CPSC 457

Deadlocks

Michelle Nguyen

Contains slides from Dr. Pavol Federl and Xining Chen…
and Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos

Deadlock definition

From Dr. Pavol Federl's CPSC 457 Slides (15ab-deadlock)

- a set of processes are in deadlock if:
    1. each process in the set is waiting for an event, **AND**
    2. an event can be caused only by another process in the set.

- in other words, every process is blocked, and can only be unblocked by another process
- event could be anything, eg.
    - resource becoming available
    - mutex/semaphore/spinlock being unlocked
    - message arriving

System model

- we assume processes are well behaved (programs are well written)
- each process utilizes a resource in the same manner:
  1. process **requests** the resource – OS may block process
  2. process **uses** the resource – for a finite amount of time
  3. process **releases** the resource – may result in unblocking of related process(es)

Deadlock - necessary conditions

From Dr. Pavol Federl's CPSC 457 Slides (15ab-deadlock)

- **mutual exclusion condition**
  - the involved resources must be unshareable (max. one process per resource)
- **hold and wait condition**
  - a process holding at least one resource is waiting to acquire additional resources
- **no preemption condition**
  - a resource can be released only by the process holding it (voluntary)
- **circular wait condition**
  - there is an ordering of processes $\{P_1, P_2, \dots, P_n\}$, such that
    - P_1 waits for P_2
    - P_2 waits for P_3, …
    - P_n waits for P_1
  - ie. there is a cycle
- aka. Coffman conditions

Deadlock can arise if and only if all four conditions hold simultaneously!

Deadlock with mutex locks

From Dr. Pavol Federl's CPSC 457 Slides (15ab-deadlock)

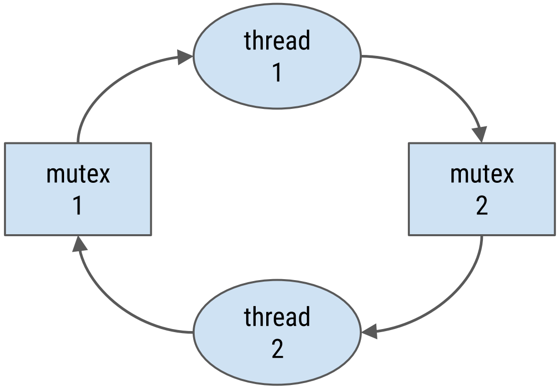
- deadlocks can occur in many different ways, eg. due to locking
- simplest example – deadlock with 2 mutexes:

```
Thread 1:

lock( mutex1 );
lock( mutex2 );
// critical section
unlock( mutex2 );
unlock( mutex1 );
```

```
Thread 2:

lock( mutex2 );
lock( mutex1 );
// critical section
unlock( mutex1 );
unlock( mutex2 );
```



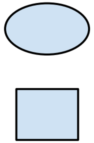
- notice that all 4 necessary conditions present:

  mutual exclusion, hold and wait, no preemption, circular wait

Resource-Allocation Graph with 1 instance per resource type

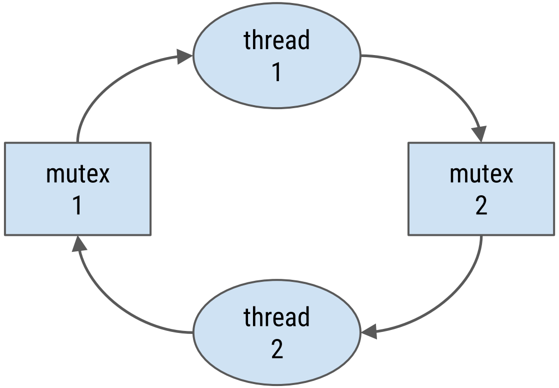
From Dr. Pavol Federl's CPSC 457 Slides (15ab-deadlock)

- graph with a set of vertices V and a set of directed 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 $P_i \rightarrow R_j$

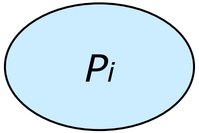
- assignment edge — directed edge $R_j \rightarrow P_i$



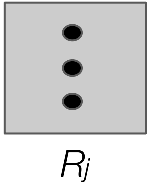
Resource-Allocation Graph with multiple instances per resource

From Dr. Pavol Federl’s CPSC 457 Slides (15ab-deadlock)

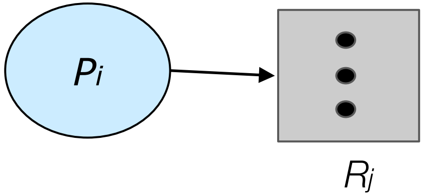
- **process** *P_i* :



- **multiple instances** of **resource type** are represented as dots inside resources, eg. resource R_j with 3 instances:

