Deadlock

Disclaimer: some slides are adopted from Dr. Kulkarni's and book authors' slides with permission

Recap: Synchronization

- Race condition
  - A situation when two or more threads **read and write** shared data at the same time
- Critical section
  - Code sections of potential race conditions
- Mutual exclusion
  - If a thread executes its critical section, *no other threads* can enter their critical sections
- Peterson's solution
  - Software only solution providing mutual exclusion

Recap: Synchronization

- Spinlock
  - Spin on waiting
  - Use synchronization instructions (test&set)
- Mutex
  - Sleep on waiting
- Semaphore
  - Powerful tool, but often difficult to use
- Monitor
  - Powerful and (relatively) easy to use

Quiz

```
Mutex lock;
Condition full, empty;

produce (item)
{
    ______________
    while (queue.isFull())
        empty.wait(&lock);
    queue.enqueue(item);
    full.signal();
    ______________
}

consume()
{
    ______________
    while (queue.isEmpty())
        ______________
    item = queue.dequeue(item);
    ______________
    ______________
    return item;
}
```

```
Semaphore mutex = 1, full = 0,
empty = N;

produce (item)
{
    P (&empty)_________;
    P(&mutex);
    queue.enqueue(item);
    V(&mutex);
    V (&full)_________;
}

consume()
{
    P (&full)_________;
    P(&mutex);
    item = queue.dequeue();
    V(&mutex);
    V (&empty)_________;
    return item;
}
```

Quiz

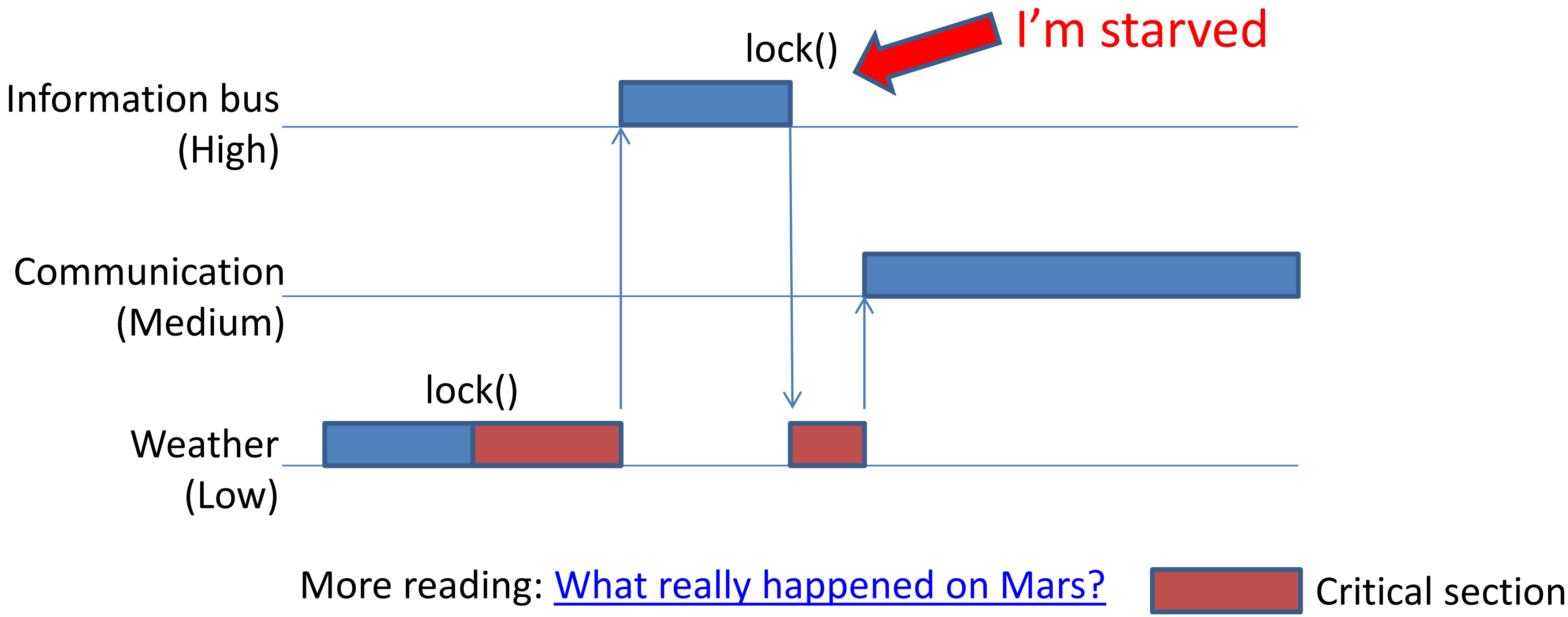
```
Mutex lock;
Condition full, empty;

produce (item)
{
    lock.acquire();
    while (queue.isFull())
        empty.wait(&lock);
    queue.enqueue(item);
    full.signal();
    lock.release();
}

consume()
{
    lock.acquire();
    while (queue.isEmpty())
        full.wait(&lock);
    item = queue.dequeue(item);
    empty.signal();
    lock.release();
    return item;
}
```

```
Semaphore mutex = 1, full = 0,
empty = N;

produce (item)
{
    P(&empty);
    P(&mutex);
    queue.enqueue(item);
    V(&mutex);
    V(&full);
}

consume()
{
    P(&full);
    P(&mutex);
    item = queue.dequeue();
    V(&mutex);
    V(&empty);
    return item;
}
```

