# Deadlocks

COMP 3361: Operating Systems I

Winter 2015

http://www.cs.du.edu/3361

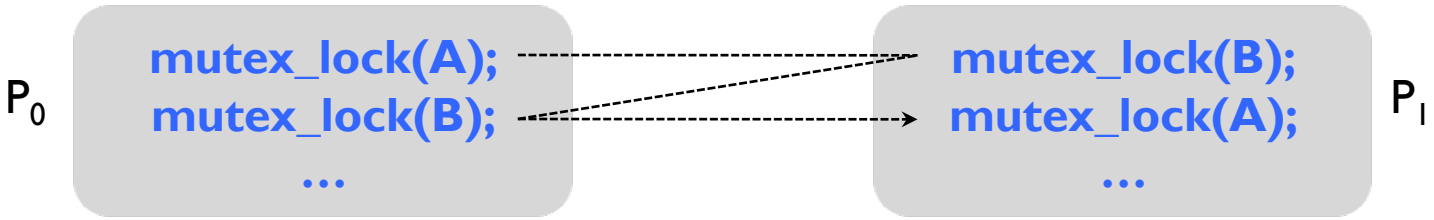
# 1 System Model

- Resource types $R_1, R_2, \ldots, R_m$
  - CPU cycles, I/O devices, mutex locks, semaphores, ...
  - resources cannot be preempted
- Each resource type $R_i$ has $E_i$ instances
- Each process utilizes a resource as follows
  - request
    - process waits if the request to acquire the resource is not granted immediately
  - use
    - process can operate on the resource
  - release
    - process no longer operates on the resource

# What is a Deadlock?

- A set of blocked processes
  - each holding a resource and
  - waiting to acquire a resource held by another process in the set

- Example
  - P1 has CD drive, needs disk drive; P2 has disk drive, needs CD drive
  - mutex A and B



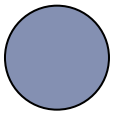
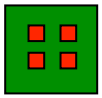
# 3 Conditions for Deadlocks

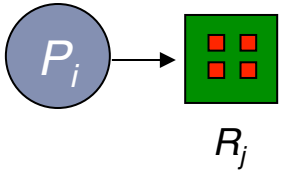
- Deadlock can arise if all of the following four conditions hold simultaneously
  - **mutual exclusion**: only one process at a time can use a resource
  - **hold and wait**: a process holding at least one resource can request additional resources (which may be held by other processes)
  - **no preemption**: a resource can be released only voluntarily
  - **circular wait**: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that $P_i$ is waiting for a resource that is held by $P_{(i+1)\%n}$

# 4 Resource-Allocation Graph

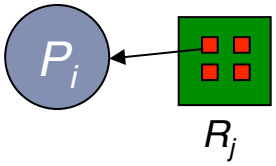
- A set of vertices V and edges E
- Vertices are partitioned into two types
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- An edge can be a
  - request edge: directed edge $P_i \rightarrow R_j$
    - $P_i$ wants $R_j$
  - assignment edge: directed edge $R_j \rightarrow P_i$
    - $R_j$ is with $P_i$

# Notations

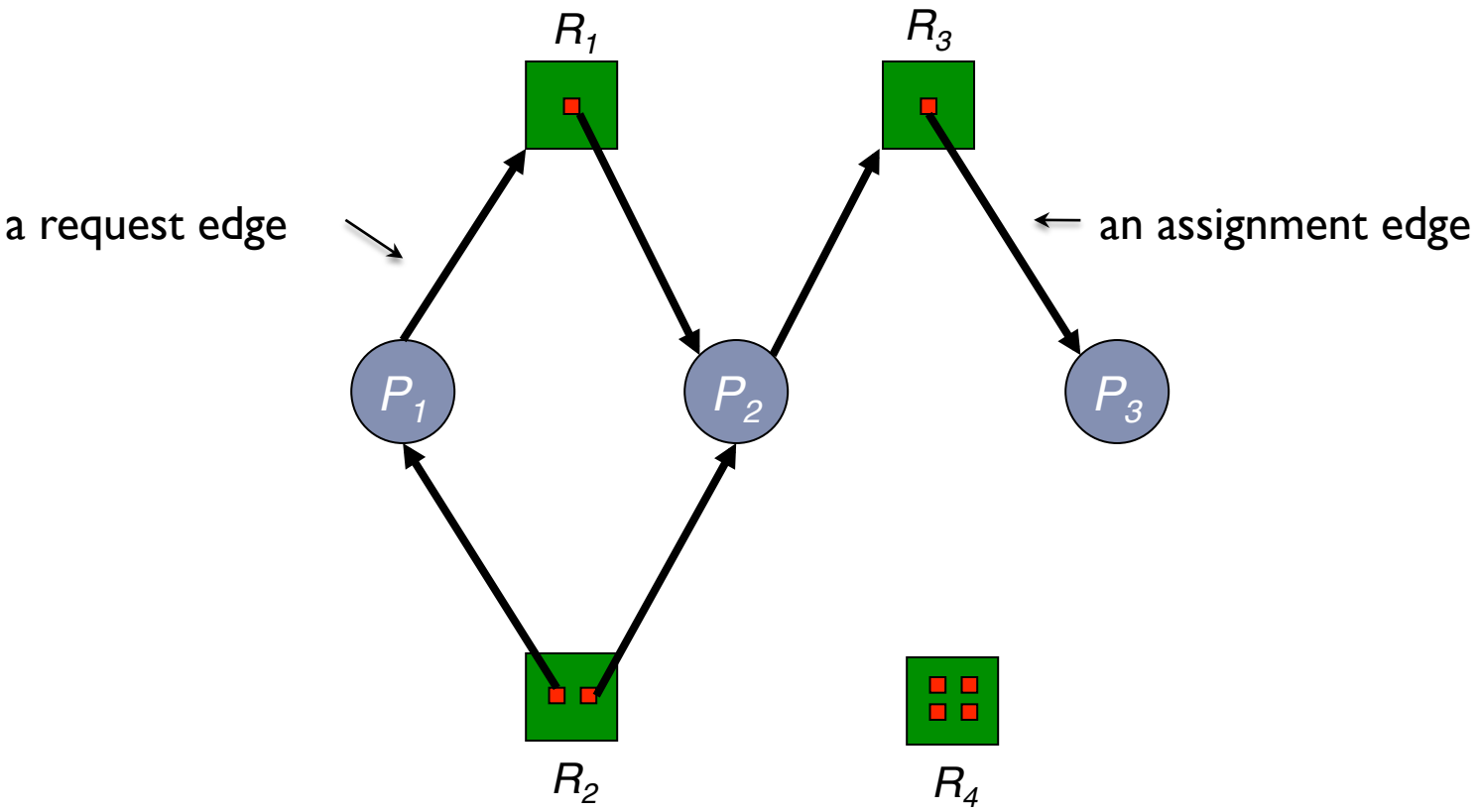
- Process 
- Resource with 4 instances 
- $P_i$ wants instance of $R_j$