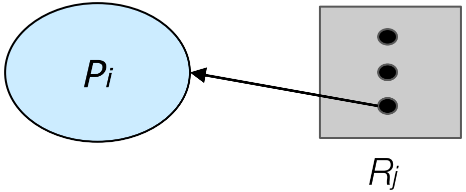


- P_i **requests** an instance of R_j :



request edge points to resource type, not resource instance

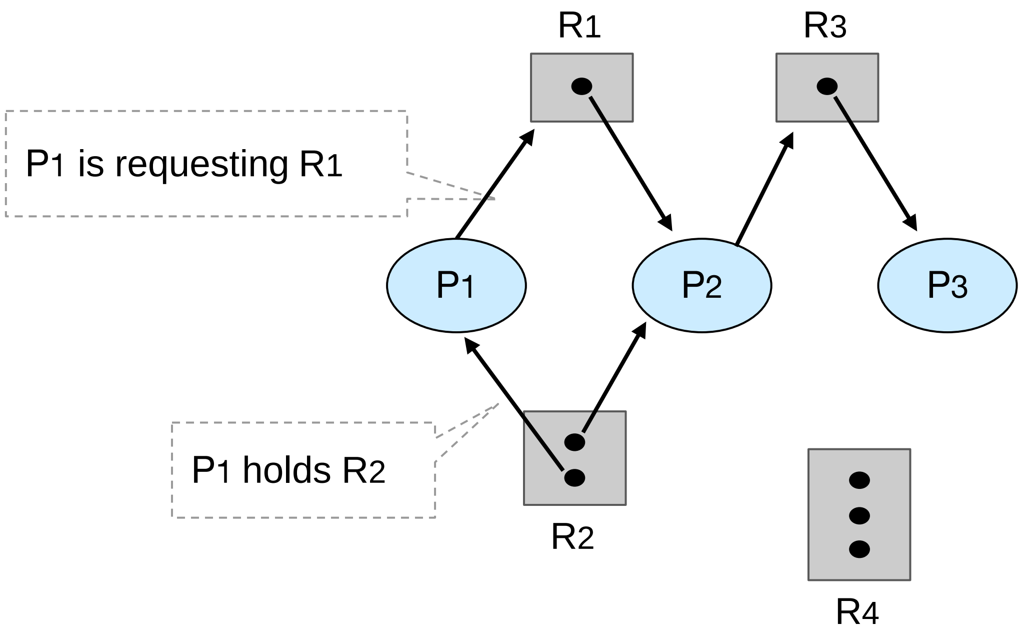
- P_i is **holding** an instance of R_j :



assignment edge originates from instance, not type

Resource Allocation Graph Example

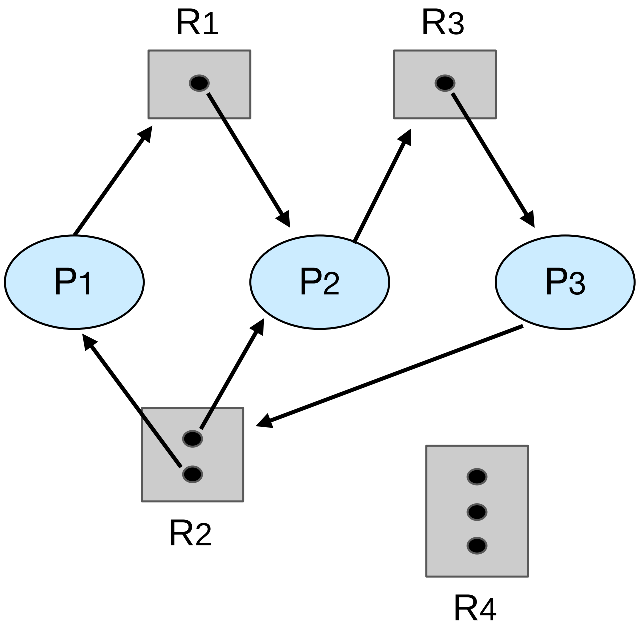
From Dr. Pavol Federl's CPSC 457 Slides (15ab-deadlock)



no cycle in the graph ⇒ no deadlock

Resource Allocation Graph With A Deadlock

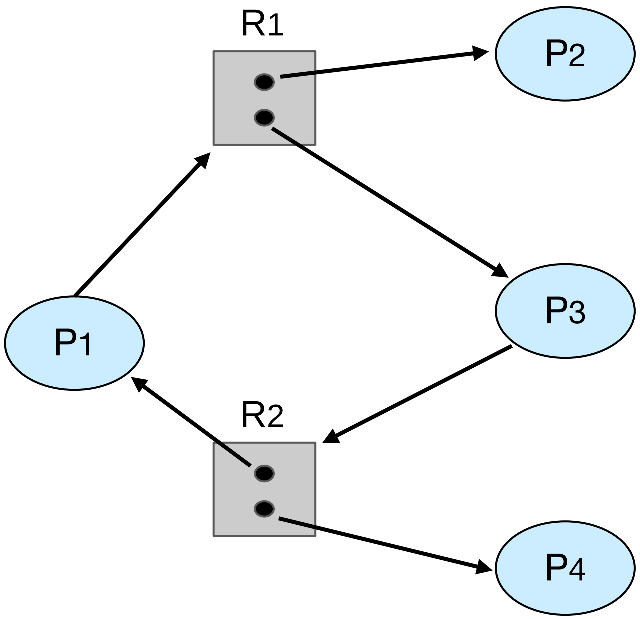
From Dr. Pavol Federl's CPSC 457 Slides (15ab-deadlock)



deadlock ⇒ cycle

Graph With A Cycle But No Deadlock

From Dr. Pavol Federl's CPSC 457 Slides (15ab-deadlock)

