizing topological sort

- Recall (Topological sort):
  1. Remove nodes with an out-degree of 0
  2. Update incoming nodes out-degree count

Optimizing topological sort

- Recall (Topological sort):
  1. Remove nodes with an out-degree of 0
  2. Update incoming nodes out-degree count

Finding nodes with out-degree = 0

Optimizing topological sort

- Recall (Topological sort):
  1. Remove nodes with an out-degree of 0
  2. Update incoming nodes out-degree count

Finding nodes with out-degree = 0

Option 1:
Keep adjacency list sorted by out-degree.

Optimizing topological sort

- Recall (Topological sort):
  1. Remove nodes with an out-degree of 0
  2. Update incoming nodes out-degree count

Finding nodes with out-degree = 0

Option 1:
Keep adjacency list sorted by out-degree.

Problem:
Lots of sorting slows down program.

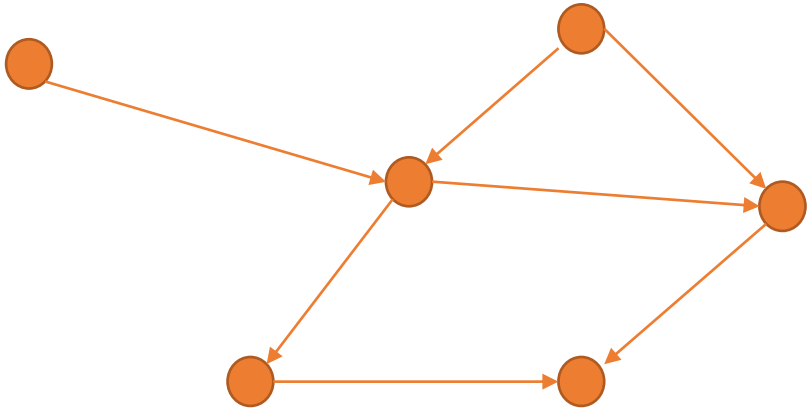
Optimizing topological sort

- Keep another list of nodes with out-degree 0.
  - Call this list2 for this example
- Sort the adjacency list only once.
  - Every time you perform step #2 (updating out-degree), if the out-degree becomes 0, add this node to list2.
- Step #1 (removing nodes) will always be done on list2.