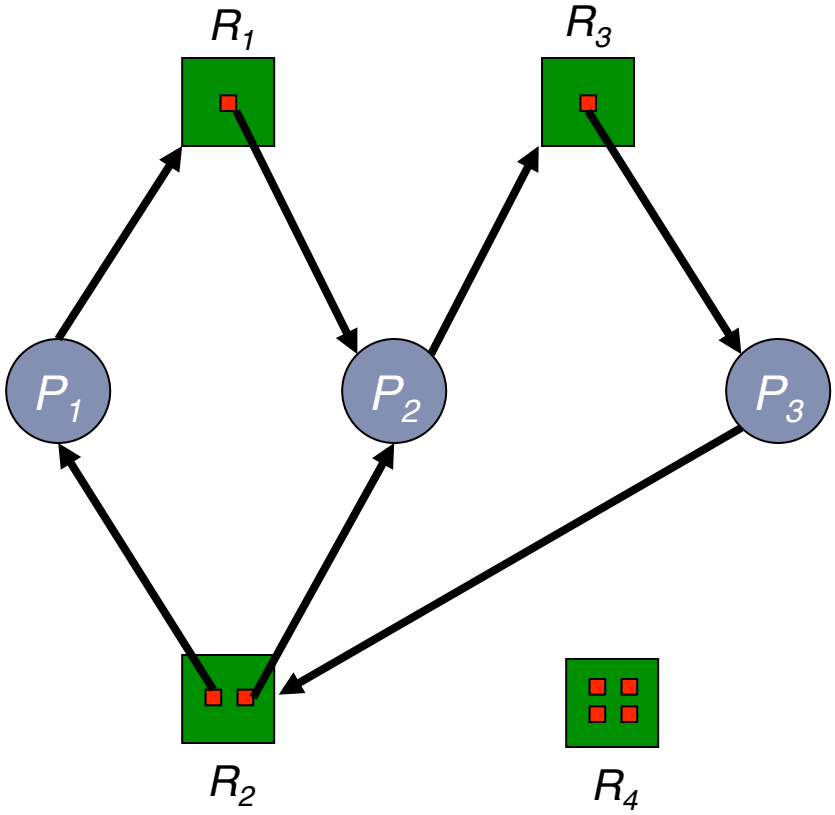

- $P_i$ has an instance of $R_j$



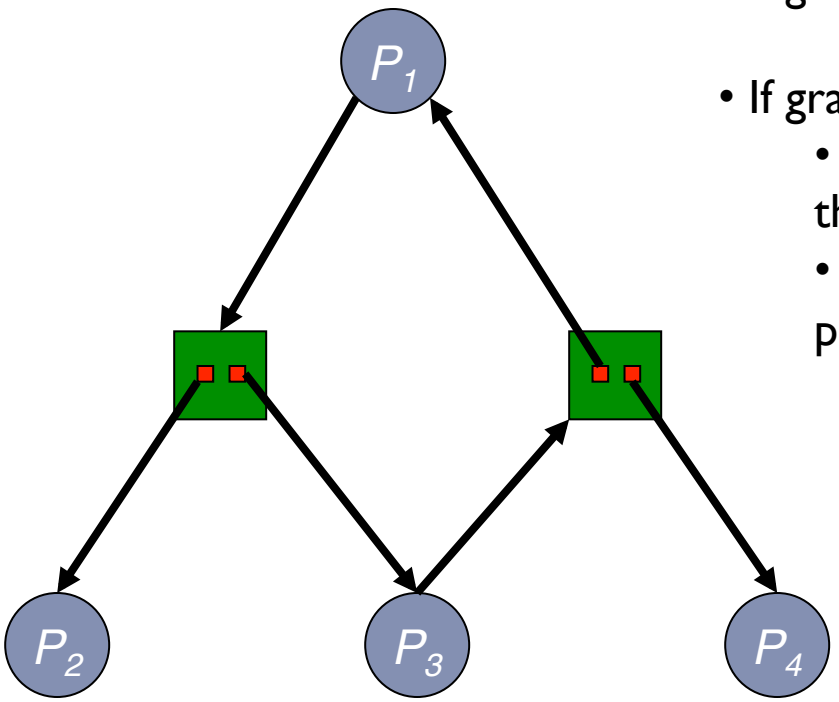
# Example



# Resource-Allocation Graph Deadlock



# 8 A Cycle Is Not Necessarily a Deadlock




- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

# Handling Deadlocks

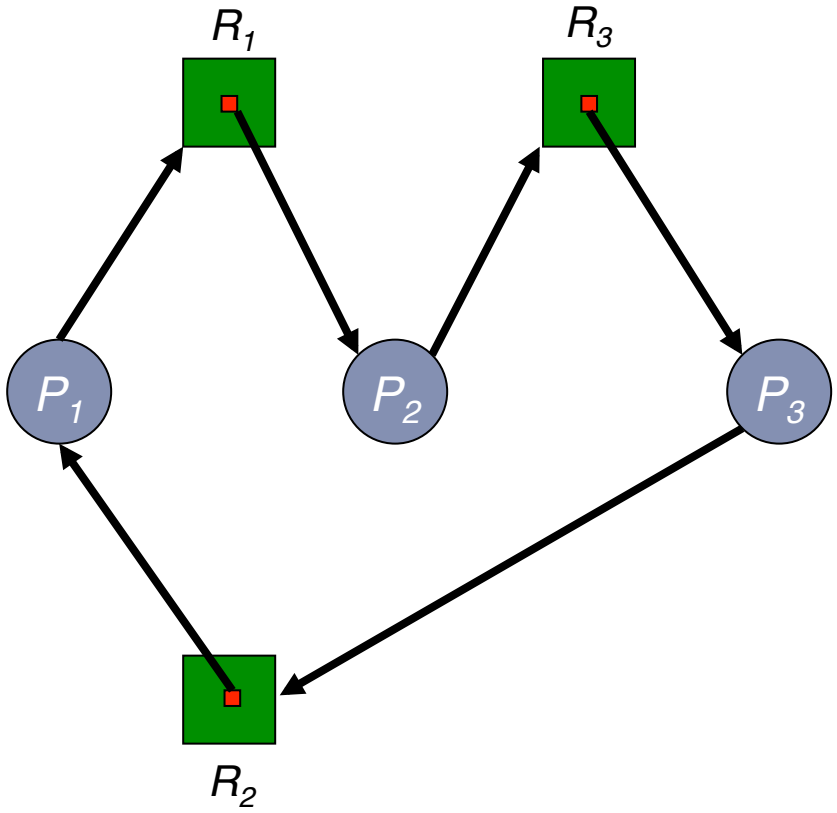
- Ignore the problem and pretend that deadlocks never occur in the system
  - used by most operating systems, including UNIX
- Allow the system to enter a deadlock state and then recover
  - deadlock detection
- Ensure that the system will never enter a deadlock state
  - deadlock avoidance
  - deadlock prevention

# Deadlock Detection

- Allow system to enter deadlock state
- Examine the state of the system to determine if a deadlock has occurred
  - check whenever a new request has been granted
  - check periodically
  - check when CPU utilization drops below a threshold
- Have a policy on what to do if a deadlock has occurred
  - recovery scheme

# Single Instance Resources

- A deadlock exists in the system if and only if there is a cycle in the resource allocation graph



# Some Notations

- Let n be number of processes and m be number of resource types
  - **Existing**: vector of length m. If Existing[j] = k, there are k total instances of resource type $R_j$
  - **Available**: vector of length m. If Available[j] = k, there are k instances of resource type $R_j$ available
  - **Allocation**: n x m matrix. If Allocation[i,j] = k, then $P_i$ currently has k instances of $R_j$
  - **Request**: n x m matrix. If Request[i,j] = k, then $P_i$ wants k more instances of $R_j$

# Some Notations

- **$Request_i$**
  - refers to the vector of length m specifying the number of instances of each resource type that process $P_i$ wants
  - $Request_i$ = (Request[i,1],…,Request[i,m])
- Similarly, $Allocation_i$
- Given two vectors $X = (x_1,\ldots,x_m)$ and $Y = (y_1,\ldots,y_m)$ we shall say
  - $X \leq Y$ if $x_i \leq y_i$ for all $i = 1,\ldots,m$
  - $X \neq 0$ if $x_i \neq 0$ for all $i = 1,\ldots,m$

