Operating Systems

Dr. Shu Yin

Part II: Process Management

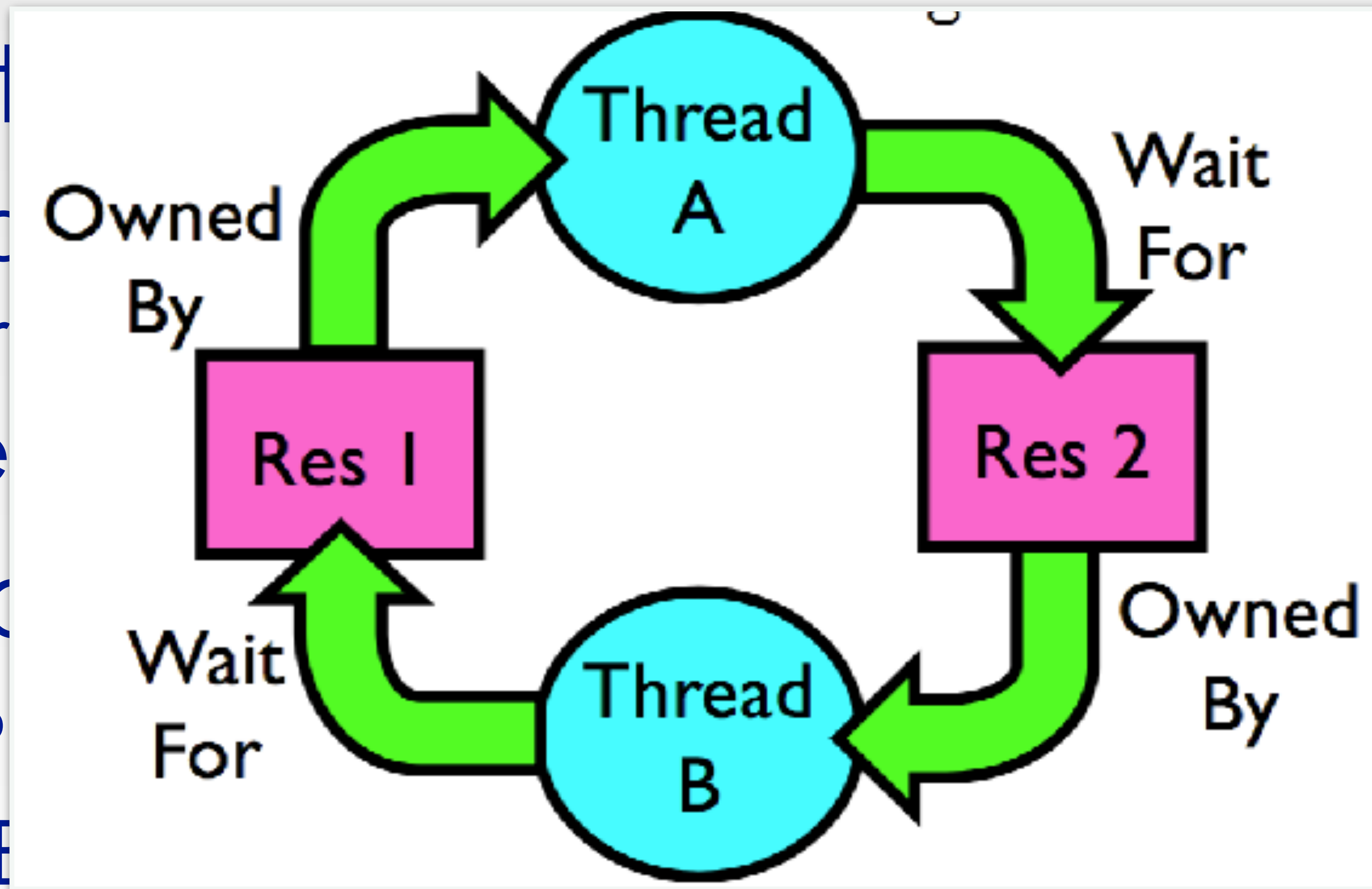
- Processes
- Threads
- Process Synchronization
- CPU Scheduling
- Deadlocks