- Recommended data structure: **unordered map**

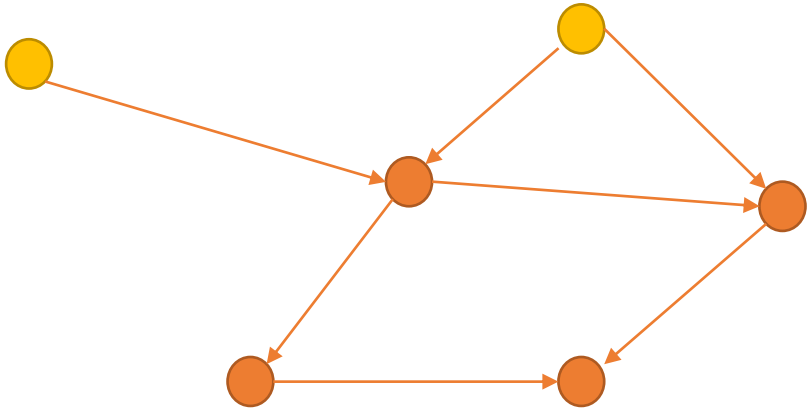
DFS for cycle detection

- Input: Directed graph



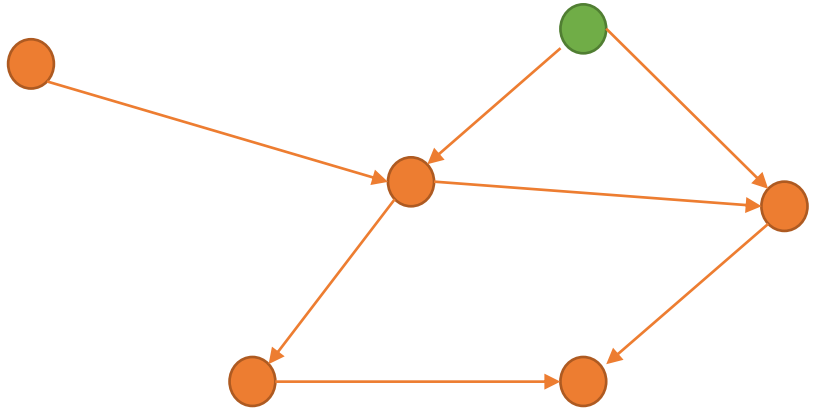
DFS for cycle detection

- Input: Directed graph



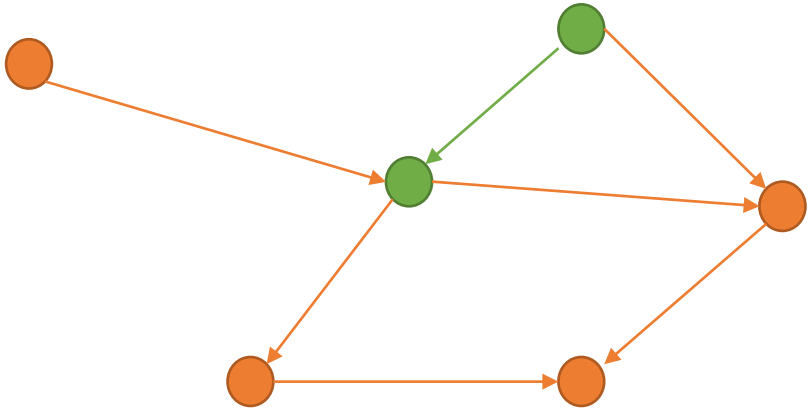
DFS for cycle detection

- Input: Directed graph



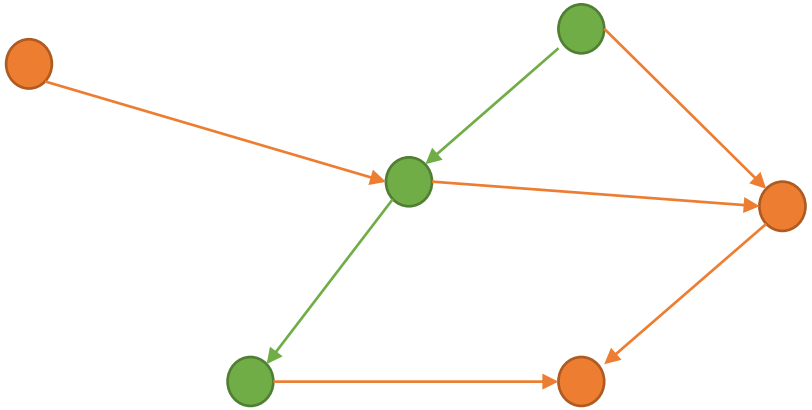