# Multiple Instance Resources

***Finish*** is a boolean vector of length *n*.

1. Let ***Finish[i] = false*** for all *i* with ***Allocation***$_i \neq 0$; otherwise ***Finish[i] = true***

2. Find an *i* such that
   ***Finish[i] == false*** and ***Request***$_i \leq$ ***Available***
   **Goto 4** if no such *i* is found

3. ***Available = Available + Allocation***$_i$
   ***Finish[i] = true***
   **Goto 2**

4. If ***Finish[i] == false*** for some *i*, then process $P_i$ is in a deadlocked state

# Detection Algorithm Example

5 processes; 3 resources types (A, B and C)
7 instances of A, 2 of B and 6 of C

| | Allocation | | | Request | | | Available | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | A | B | C | A | B | C |
| $P_0$ | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $P_1$ | 2 | 0 | 0 | 2 | 0 | 2 | | | |
| $P_2$ | 3 | 0 | 3 | 0 | 0 | 0 | | | |
| $P_3$ | 2 | 1 | 1 | 1 | 0 | 0 | | | |
| $P_4$ | 0 | 0 | 2 | 0 | 0 | 2 | | | |

state at a certain point in time

Not Deadlocked:: sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in

***Finish[i] = true*** for all ***i***

# Detection Algorithm Example

| | Allocation | | | Request | | | Available | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | A | B | C | A | B | C |
| $P_0$ | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $P_1$ | 2 | 0 | 0 | 2 | 0 | 2 | | | |
| $P_2$ | 3 | 0 | 3 | 0 | 0 | 1 | | | |
| $P_3$ | 2 | 1 | 1 | 1 | 0 | 0 | | | |
| $P_4$ | 0 | 0 | 2 | 0 | 0 | 2 | | | |

After $P_0$ executes, ***Available = (0, 1, 0)***;
but after that no other process can progress

$P_1$, $P_2$, $P_3$ and $P_4$ are in a deadlocked state

# Recovery Schemes

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- Abort a process that is not deadlocked

- Preemption: take resource away from another process

- Rollback: return to some safe state (when resource was not needed) and resume process from that state
  - much easier to terminate and restart

# Deadlock Avoidance

- Requires that each process declare the maximum number of resources of each type that it may need
- Deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
  - resource-allocation state is the number of available and allocated resources, and the maximum demands of the processes

# Safe State

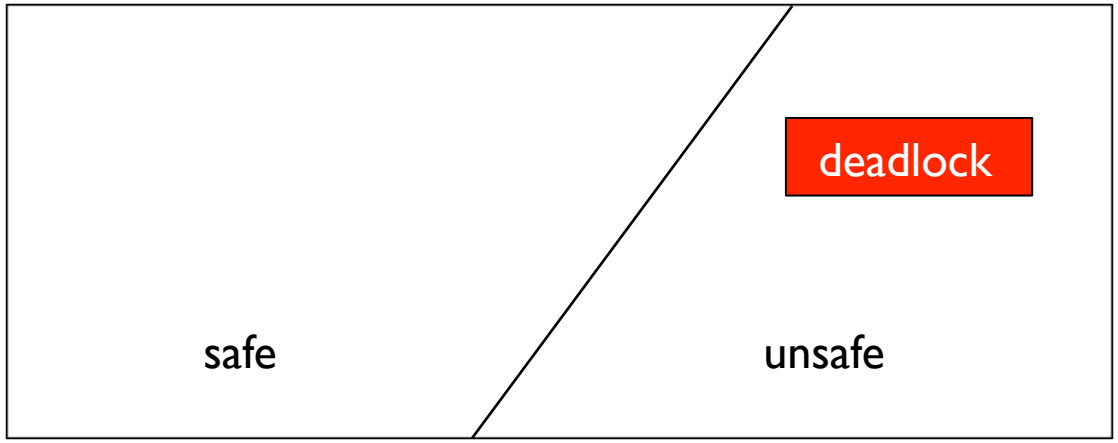
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- **Safe state**: there exists a safe sequence $<P_1, P_2, \ldots, P_n>$ of all processes in the system such that
  - for each $P_i$, the resources that $P_i$ may still request can be satisfied by
    - currently available resources, and
    - resources held by all the $P_j$ with $j < i$
- If no such sequence exists, then system is **unsafe**

# Execution in a Safe State

- If $P_i$'s resource needs are not immediately met, then $P_i$ can wait until all $P_j$ have finished ( $j < i$ )
- When $P_j$ is (are) finished, $P_i$ can obtain needed resources, execute, return allocated resources, and terminate
- When $P_i$ terminates, $P_{i+1}$ may obtain its needed resources
- …and so on