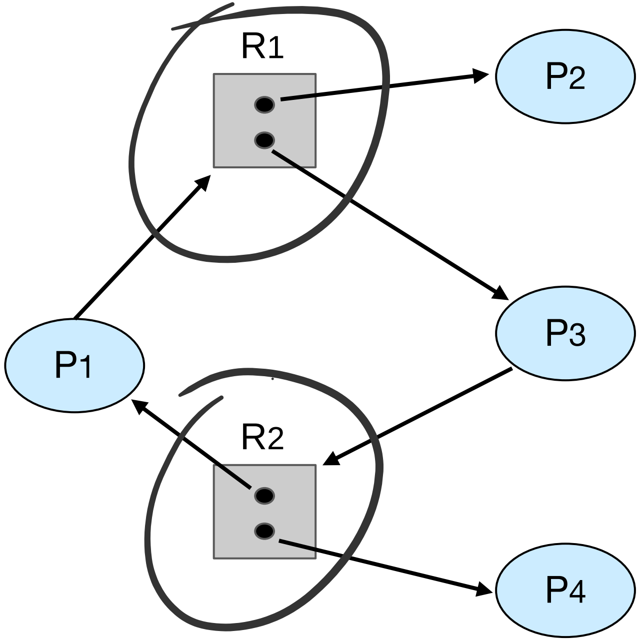


cycle $\nRightarrow$ deadlock

Graph With A Cycle But No Deadlock

Modified from Dr. Pavol Federl's CPSC 457 Slides (15ab-deadlock)

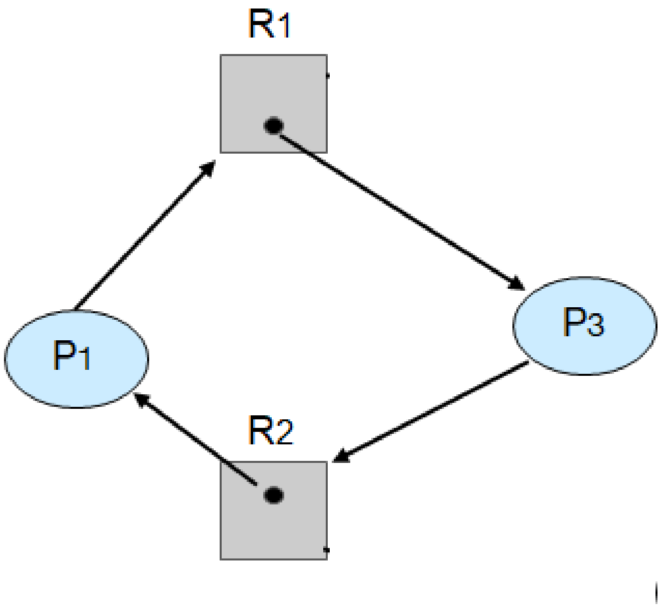
multiple instances of resource type



cycle $\not\Rightarrow$ deadlock

Graph With A Cycle But No Deadlock

Modified from Dr. Pavol Federl's CPSC 457 Slides (15ab-deadlock)



For single instance per resource type:
cycle ⇒ deadlock

Cycle Detection

- Cycle Detection:
  - Let's look at how we can use **Topological Sort** to detect **cycles** in a Resource-Allocation graph
  - For the **following examples, we will assume each resource type only has a single instance**
    - Again, when each resource type only has a single instance,

      then if there is a cycle detected then we have a deadlock!
  - Wiki for topological sort: https://en.wikipedia.org/wiki/Topological_sorting
    - We're going to do something similar to the pseudo-code listed based Kahn's algorithm
    - Slight variation in that we are keeping track of "outgoing degree" and "incoming nodes" instead but it's more-or-less the same thing/concept

From TA Xining Chen’s Slides
(CPSC 457 -Scheduling - Deadlocks)

Deadlock Detection

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

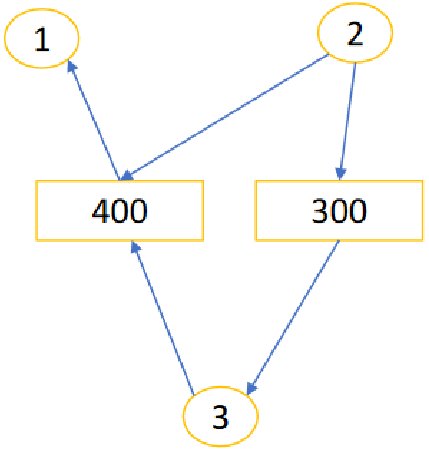
“If I don’t need anything, I can execute and release my acquired resources”

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

From TA Xining Chen's Slides
(CPSC 457 -Scheduling - Deadlocks)

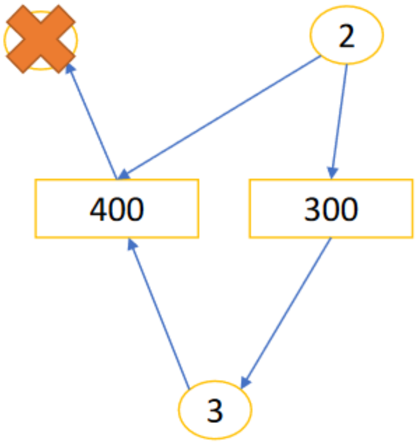
Deadlock Detection



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

From TA Xining Chen's Slides
(CPSC 457 -Scheduling - Deadlocks)

Deadlock Detection

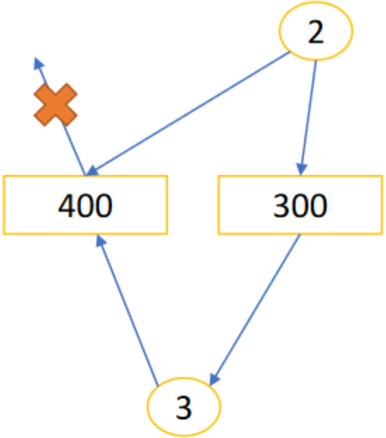


Nodes	Incoming nodes	Outgoing degree
1	[400]	0
2	[]	2
400	[2,3]	1
300	[2]	1
3	[300]	1

Remove!