Agenda

- Deadlock
  - Starvation vs. deadlock
  - Deadlock conditions
  - General solutions: detection and prevention
  - Detection algorithm
  - Banker's algorithm

Starvation



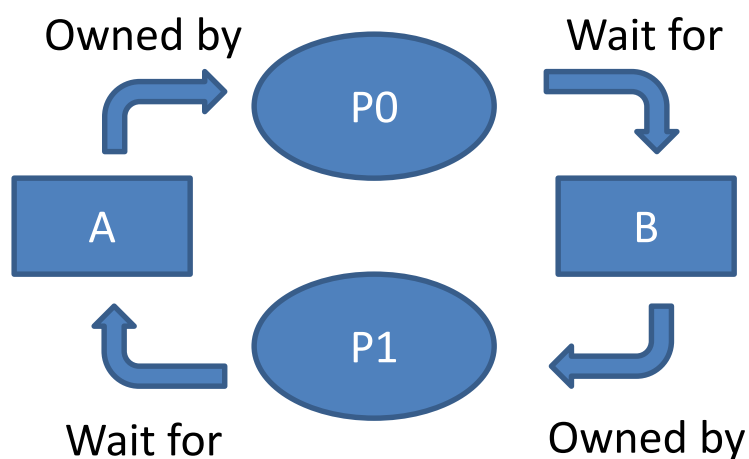
More reading: [What really happened on Mars?]

- Starvation
  - Wait potentially **indefinitely,** but it **can end**

Starvation vs. Deadlock

- Deadlock: circular waiting for resources
  - Example: semaphore A = B = 1

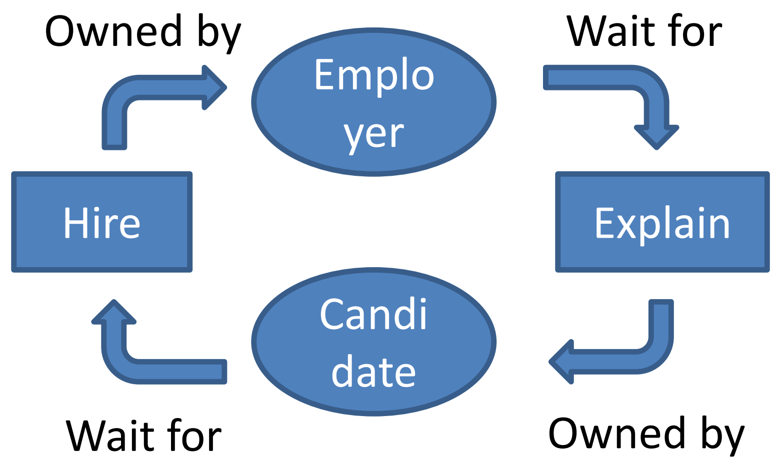
P0	**P1**
P(A)	P(B)
P(B)	P(A)



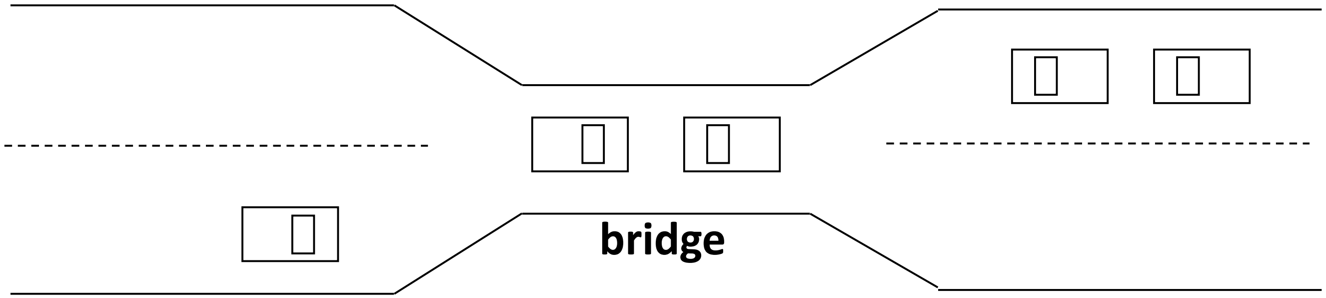
- Deadlock ➔ Starvation
  - But reverse is not true
  - Deadlock can't end but starvation can

Deadlock


IMPERIAL COLLEGE OF
Explain Deadlock, and we'll hire you.
Hire me, and I'll explain it to you.
let's discuss the salary.