# Deadlocks and Safe State

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state



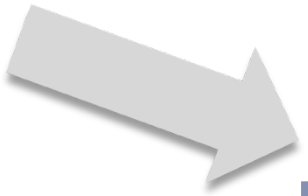
# Safe To Unsafe

3 more available

| | Maximum Need | Currently Has |
|---|---|---|
| $P_0$ | 10 | 5 |
| $P_1$ | 4 | 2 |
| $P_2$ | 9 | 2 |

assume there are 12 instances of a resource

safe sequence: $<P_1, P_0, P_2>$



$P_2$ requests one more and is given

| | Maximum Need | Currently Has |
|---|---|---|
| $P_0$ | 10 | 5 |
| $P_1$ | 4 | 2 |
| $P_2$ | 9 | **3** |

2 more available

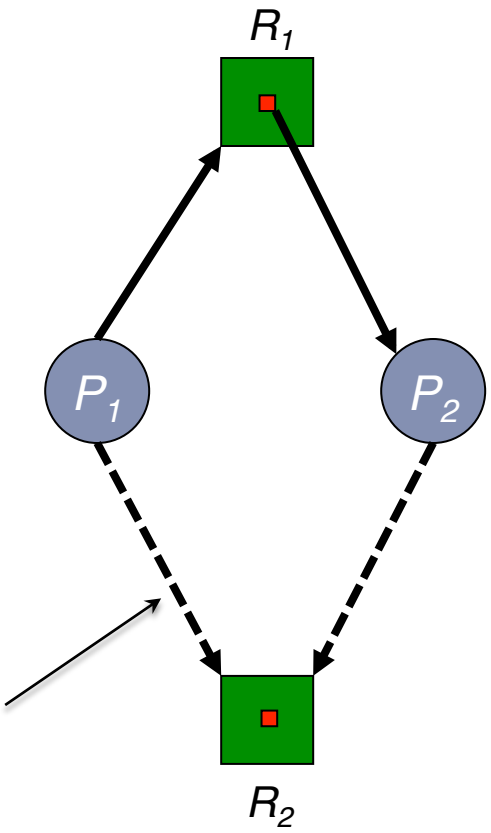
safe sequence: ??

**unsafe state !!**

# Avoidance Algorithms

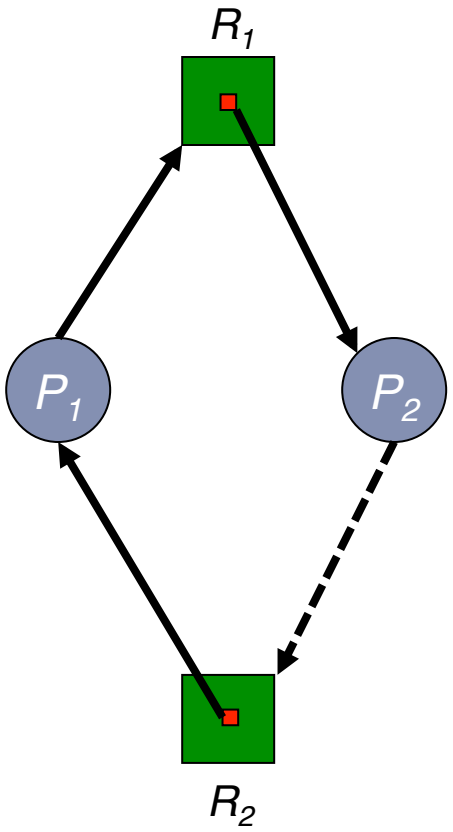
- Single instance of a resource type
  - use a resource-allocation graph
- Multiple instances of a resource type
  - use the Banker's algorithm

# Example




a claim edge
(a request that will be made in future)

# Unsafe State Example




**$P_1$ requests $R_2$**
**Should it be given?**

If $R_2$ is assigned to $P_1$, neither $<P_1, P_2>$ nor $<P_2, P_1>$ is a safe sequence; there is a cycle in the graph

**So, NO!**

# Banker's Algorithm

- Used when resources can have multiple instances
- Each process must declare the maximum number of instances of each resource type that it may need
- When a process requests a resource it may have to wait
- When a process gets all its resources, it must return them in a finite amount of time
- **Need**: n x m matrix. If Need[i,j] = k, then $P_i$ needs k more instances of $R_j$ to complete its task

# Safety Algorithm

- Determine if a system is in a safe state or not

***Finish*** is a boolean vector of length *n*.

1. Let ***Finish[i] = false*** for all *i*

2. Find an *i* such that
   ***Finish[i] == false*** and $Need_i \leq Available$
   **Goto 4** if no such *i* is found

3. $Available = Available + Allocation_i$
   ***Finish[i] = true***
   **Goto 2**

4. If ***Finish[i] == true*** for all *i*, then system is in a safe state

A request can be granted only if it ensures that the system will still be in a safe state