From TA Xining Chen's Slides
(CPSC 457 -Scheduling - Deadlocks)

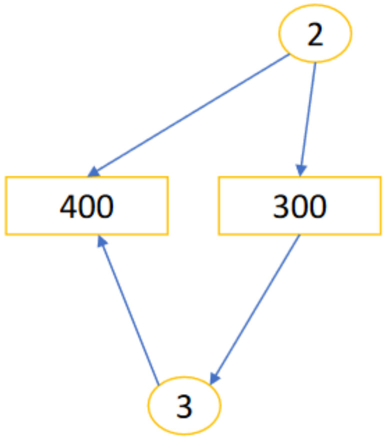
Deadlock Detection



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

From TA Xining Chen’s Slides
(CPSC 457 -Scheduling - Deadlocks)

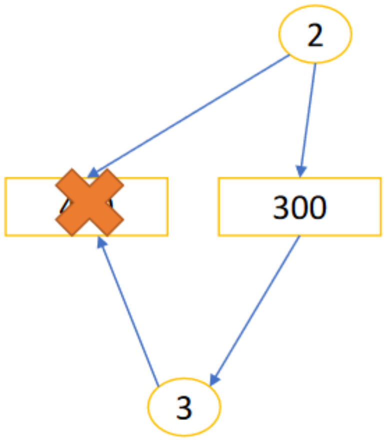
Deadlock Detection



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

From TA Xining Chen's Slides
(CPSC 457 -Scheduling - Deadlocks)

Deadlock Detection

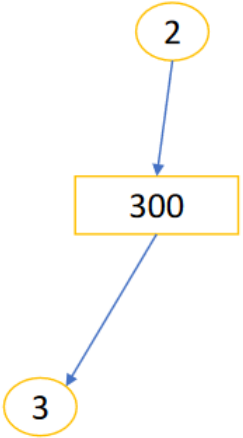


Nodes	Incoming nodes	Outgoing degree
2	[]	2
400	[2,3]	0
300	[2]	1
3	[300]	1

Remove!

From TA Xining Chen’s Slides
(CPSC 457 -Scheduling - Deadlocks)

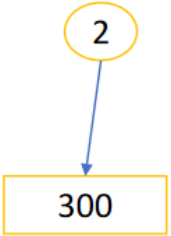
Deadlock Detection



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

From TA Xining Chen's Slides
(CPSC 457 -Scheduling - Deadlocks)

Deadlock Detection



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

Deadlock Detection

From TA Xining Chen's Slides (CPSC 457 -Scheduling - Deadlocks)

Nodes	Incoming nodes	Outgoing degree
2	[]	0

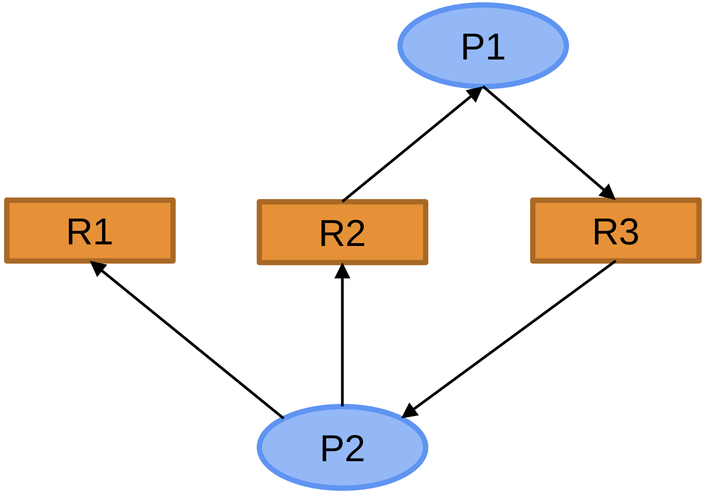
From TA Xining Chen's Slides
(CPSC 457 -Scheduling - Deadlocks)

Deadlock Detection

No Deadlock! ☺

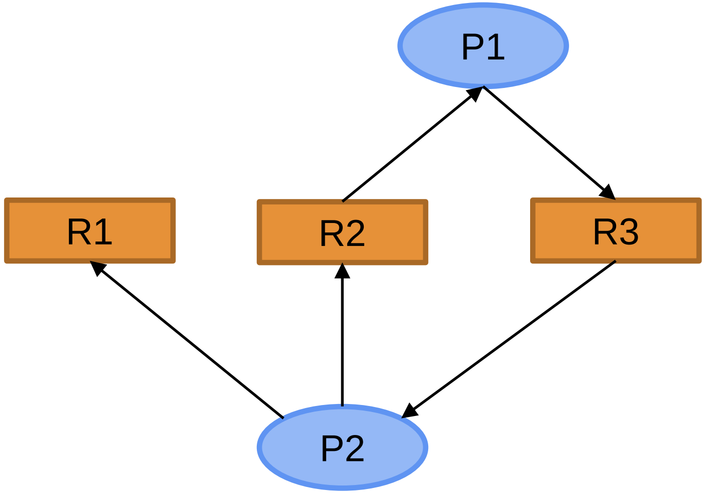
Nodes	Incoming nodes	Outgoing degree

Topological sort (Example with Deadlock)



Nodes	Incoming Nodes	Outgoing degree
P1	[R2]	1
P2	[R3]	2
R1	[P2]	0
R2	[P2]	1
R3	[P1]	1