DFS for cycle detection

- Input: Directed graph



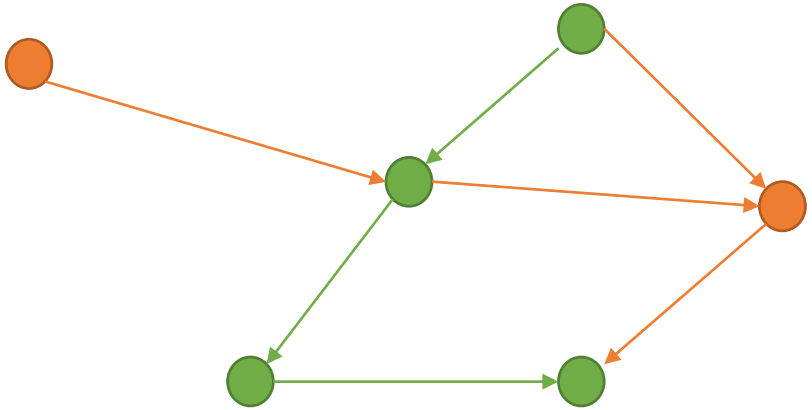
DFS for cycle detection

- Input: Directed graph



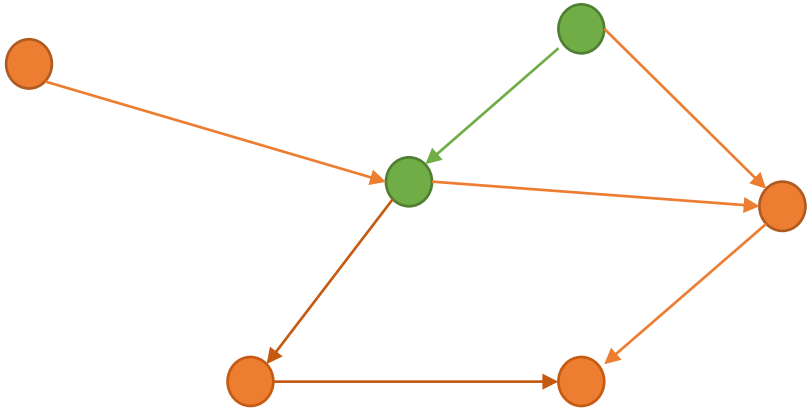
DFS for cycle detection

- Input: Directed graph



DFS for cycle detection

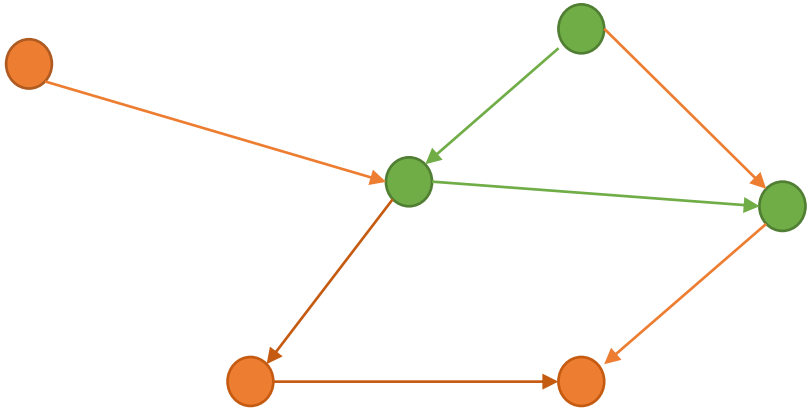
- Input: Directed graph



“unmark” when you “back track”.