Owned by
Emplo
yer
Wait for
Hire
Explain
Candi
date
Wait for
Owned by

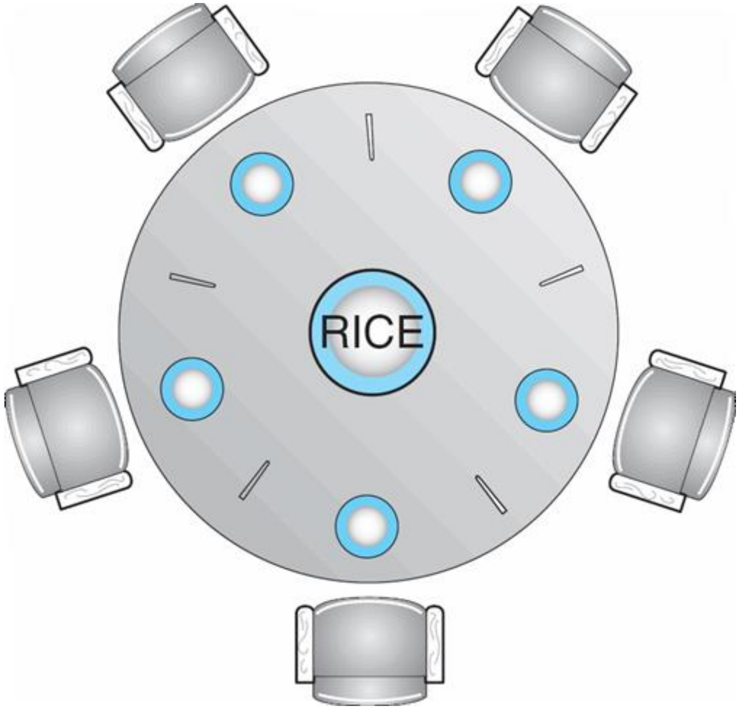
Bridge Crossing



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, how to fix it?
  - Make one car backs up
  - Several cars may have to be backed up if a deadlock occurs

Dining Philosophers

- Problem synopsis
  - Need two chopsticks to eat
  - Grab one chopsticks at a time
- What happens if all grab left chopstick at the same time??
  - Deadlock!!!
- How to fix it?
- How to avoid it?



Conditions for Deadlocks

- Mutual exclusion
  - only one process at a time can use a resource
- No preemption
  - resources cannot be preempted, release must be voluntary
- Hold and wait
  - a process must be holding at least one resource, and waiting to acquire additional resources held by other processes
- Circular wait
  - There must be a circular dependency. For example, A waits B, B waits C, and C waits A.

- **All four conditions must simultaneously hold**

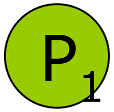
KU THE UNIVERSITY OF KANSAS

Resource-Allocation Graph

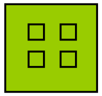
- To illustrate deadlock conditions.
- Graph consists of a set of vertices V and a set of edges E
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph