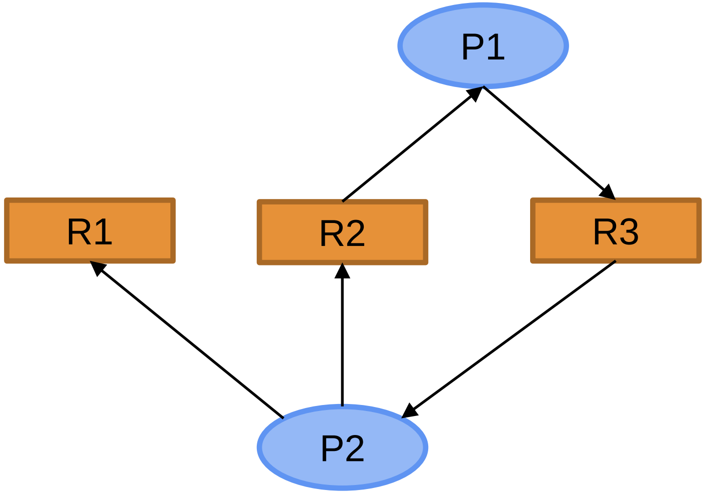
Topological sort (Example with Deadlock)



Nodes	Incoming Nodes	Outgoing degree
P1	[R2]	1
P2	[R3]	2
R1	[P2]	0
R2	[P2]	1
R3	[P1]	1

Remove b/c outgoing degree is 0

Topological sort (Example with Deadlock)

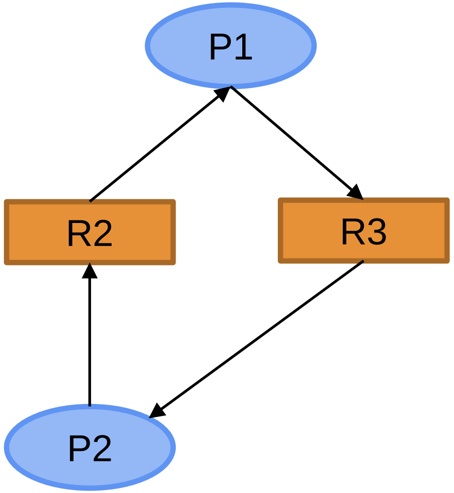


Nodes	Incoming Nodes	Outgoing degree
P1	[R2]	1
P2	[R3]	2 -1
R1	[P2]	0
R2	[P2]	1
R3	[P1]	1

Remember to decrement outgoing degree for any nodes listed as incoming nodes

Remove b/c outgoing degree is 0

Topological sort (Example with Deadlock)



Nodes	Incoming Nodes	Outgoing degree
P1	[R2]	1
P2	[R3]	1
R2	[P2]	1
R3	[P1]	1

Uh oh! No more nodes with outgoing degree == 0 but there are still nodes remaining…

Deadlock detected!

From TA Xining Chen’s Slides
(CPSC 457 -Scheduling - Deadlocks)

Optimizing topological sort

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

From TA Xining Chen's Slides (CPSC 457 -Scheduling - Deadlocks)

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

From TA Xining Chen's Slides (CPSC 457 -Scheduling - Deadlocks)

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.

From TA Xining Chen’s Slides (CPSC 457 -Scheduling - Deadlocks)

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

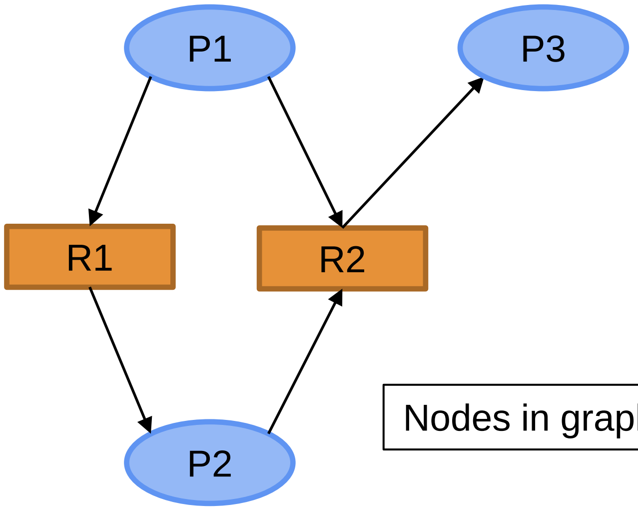
Modified from TA Xining Chen's Slides (CPSC 457 -Scheduling - Deadlocks)

- Keep another list of nodes with out-degree 0.
- Every time you perform step #2 (updating out-degree), if the out-degree becomes 0, add this node to your list of nodes with out-degree 0

=> Let's do the first example again

Topological sort (Re-doing the first example)

1. To start: find the nodes with outgoing degree equal to 0, add them to "nodes with outgoing degree equal to 0"



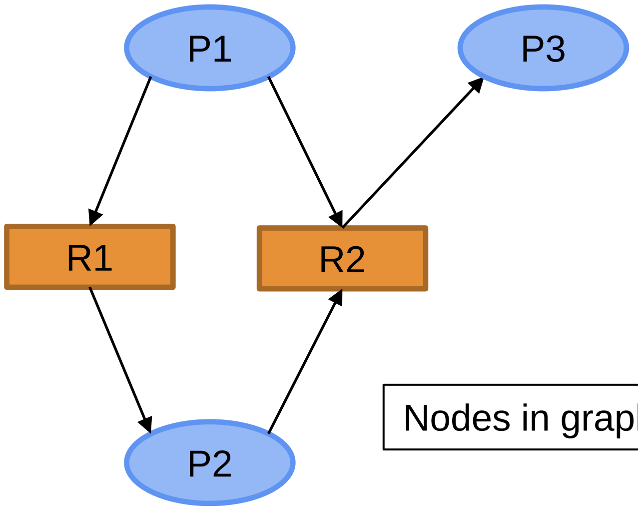
Nodes in graph: 5

Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	2
P2	[R1]	1
P3	[R2]	0
R1	[P1]	1
R2	[P1, P2]	1