Starvation vs. Deadlock

- Starvation: process waits indefinitely
  - E.g., low-priority processes waiting for resources constantly in use by high-priority processes
- Deadlock: circular waiting for resources:
  - E.g., Proc.A owns Res.1 and is waiting for Res.2; Proc. B owns Res. 2 and is waiting for Res. 1

Starvation vs. Deadlock

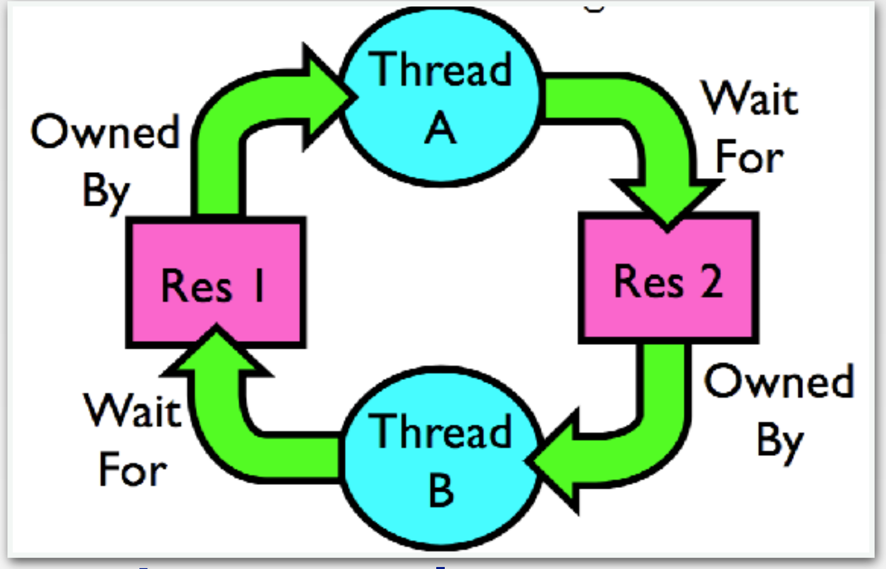
- Starvat ... ely
  - E.g., lo ... resour ... riority proce
- Deadlo ... urces:
  - E.g., P ... for Res.2; Proc. E ... es. 1



ShanghaiTech University 上海科技大学

Starvation vs. Deadlock (cont.)

- Deadlock → Starvation
- NOT vice versa
- Starvation can end
- Deadlock can't end w/o external intervention