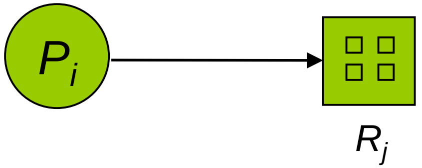
- Process



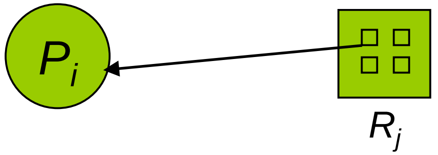
- Resource Type with 4 instances



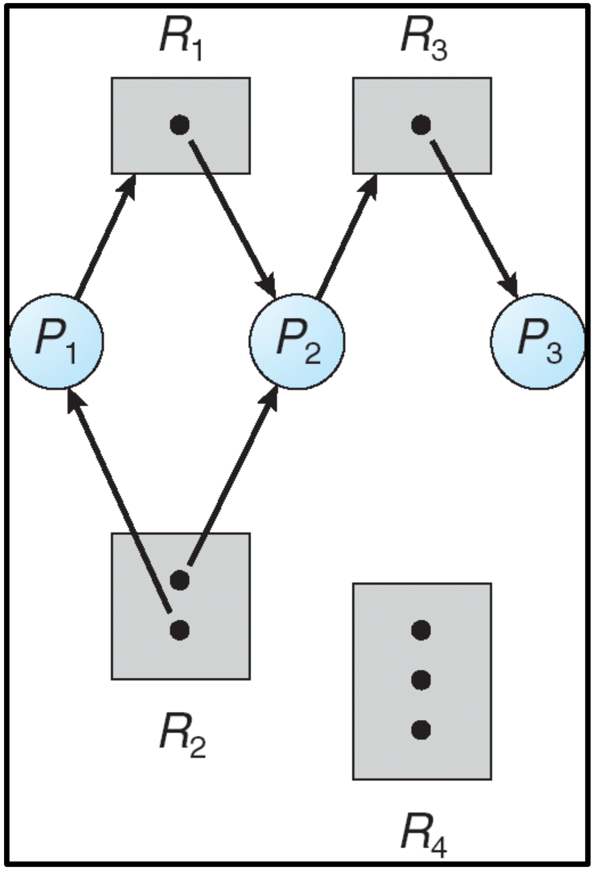
- P_i requests instance of R_j



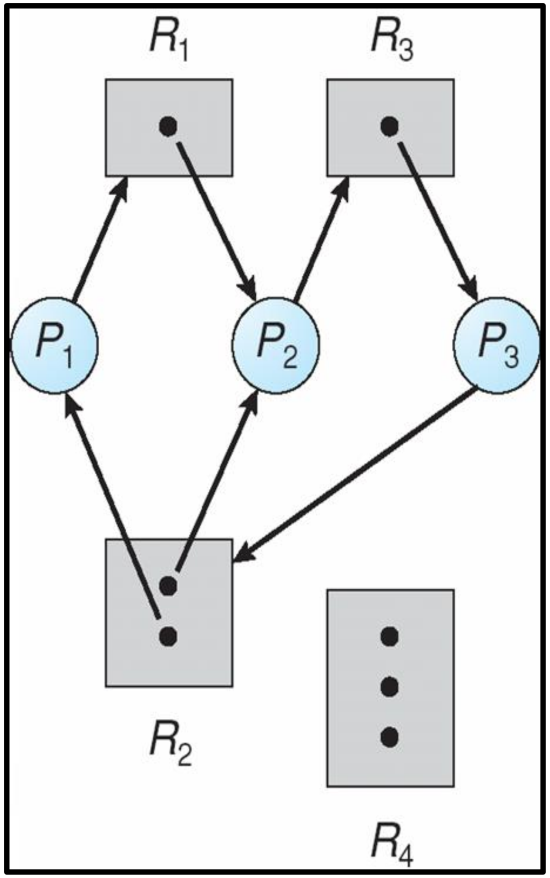
- P_i is holding an instance of R_j



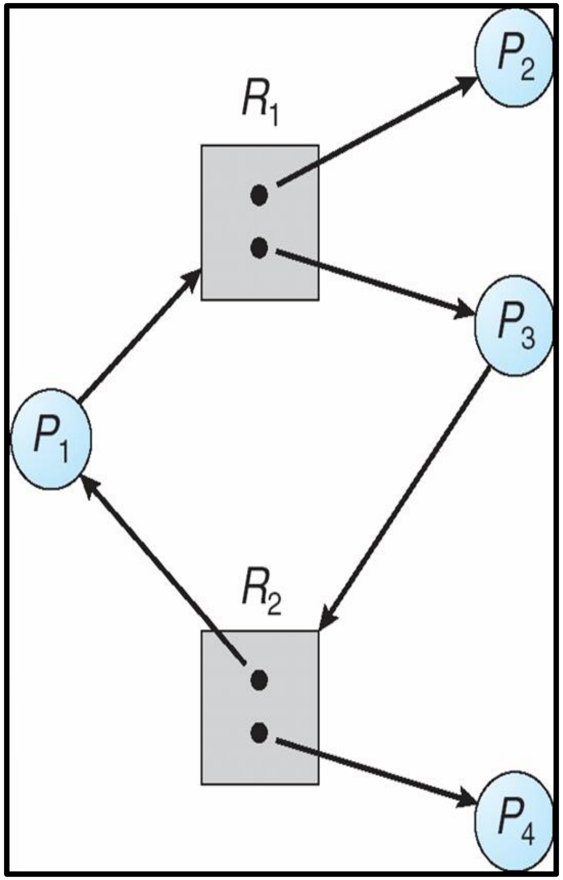
Resource Allocation Graph



Simple example



Deadlock example



With cycle, but no deadlock

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

Methods for Handling Deadlocks

- Detection and recovery
  - Allow a system to enter a deadlock and then recover
    - Need a *detection algorithm*
    - Somehow "preempt" resources
- Prevention and avoidance
  - Ensure a system never enter a deadlock
  - Possible solutions
    - have "Infinite resources"
    - prevent "hold and wait"
    - prevent "circular wait"

Recall four deadlock conditions:
(1) Mutual exclusion, (2) no preemption, (3) hold and wait, (4) circular wait