Nodes with outgoing degree equal to 0:

Topological sort (Re-doing the first example)

1. To start, find the nodes with outgoing degree equal to 0, add them to “nodes with outgoing degree equal to 0”



Nodes in graph: 5

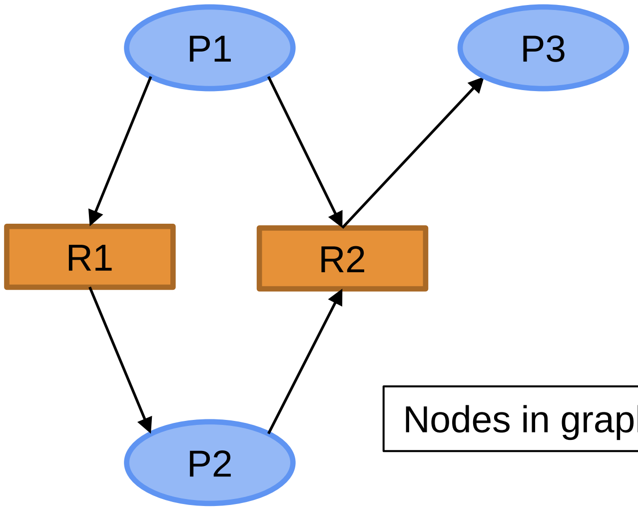
Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	2 ✘
P2	[R1]	1 ✘
P3	[R2]	0 ✓
R1	[P1]	1 ✘
R2	[P1, P2]	1 ✘

Nodes with outgoing degree equal to 0:

P3

Topological sort (Re-doing the first example)

2. **While the list of "nodes with outgoing degree equal to 0" is non-empty**, keep removing nodes…



Nodes in graph: 5

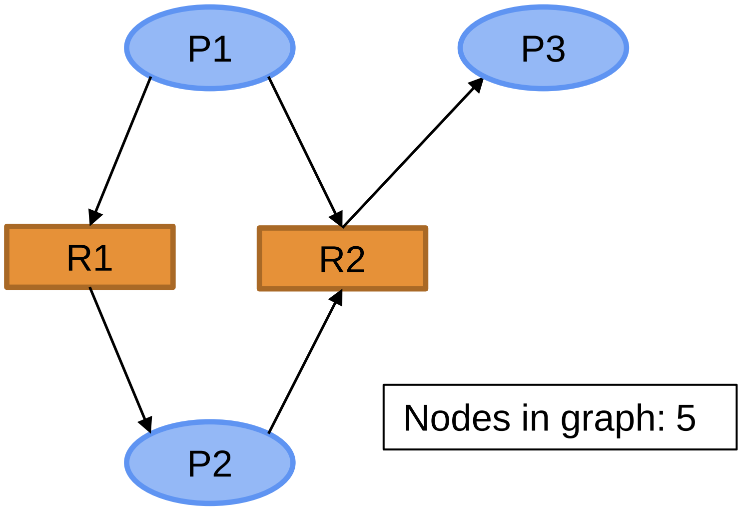
Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	2
P2	[R1]	1
P3	[R2]	0
R1	[P1]	1
R2	[P1, P2]	1

Nodes with outgoing degree equal to 0:

P3

Topological sort (Re-doing the first example)

2. While the list of “nodes with outgoing degree equal to 0” is non-empty, keep **removing a node**…



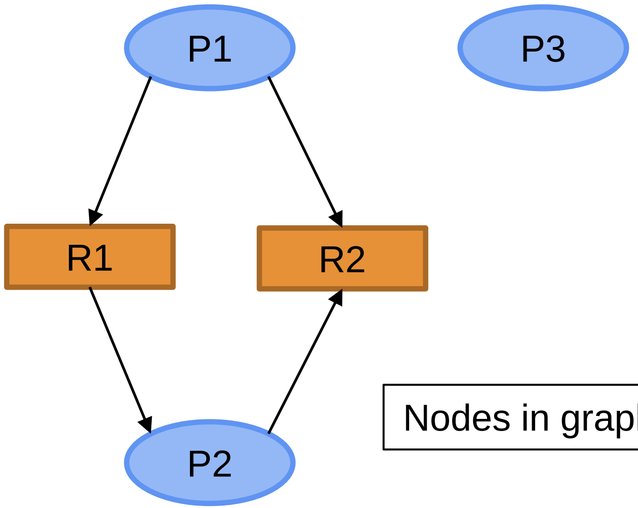
Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	2
P2	[R1]	1
P3	[R2]	0
R1	[P1]	1
R2	[P1, P2]	1 -1

Nodes with outgoing degree equal to 0:

<u>Current node being removed:</u> P3

Topological sort (Re-doing the first example)

2. While the list of “nodes with outgoing degree equal to 0” is non-empty, keep **removing a node**…



Nodes in graph: 5

Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	2
P2	[R1]	1
P3	[R2]	0
R1	[P1]	1
R2	[P1, P2]	0