上海科技大学
ShanghaiTech University

Goals

- Description of Deadlocks
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

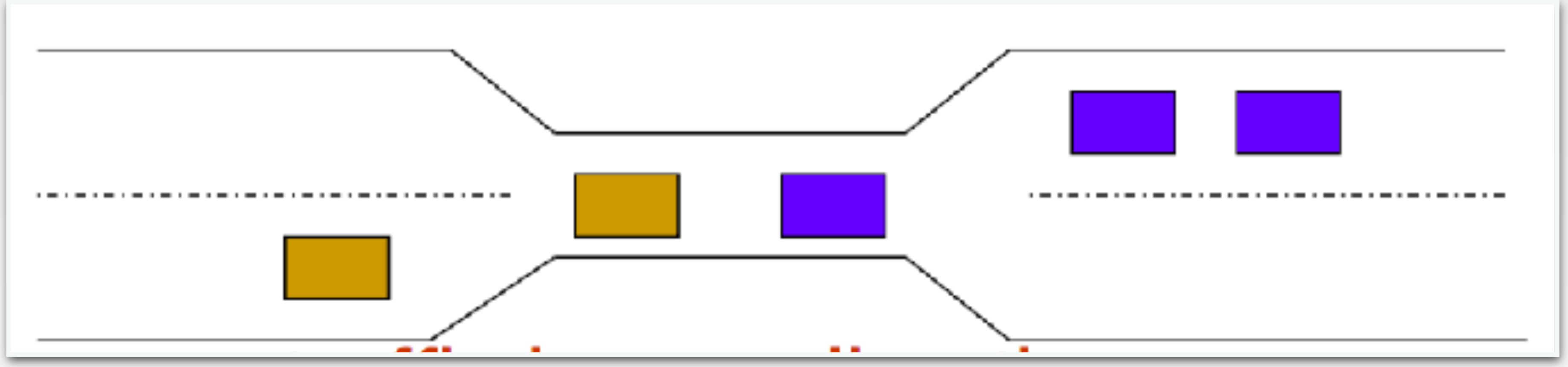
Deadlock Problem

- A Set of blocked processes
- Each holding a resource
- Waiting to acquire a resource held by another process in the set

Deadlock: Definitions

- A process is *Deadlocked*
  - if it is waiting for an event that will never occur
  - Typically, more than one process will be involved in a deadlock (the deadly embrace)
- A process is *indefinitely postponed*
  - if it is delayed repeatedly over a long period of time while the attention of the system is given to other processes

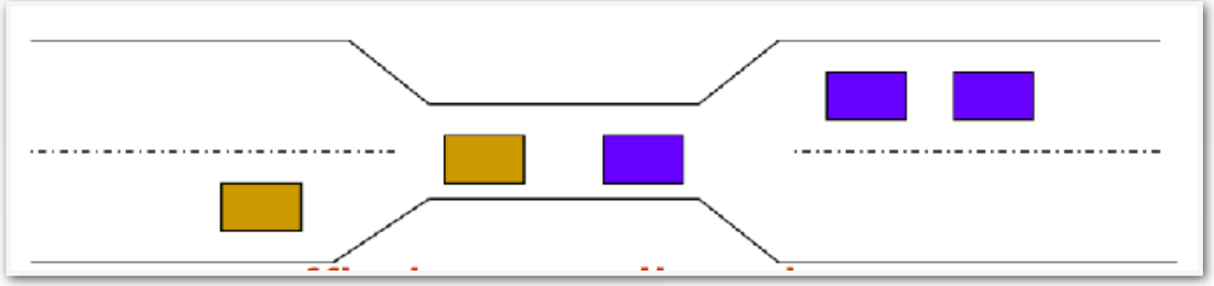
Example: Bridge Crossing



- Assume traffic in one direction
  - Each section of the bridge is viewed as a resource

Example: Bridge Crossing

- If a deadlock occurs, it can be resolved only if one car backs up (preempt resources and rollback)
  - Several cars may have to be hacked up if a deadlock occurs
  - Starvation is possible



Resources

- Commodity required by a process to execute
- Resources can be of several types
  - Serially reusable resources
    - CPU cycles, memory space, I/O devices, files
    - acquire → use → release
  - Consumable resources
    - Produced by a process, needed by a process
      - e.g. message, buffer of information, interrupts
    - create → acquire → use
    - Resource ceases to exist after it has been used

System Model

- Resource types (R_1 … R_m)
- Each resource type R_i has W_i instances
- Assume serially reusable resources
  - request → use → release

上海科技大学 ShanghaiTech University

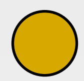
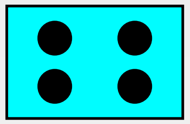
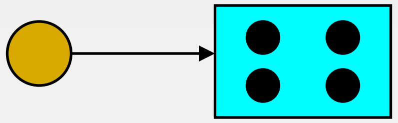
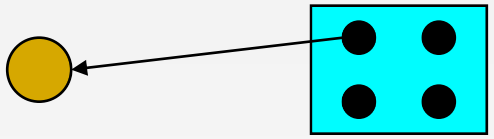
Conditions for Deadlock

- Following condition are necessary and sufficient for deadlock (simultaneously)
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait