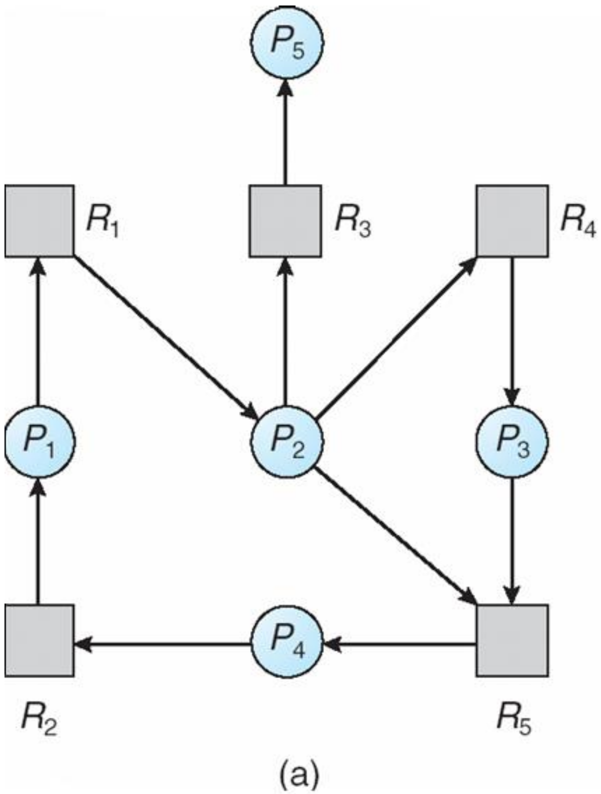
Deadlock Detection

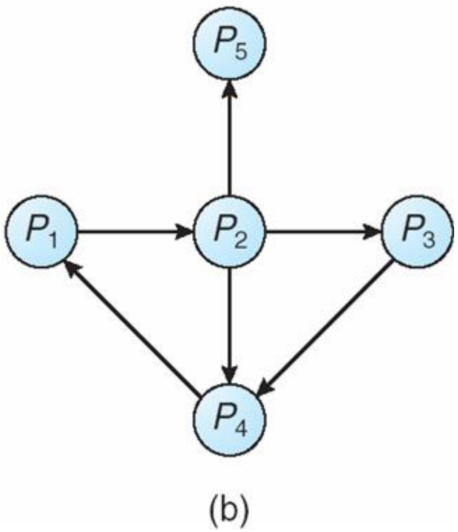
- Deadlock detection algorithms
  - Single instance for each resource type
  - Multiple instances for each resource type

Single Instance Per Resource

- Each resource is unique
  - E.g., one printer, one audio card, …
- Wait-for-graph
  - Variant of the simplified resource allocation graph
  - Remove resource nodes, collapse corresponding edges
- Detection algorithm
  - Searches for a cycle in the wait-for graph
  - Presence of a cycle points to the existence of a deadlock

Wait-for Graph

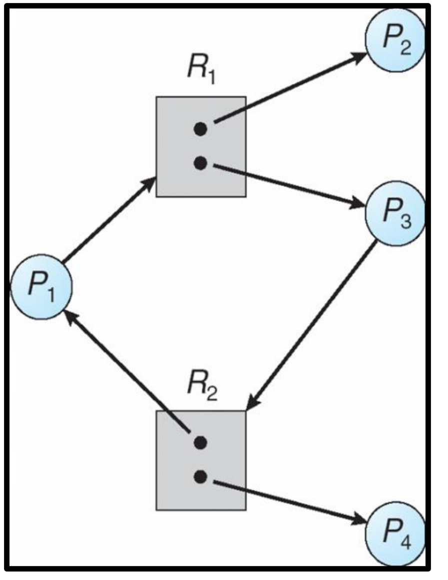




Resource-Allocation Graph Corresponding wait-for graph

Multiple Instances Per Resource

- **n** processes, **m** resources
- **FreeResources**: resource vector (of size m)
  - indicates the number of available resources of each type
  - [R1, R2] = [0,0]
- **Alloc[i]:** process i's allocated resource vector
  - defines the number of resources of each type currently allocated to each process
  - Alloc[1] = [0,1],
  - Alloc[2] = [1, 0], …
- **Request[i]:** process i's requesting resource vector
  - indicates the resources each process requests
  - Request[1] = [1,0],
  - Request[2] = [0,0], …



Detection Algorithm

1. Initialize **Avail** and **Finish** vectors

   Avail = FreeResources;

   For i = 1,2, …, n, Finish[i] = false

2. Find an index i such that

   Finish[i] == false AND Request[i] ≤ Avail

   If no such i exists, go to step 4

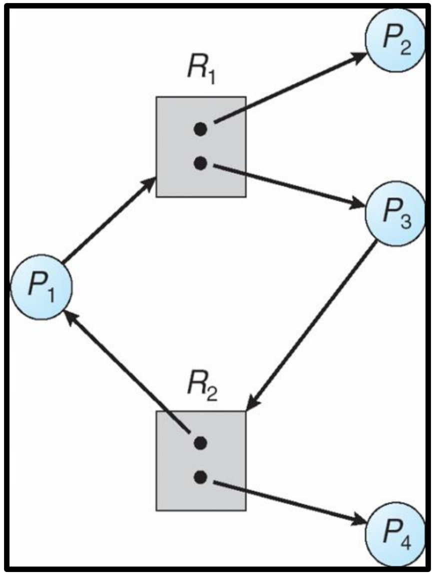
3. Avail = Avail + Alloc[i] , Finish[i] = true

   Go to step 2

4. If Finish[i] == false, for some i, 1 ≤ i ≤ n,

   (a) then the system is in deadlock state

- **FreeResources**: resource vector [R1, R2] = [0,0]
- **Alloc[i]**: process i's allocated resource vector: Alloc[1] = [0,1], Alloc[2] = [1, 0]
- **Request[i]**: process i's requesting vector: Request[1] = [1,0] Request[2] = [0,0]



Recovery from Deadlock

- Terminate
  - Preempt the resources
  - Bridge example: throw the car to the river
  - Kill the deadlocked threads and return the resources