After decrementing, notice that R2 has outgoing degree == 0. So add R2 to list of nodes with outgoing degree equal to 0

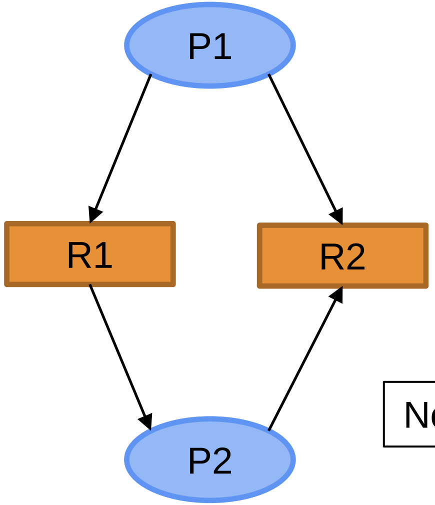
Nodes with outgoing degree equal to 0:

R2

Current node being removed: P3

Topological sort (Re-doing the first example)

2. **While the list of "nodes with outgoing degree equal to 0" is non-empty**, keep removing a node…



Nodes in graph: 4

Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	2
P2	[R1]	1
R1	[P1]	1
R2	[P1, P2]	0

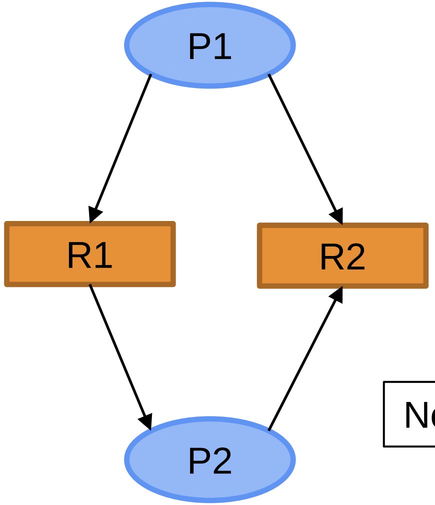
Nodes with outgoing degree equal to 0:

R2

Current node being removed:

Topological sort (Re-doing the first example)

2. While the list of "nodes with outgoing degree equal to 0" is non-empty, keep **removing a node**…



Nodes in graph: 4

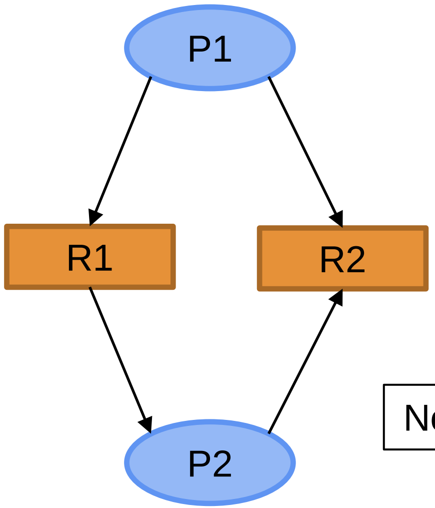
Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	2
P2	[R1]	1
R1	[P1]	1
R2	[P1, P2]	0

Nodes with outgoing degree equal to 0:

<u>Current node being removed:</u> R2

Topological sort (Re-doing the first example)

2. While the list of "nodes with outgoing degree equal to 0" is non-empty, keep **removing a node**…



Nodes in graph: 4

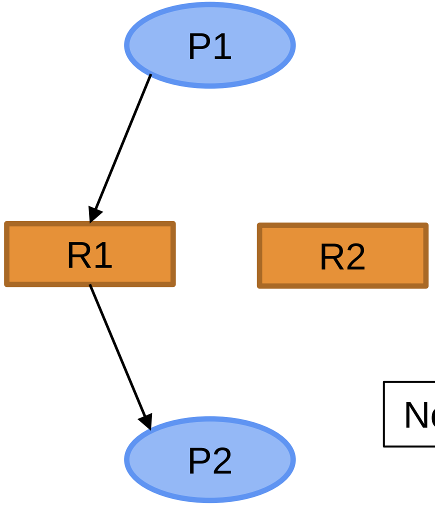
Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	2 -1
P2	[R1]	1 -1
R1	[P1]	1
R2	[P1, P2]	0

Nodes with outgoing degree equal to 0:

<u>**Current node being removed:**</u> **R2**

Topological sort (Re-doing the first example)

2. While the list of “nodes with outgoing degree equal to 0” is non-empty, keep **removing a node**…



Nodes in graph: 4

Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	1
P2	[R1]	0
R1	[P1]	1
R2	[P1, P2]	0

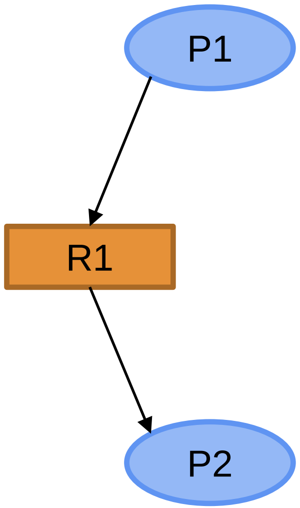
Nodes with outgoing degree equal to 0:

P2

<u>Current node being removed:</u> R2

Topological sort (Re-doing the first example)

2. **While the list of "nodes with outgoing degree equal to 0" is non-empty**, keep removing a node…