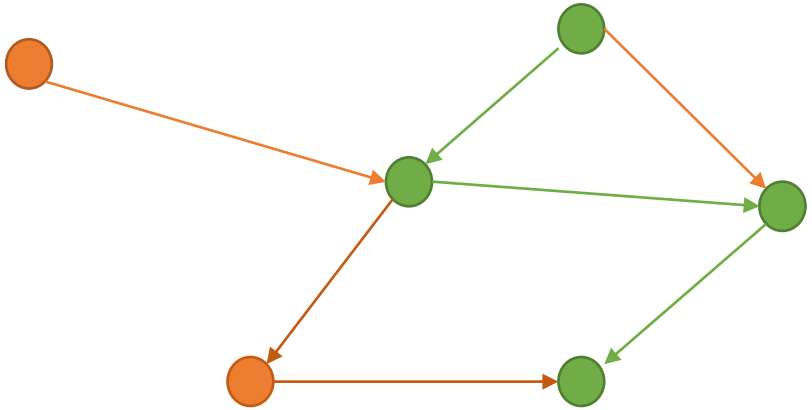
DFS for cycle detection

- Input: Directed graph



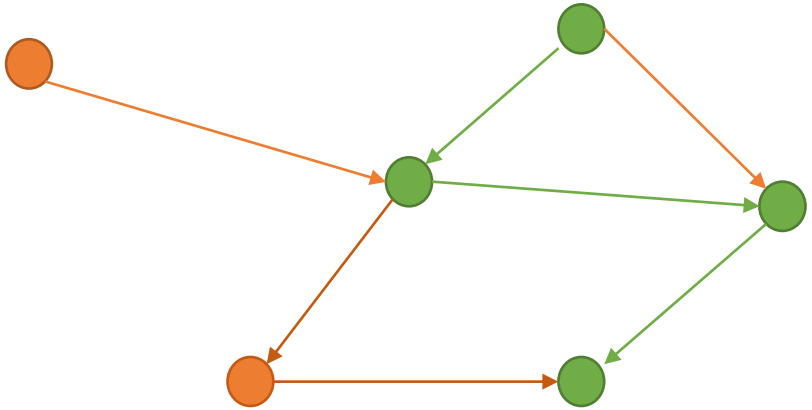
DFS for cycle detection

- Input: Directed graph



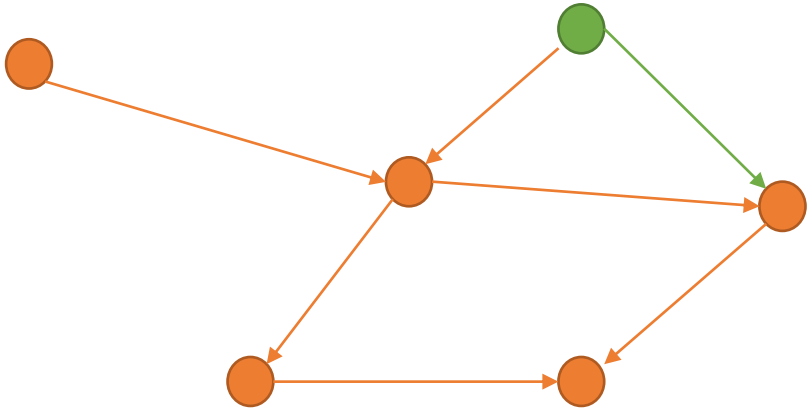
DFS for cycle detection

- Input: Directed graph



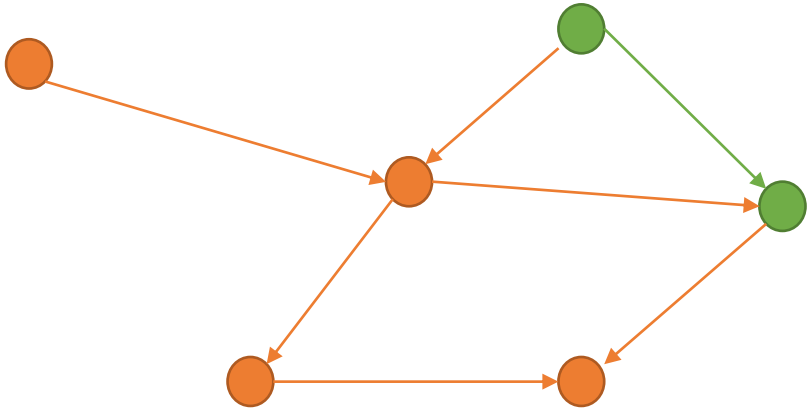
DFS for cycle detection

- Input: Directed graph



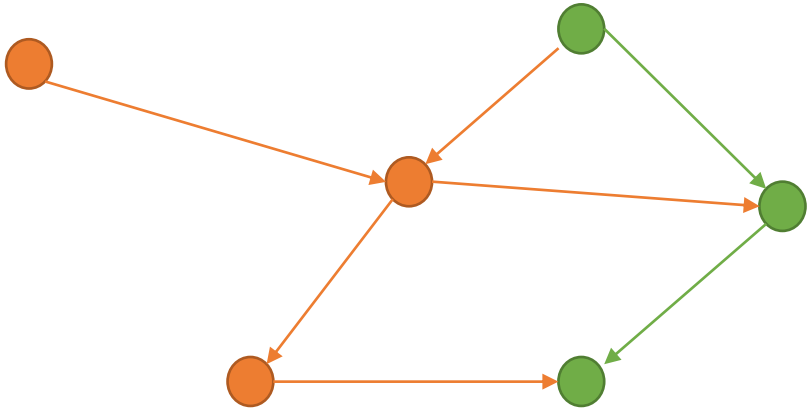
DFS for cycle detection

- Input: Directed graph



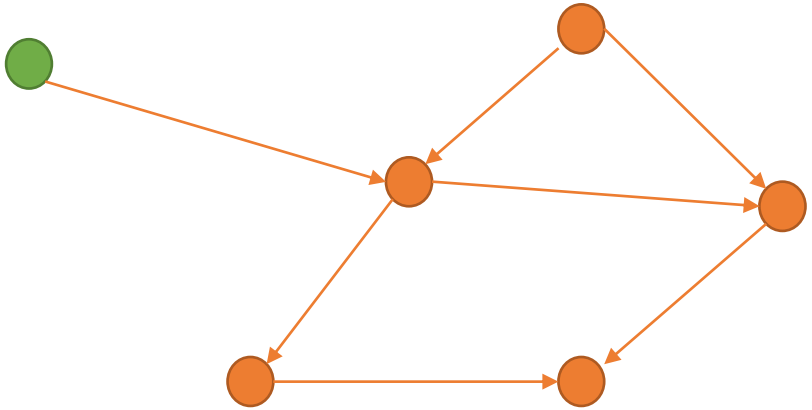
DFS for cycle detection