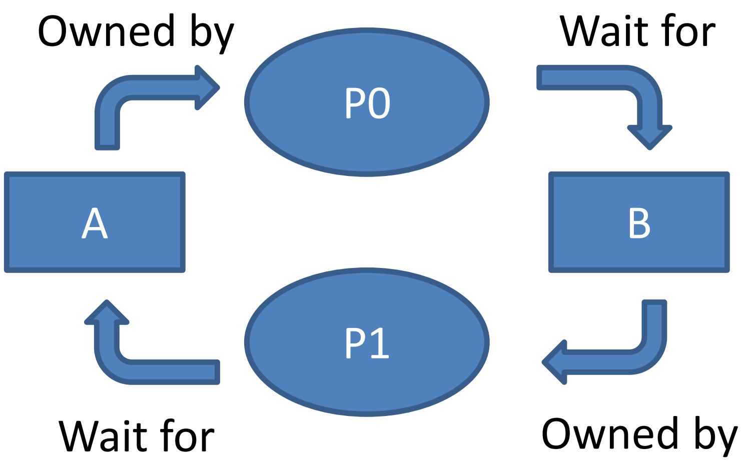
- Rollback
  - Return to a known safe state
  - Bridge example: move one car backward
  - Dining philosopher: make one philosopher give up a chopstick

- Not always possible!

Recap: Starvation vs. Deadlock

- Deadlock: circular waiting for resources
  - Example: semaphore A = B = 1

P0	**P1**
P(A)	P(B)
P(B)	P(A)



- Deadlock ➔ Starvation
  - But reverse is not true
  - Deadlock can’t end but starvation can

Recap: Conditions for Deadlocks

- Mutual exclusion
  - only one process at a time can use a resource
- No preemption
  - resources cannot be preempted, release must be voluntary
- Hold and wait
  - a process must be holding at least one resource, and waiting to acquire additional resources held by other processes
- Circular wait
  - There must be a circular dependency. For example, A waits B, B waits C, and C waits A.

- **All four conditions must simultaneously hold**

Recap: Detection Algorithm

1. Initialize **Avail** and **Finish** vectors

   Avail = FreeResources;

   For i = 1,2, …, n, Finish[i] = false

2. Find an index i such that

   Finish[i] == false AND Request[i] $\leq$ Avail

   If no such i exists, go to step 4

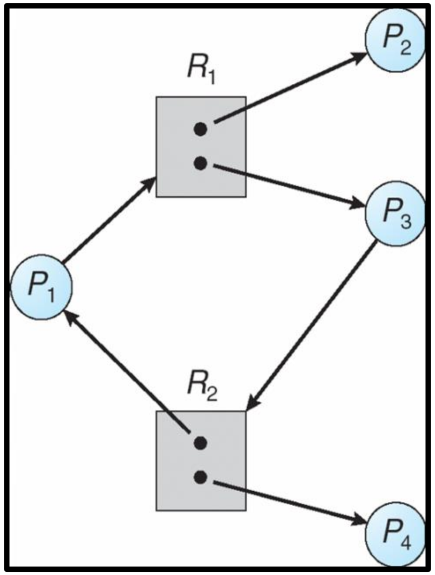
3. Avail = Avail + Alloc[i] , Finish[i] = true

   Go to step 2

4. If Finish[i] == false, for some i, $1 \leq i \leq n$,

   (a) then the system is in deadlock state

- **FreeResources**: resource vector [R1, R2] = [0,0]
- **Alloc[i]**: process i's allocated resource vector: Alloc[1] = [0,1], Alloc[2] = [1, 0]
- **Request[i]**: process i's requesting vector: Request[1] = [1,0] Request[2] = [0,0]



Recap: Recovery from Deadlock

- Terminate
  - Preempt the resources
  - Bridge example: throw the car to the river
  - Kill the deadlocked threads and return the resources

- Rollback
  - Return to a known safe state
  - Bridge example: move one car backward
  - Dining philosopher: make one philosopher give up a chopstick

- Not always possible!

Deadlock Prevention

- Break any of the four deadlock conditions
  - Mutual exclusion
  - No preemption
  - Hold and wait
  - Circular wait

Deadlock Prevention

- Break any of the four deadlock conditions
  - **Mutual exclusion → allow sharing**
    - Well, not all resources are sharable
  - No preemption
  - Hold and wait
  - Circular wait

Deadlock Prevention

- Break any of the four deadlock conditions
  - Mutual exclusion → allow sharing
    - Well, not all resources are sharable
  - **No preemption → allow preemption**
    - This is also quite hard (kill the threads)
  - Hold and wait
  - Circular wait

Deadlock Prevention

- Break any of the four deadlock conditions
  - Mutual exclusion → allow sharing
    - Well, not all resources are sharable
  - No preemption → allow preemption
    - This is also quite hard (kill the threads)
  - **Hold and wait → get all resources at once**
    - Dining philosopher: get *both* chopsticks or *none*
  - Circular wait

Deadlock Prevention

- Break any of the four deadlock conditions
  - Mutual exclusion → allow sharing
    - Well, not all resources are sharable
  - No preemption → allow preemption
    - This is also quite hard (kill the threads)
  - Hold and wait → get all resources at once
    - Dining philosopher: get *both* chopsticks or *none*
  - **Circular wait → prevent cycle**
    - Dining philosopher: change the chopstick picking order; **if grabbing a chopstick will form a cycle, prevent it.**