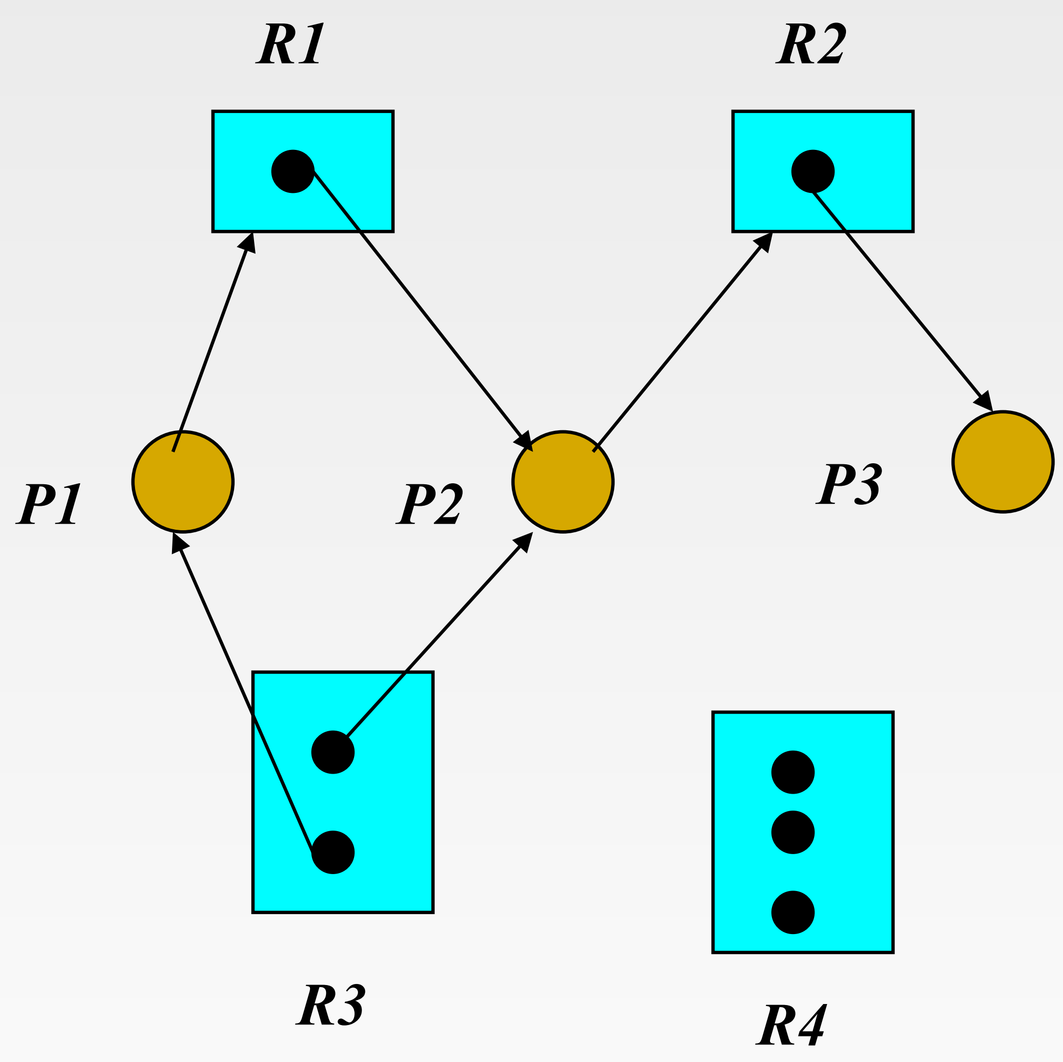
Resource Allocation Graph

- A set of vertices V and a set of edges E
- V is partitioned into 2 types
  - $P=\{P_1,\ldots P_n\}$ the set of processes in the system
  - $R=\{R_1,\ldots R_n\}$ the set of resource types in the system
- Two kinds of edges
  - Request edge - Direct edge $P_i \rightarrow R_j$
  - Assignment edge - Direct edge $R_j \rightarrow P_i$