Nodes in graph: 3

Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	1
P2	[R1]	0
R1	[P1]	1

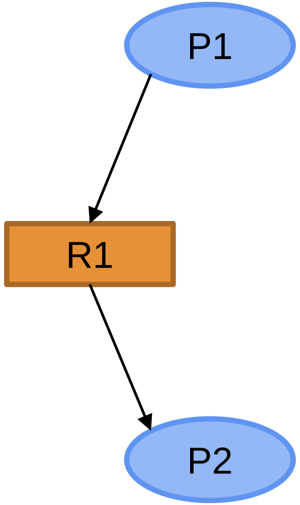
Nodes with outgoing degree equal to 0:

P2

Current node being removed:

Topological sort (Re-doing the first example)

2. While the list of “nodes with outgoing degree equal to 0” is non-empty, keep **removing a node**…



Nodes in graph: 3

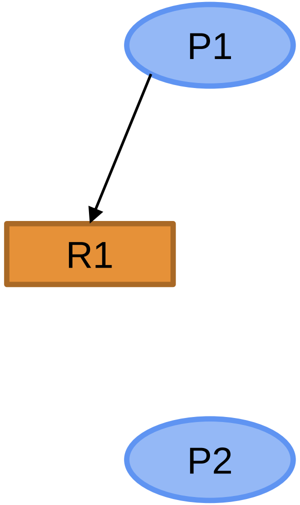
Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	1
P2	[R1]	0
R1	[P1]	1 -1

Nodes with outgoing degree equal to 0:

<u>Current node being removed:</u> P2

Topological sort (Re-doing the first example)

2. While the list of “nodes with outgoing degree equal to 0” is non-empty, keep **removing a node**…



Nodes in graph: 3

Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	1
P2	[R1]	0
R1	[P1]	0

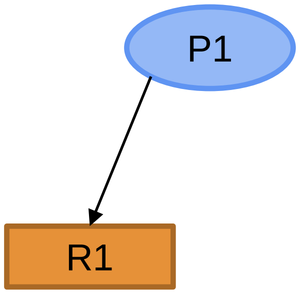
Nodes with outgoing degree equal to 0:

R1

Current node being removed: P2

Topological sort (Re-doing the first example)

2. **While the list of "nodes with outgoing degree equal to 0" is non-empty**, keep removing a node…



Nodes in graph: 2

Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	1
R1	[P1]	0

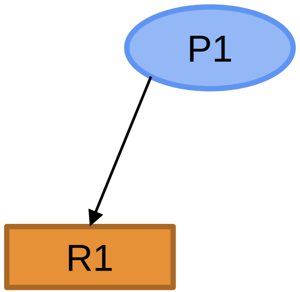
Nodes with outgoing degree equal to 0:

R1

<u>Current node being removed:</u>

Topological sort (Re-doing the first example)

2. While the list of “nodes with outgoing degree equal to 0” is non-empty, keep **removing a node**…



Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	1
R1	[P1]	0

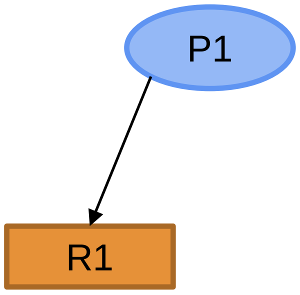
Nodes in graph: 2

Nodes with outgoing degree equal to 0:

<u>Current node being removed:</u> R1

Topological sort (Re-doing the first example)

2. While the list of “nodes with outgoing degree equal to 0” is non-empty, keep **removing a node**…



Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	1 -1
R1	[P1]	0

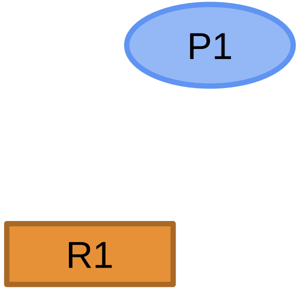
Nodes in graph: 2

Nodes with outgoing degree equal to 0:

<u>Current node being removed:</u> R1

Topological sort (Re-doing the first example)

2. While the list of “nodes with outgoing degree equal to 0” is non-empty, keep **removing a node**…



Nodes in graph: 2

Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	0
R1	[P1]	0

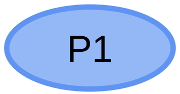
Nodes with outgoing degree equal to 0:

P1

Current node being removed: R1

Topological sort (Re-doing the first example)

2. **While the list of "nodes with outgoing degree equal to 0" is non-empty**, keep removing a node…



Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	0

Nodes in graph: 1

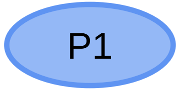
Nodes with outgoing degree equal to 0:

P1

Current node being removed:

Topological sort (Re-doing the first example)

2. While the list of “nodes with outgoing degree equal to 0” is non-empty, keep **removing a node**…



Nodes	Incoming Nodes	Outgoing degree
P1	[ ]	0

Nodes in graph: 1

Nodes with outgoing degree equal to 0:

Current node being removed: **P1**

Topological sort (Re-doing the first example)

2. **While the list of "nodes with outgoing degree equal to 0" is non-empty**, keep removing a node…

The list of "nodes with outgoing degree equal to 0 is empty!

Nodes	Incoming Nodes	Outgoing degree

Nodes in graph: 0

Nodes with outgoing degree equal to 0:

Current node being removed:

Topological sort (Re-doing the first example)

3. Check if any nodes in graph remaining. If no nodes remaining, no cycle/deadlock!

No more nodes left.
No deadlock!

Nodes in graph: 0

Nodes	Incoming Nodes	Outgoing degree

Nodes with outgoing degree equal to 0:

Current node being removed:

- Thanks for coming to tutorial
- Questions