# Banker's Algorithm Example

5 processes; 3 resources types (A, B and C)
10 instances of A, 5 of B and 7 of C

Each process must declare how much it will ever need

| | Allocation | | | Max | | | Available | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | A | B | C | A | B | C |
| $P_0$ | 0 | 1 | 0 | 7 | 5 | 3 | 3 | 3 | 2 |
| $P_1$ | 2 | 0 | 0 | 3 | 2 | 2 | | | |
| $P_2$ | 3 | 0 | 2 | 9 | 0 | 2 | | | |
| $P_3$ | 2 | 1 | 1 | 2 | 2 | 2 | | | |
| $P_4$ | 0 | 0 | 2 | 4 | 3 | 3 | | | |

state at a certain point in time

# Banker's Algorithm Example

state is safe; $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ is a safe-sequence

| | Allocation | | | Max | | | Available | | | Need | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| $P_0$ | 0 | 1 | 0 | 7 | 5 | 3 | 3 | 3 | 2 | 7 | 4 | 3 |
| $P_1$ | 2 | 0 | 0 | 3 | 2 | 2 | | | | 1 | 2 | 2 |
| $P_2$ | 3 | 0 | 2 | 9 | 0 | 2 | | | | 6 | 0 | 0 |
| $P_3$ | 2 | 1 | 1 | 2 | 2 | 2 | | | | 0 | 1 | 1 |
| $P_4$ | 0 | 0 | 2 | 4 | 3 | 3 | | | | 4 | 3 | 1 |

# Example: Request$_1$ = (1,0,2)

- Assume allocation is done:

| | Allocation | | | Max | | | Available | | | Need | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| $P_0$ | 0 | 1 | 0 | 7 | 5 | 3 | **2** | **3** | **0** | 7 | 4 | 3 |
| $P_1$ | **3** | **0** | **2** | 3 | 2 | 2 | | | | **0** | **2** | **0** |
| $P_2$ | 3 | 0 | 2 | 9 | 0 | 2 | | | | 6 | 0 | 0 |
| $P_3$ | 2 | 1 | 1 | 2 | 2 | 2 | | | | 0 | 1 | 1 |
| $P_4$ | 0 | 0 | 2 | 4 | 3 | 3 | | | | 4 | 3 | 1 |

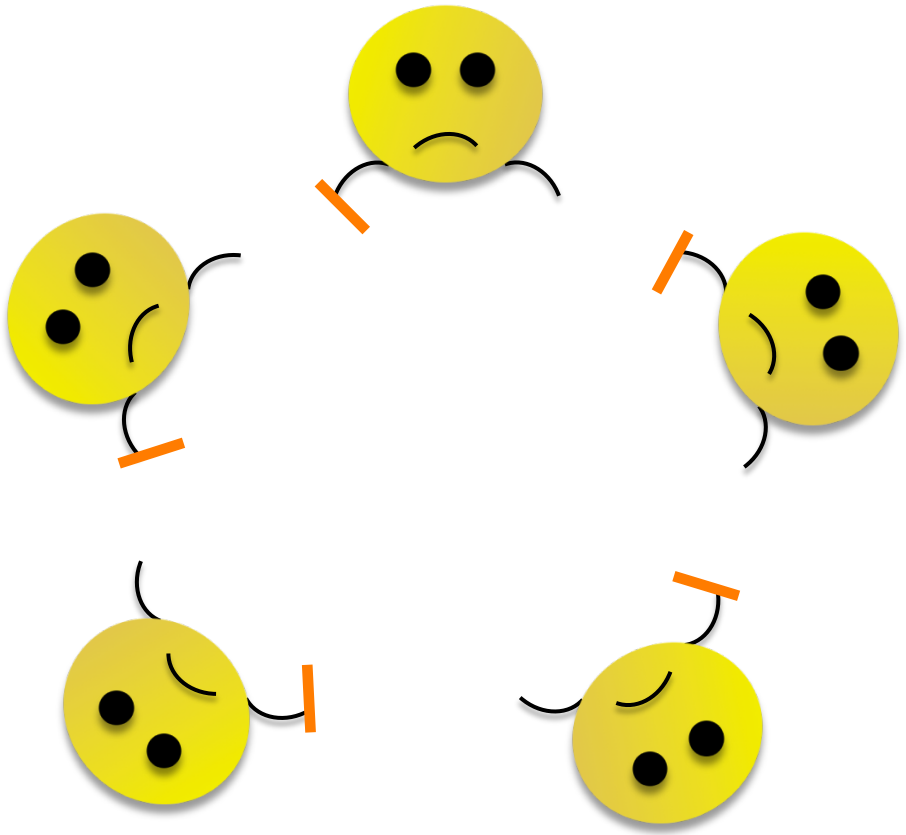
- Still a safe state: <P1, P3, P4, P0, P2> is the safe-sequence
  - P1 is given the resources it wants