- Input: Directed graph



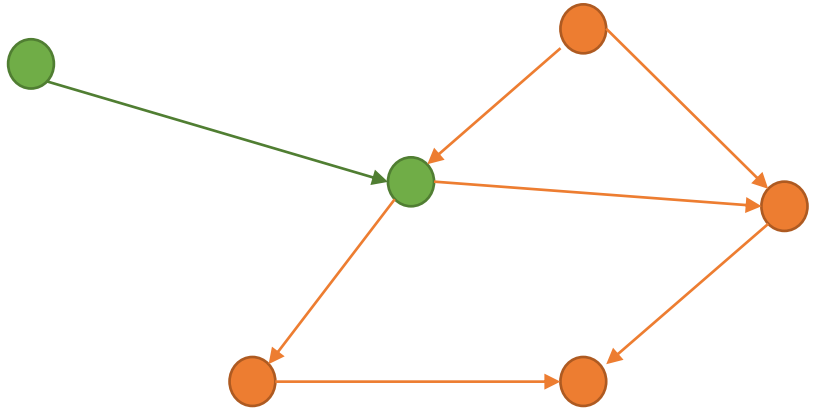
DFS for cycle detection

- Input: Directed graph



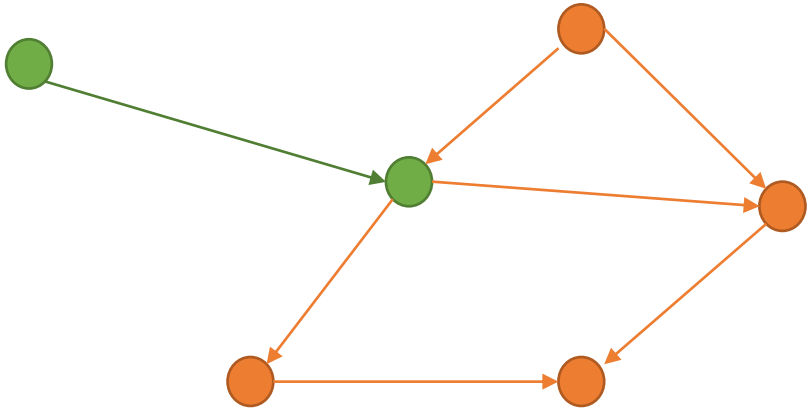
DFS for cycle detection

- Input: Directed graph



DFS for cycle detection

- Input: Directed graph

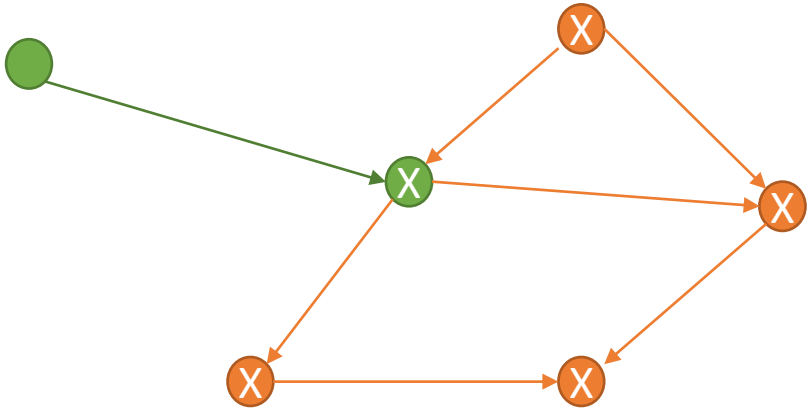


Repeat…

Is there a way to not repeat check this?
This node was already checked.

DFS for cycle detection

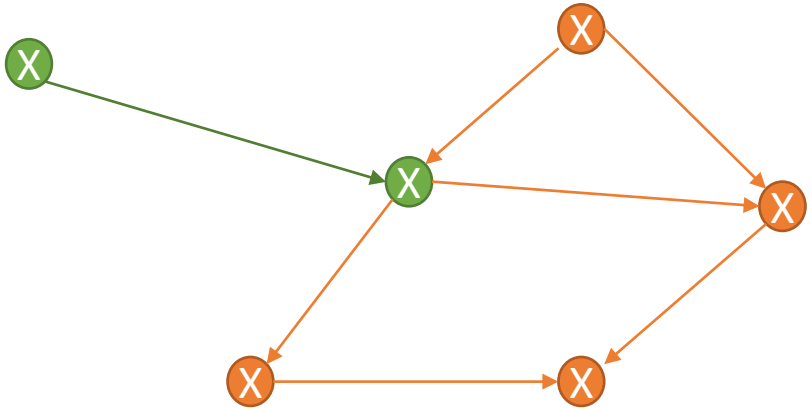
- Input: Directed graph



Use a “No-cycle” flag for each vertex!