Banker's Algorithm

- General idea
  - Assume that each process's maximum resource demand is known in advance
    - Max[i] : process i's maximum resource demand vector
  - **Pretend** each request is granted, then run the deadlock detection algorithm
  - If a deadlock is detected, the do not grant the request to keep the system in a **safe** state

Banker's Algorithm

1. Initialize **Avail** and **Finish** vectors

   Avail = FreeResources;

   For i = 1,2, …, n, Finish[i] = false

2. Find an index i such that

   Finish[i] == false AND

   Max[i] – Alloc[i] $\leq$ Avail

   If no such i exists, go to step 4

3. Avail = Avail + Alloc[i] , Finish[i] = true

   Go to step 2

4. If Finish[i] == false, for some i, $1 \leq i \leq n$,

   (a) then the system is in deadlock state

   (b) if Finish[i] == false, then P_i is deadlocked

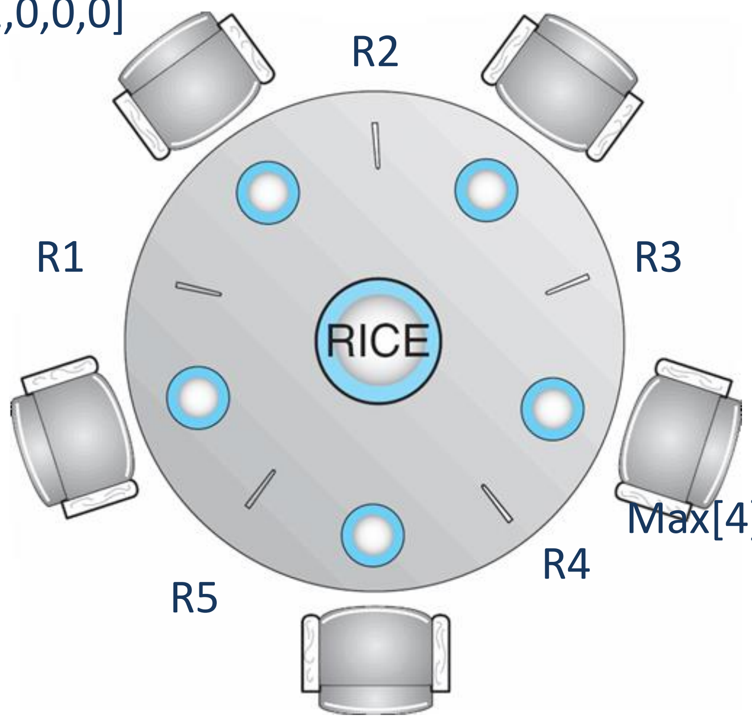
- FreeResources: resource vector [R1, R2] = [0,0]
- Alloc[i]*:* process i's allocated resource vector: Alloc[1] = [0,1], Alloc[2] = [1, 0]
- Request[i]: process i's requesting vector: Request[1] = [1,0] Request[2] = [0,0]
- **Max[i]:** process i's maximum resource demand vector

Example

Free = [1,1,1,1,1]

Max[3] = [0,1,1,0,0]

Max[2] = [1,1,0,0,0]



Max[1] = [1,0,0,0,1]

Max[4] = [0,0,1,1,0]

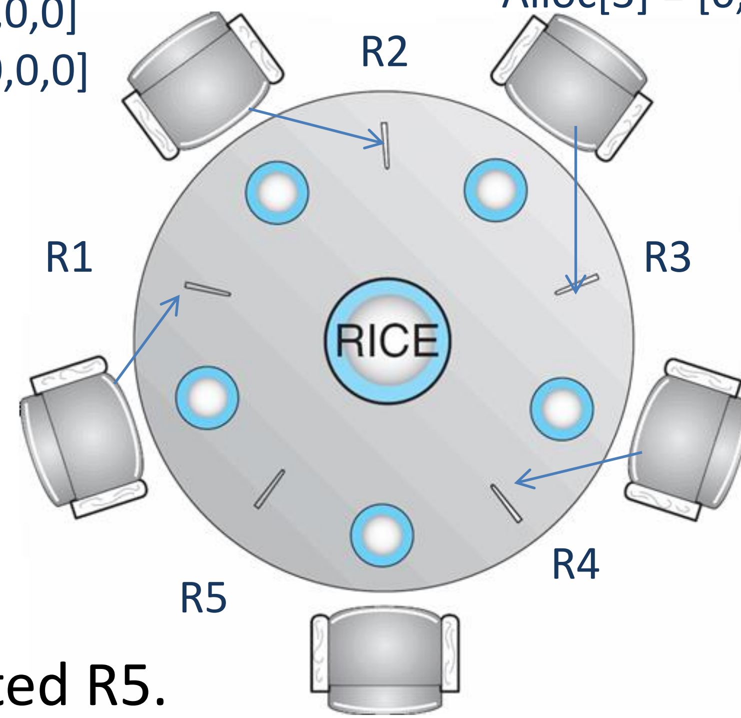
Max[5] = [0,0,0,1,1]

Example

Free = [0,0,0,0,1]
Avail = [0,0,0,0,1]