# Example: Can We Do These?

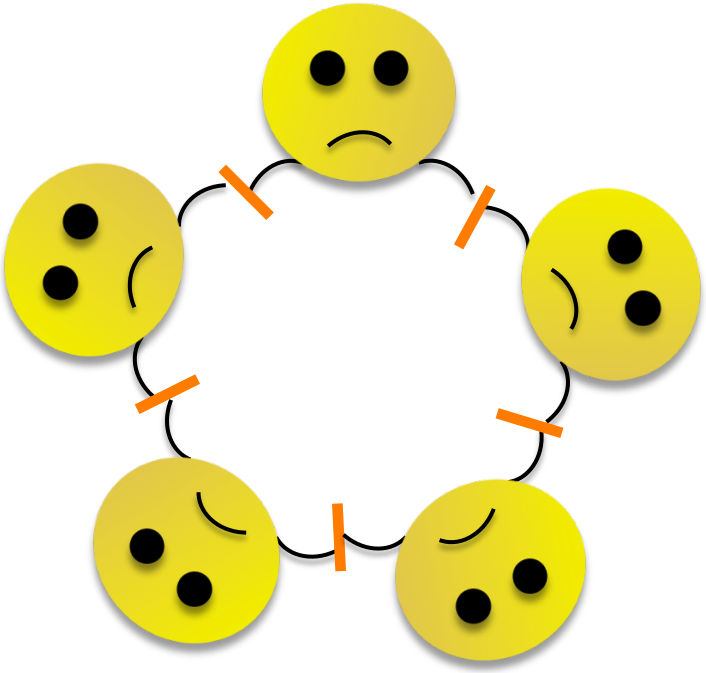
- Request$_4$ = (3, 3, 0)?
- Request$_0$ = (0, 2, 0)?

| | Allocation | | | Max | | | Available | | | Need | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | A | B | C | A | B | C | A | B | C |
| $P_0$ | 0 | 1 | 0 | 7 | 5 | 3 | 2 | 3 | 0 | 7 | 4 | 3 |
| $P_1$ | 3 | 0 | 2 | 3 | 2 | 2 | | | | 0 | 2 | 0 |
| $P_2$ | 3 | 0 | 2 | 9 | 0 | 2 | | | | 6 | 0 | 0 |
| $P_3$ | 2 | 1 | 1 | 2 | 2 | 2 | | | | 0 | 1 | 1 |
| $P_4$ | 0 | 0 | 2 | 4 | 3 | 3 | | | | 4 | 3 | 1 |