DFS for cycle detection

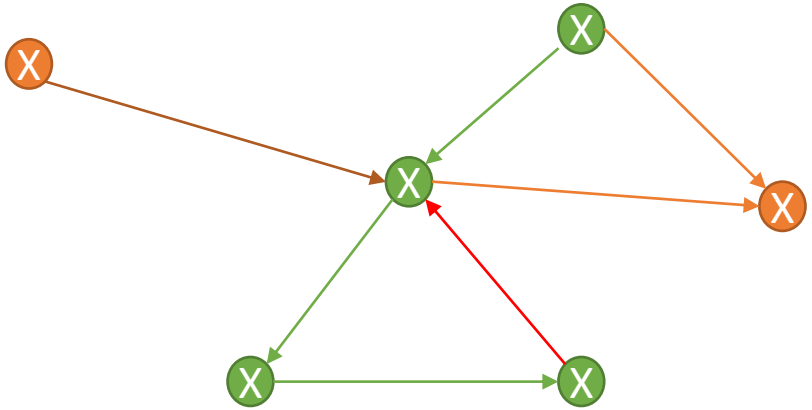
- Input: Directed graph



Use a “No-cycle” flag for each vertex!

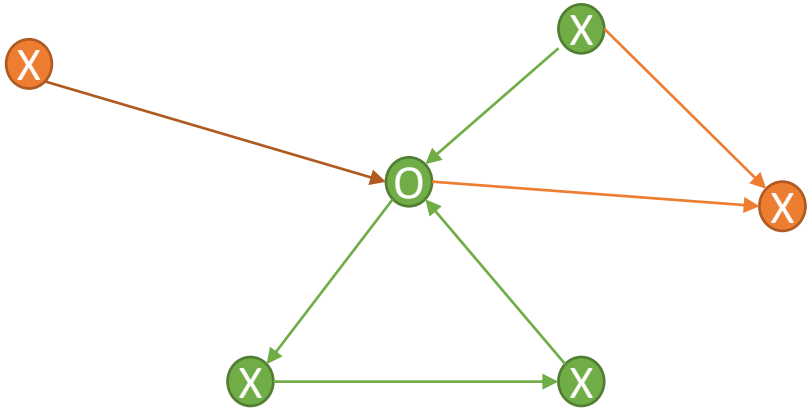
DFS for cycle detection

- Example with a cycle



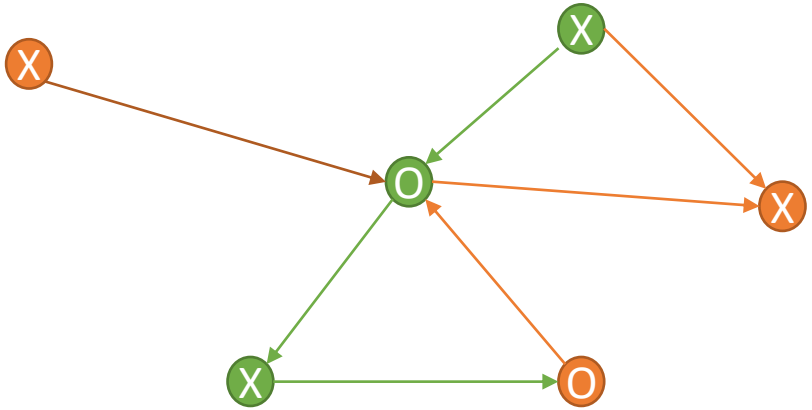
DFS for cycle detection

- Example with a cycle



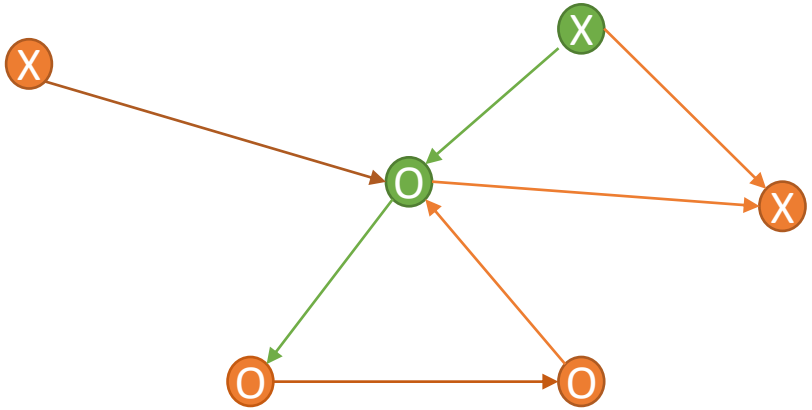
DFS for cycle detection