Max[3] = [0,1,1,0,0]
Alloc[3] = [0,0,1,0,0]

Max[2] = [1,1,0,0,0]
Alloc[2] = [0,1,0,0,0]



Max[1] = [1,0,0,0,1]
Alloc[1]= [1,0,0,0,0]

Max[4] = [0,0,1,1,0]
Alloc[4] = [0,0,0,1,0]

Max[5] = [0,0,0,1,1]
Alloc[5] = [0,0,0,0,0]

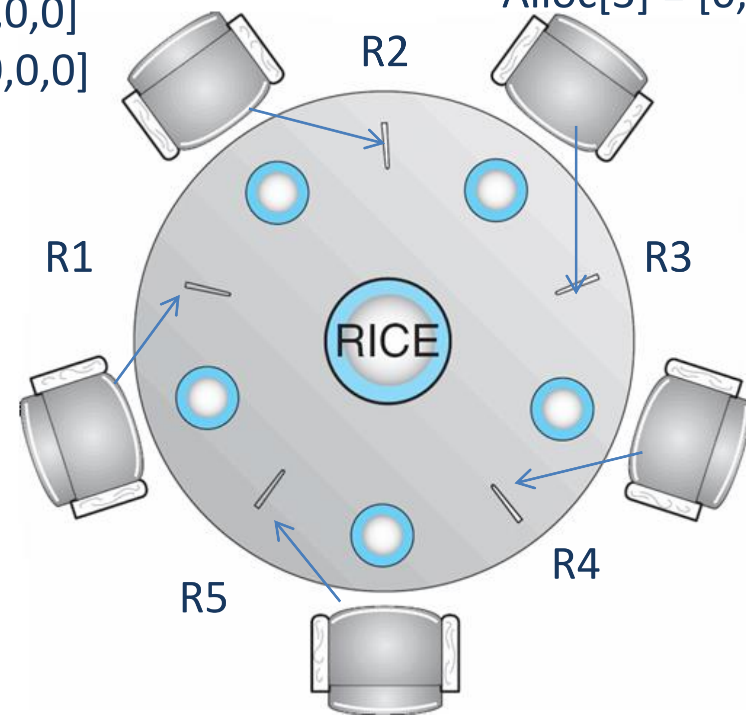
- Philosopher 5 requested R5.
- Safe or Unsafe?

Example

Free = [0,0,0,0,0]
Avail = [0,0,0,0,0]

Max[3] = [0,1,1,0,0]
Alloc[3] = [0,0,1,0,0]

Max[2] = [1,1,0,0,0]
Alloc[2] = [0,1,0,0,0]



Max[1] = [1,0,0,0,1]
Alloc[1]= [1,0,0,0,0]

Max[4] = [0,0,1,1,0]
Alloc[4] = [0,0,0,1,0]

Max[5] = [0,0,0,1,1]
Alloc[5] = [0,0,0,0,1]

2. Find an index i such that
   Finish[i] == false AND
      Max[i] – Alloc[i] $\leq$ Avail
   If no such i exists, go to step 4

Quiz

- Determine whether this state is safe or unsafe.

Total resources: 12

Avail resources: 3

Process	Max	Alloc
P_0	10	4
P_1	3	1
P_2	6	4

10 – 4 <= 8

3 – 1 <= 7

6 – 4 <= 3

Safe

Quiz

- Suppose P0 requested 3 additional resources. Should this request be granted?

Total resources: 12

Avail resources: 0

Process	Max	Alloc
P_0	10	7
P_1	3	1
P_2	6	4

$10 - 7 \not\le 0$

$3 - 1 \not\le 0$

$6 - 4 \not\le 0$

Unsafe

Quiz

- Suppose there are three resource types, which are needed by four processes. Is it safe now?

R1	R2	R3
2	1	2

← Free Resources

	Current Allocation			Max		
Process	R1	R2	R3	R1	R2	R3
P1	0	0	1	0	0	3
P2	2	0	0	2	4	5
P3	0	0	3	6	3	5
P4	2	3	5	4	3	5

← Current and Maximum Allocations

Quiz

- If P2 requests (1, 1, 2), should it be granted?

R1	R2	R3
2	1	2

← Free Resources

	Current Allocation			Max		
Process	R1	R2	R3	R1	R2	R3
P1	0	0	1	0	0	3
P2	2	0	0	3	4	5
P3	0	0	3	6	3	5
P4	2	3	5	4	3	5

← Current and Maximum Allocations

THE UNIVERSITY OF KANSAS

Quiz

- If P2 requests (1, 1, 2), should it be granted?

R1	R2	R3
1	0	0

← Free Resources

No

	Current Allocation			Max		
Process	R1	R2	R3	R1	R2	R3
P1	0	0	1	0	0	3
P2	3	1	2	3	4	5
P3	0	0	3	6	3	5
P4	2	3	5	4	3	5

← Current and Maximum Allocations

Summary

- Four deadlock conditions:
  - Mutual exclusion
  - No preemption
  - Hold and wait
  - Circular wait
- Detection
- Avoidance
  - Banker's algorithm