# Dining Philosophers in a Deadlock

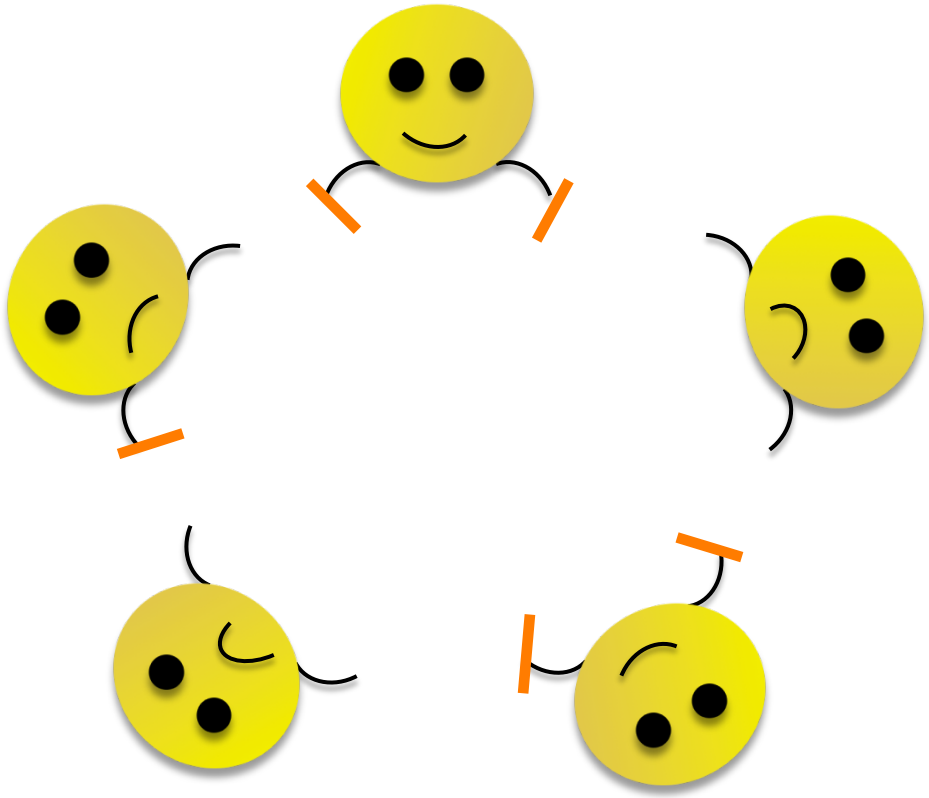


# Removing Mutual Exclusion



resources that are read-only can be concurrently accessed

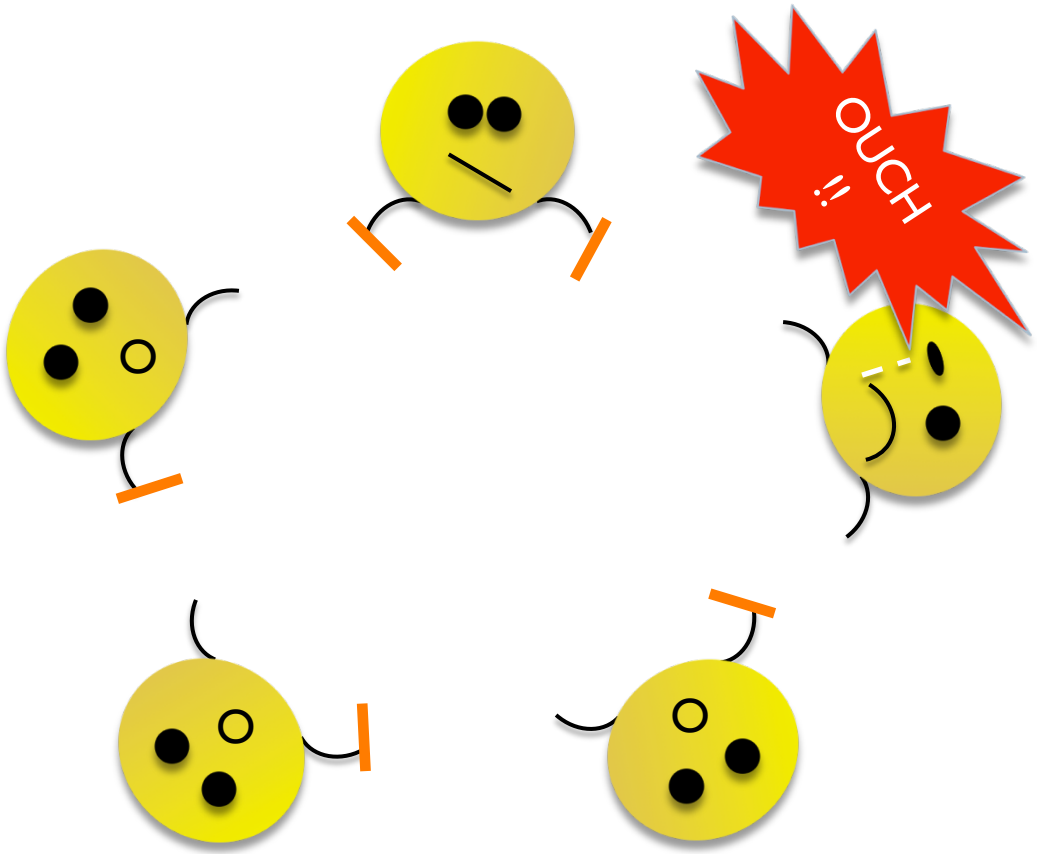
# Removing Hold and Wait



grab both chopsticks at the same time; philosopher wins if fast enough to get both

- must request all resources before using them
- release all resources on hold first, then request again

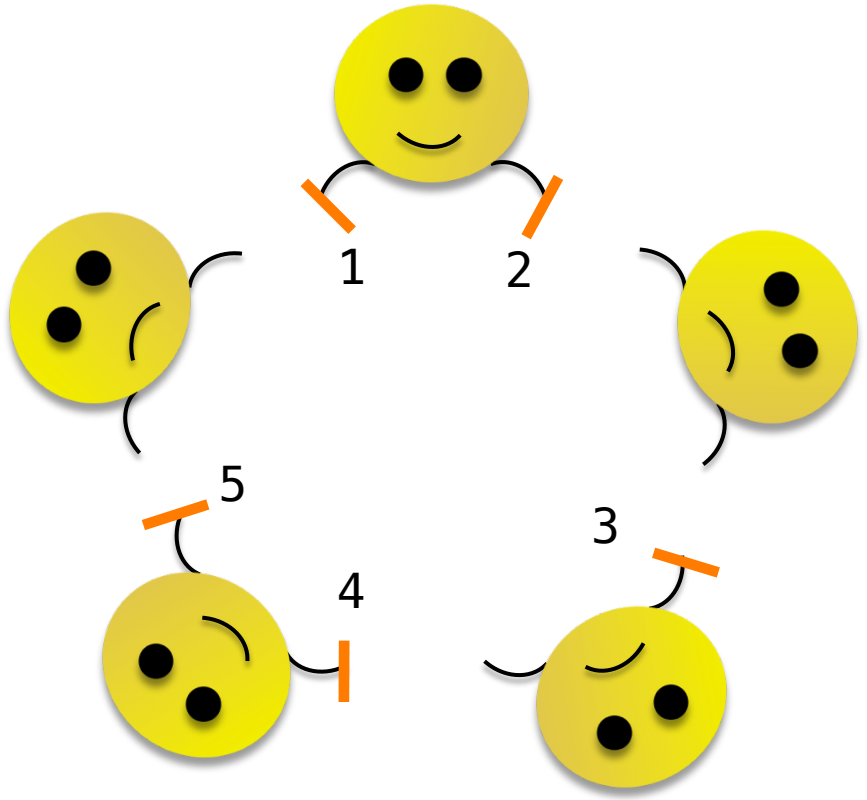
# Allowing Preemption




a philosopher forcefully takes a chopstick from another

- preempt resources from some processes

# Removing Circular Wait




number chopsticks and allow philosophers to pick them in ascending order

• obtain resources according to some global numbering; must release all higher numbered resources if need a lower numbered one later

# References

- Chapter 6, Modern Operating Systems, A. Tanenbaum and H. Bos, 4th Edition.