- Example with a cycle



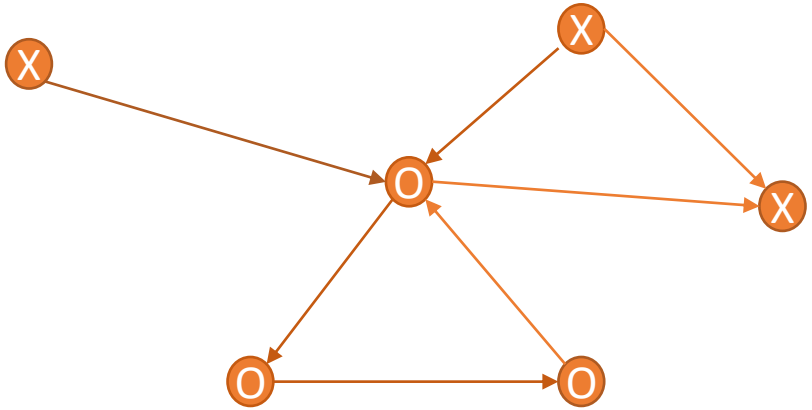
DFS for cycle detection

- Example with a cycle



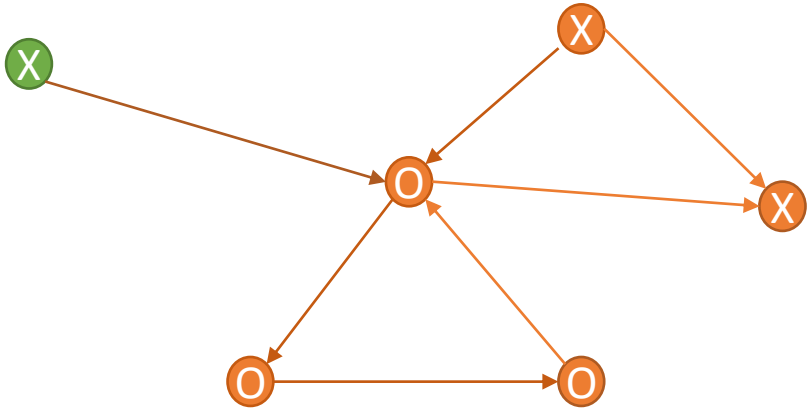
DFS for cycle detection

- Example with a cycle



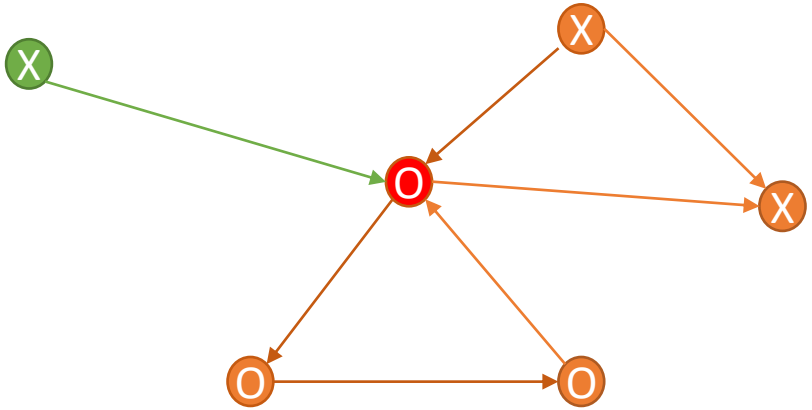
DFS for cycle detection

- Example with a cycle



DFS for cycle detection

- Example with a cycle



Use similar trick in Assignment 5