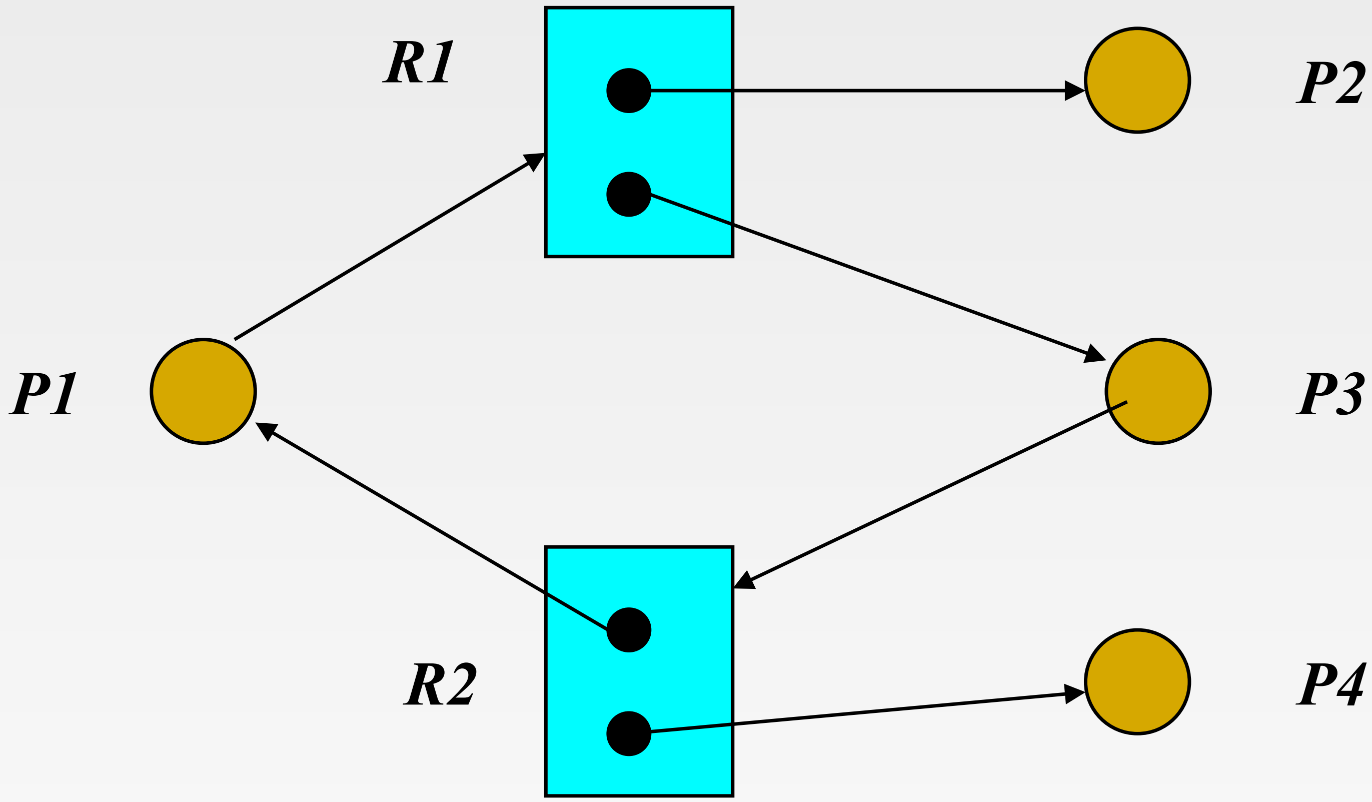
Resource Allocation Graph (cont.)

- Process 
- Resource type with 4 instances 
- P_i requests instance of R_j 
- P_i is holding an instance of R_j 

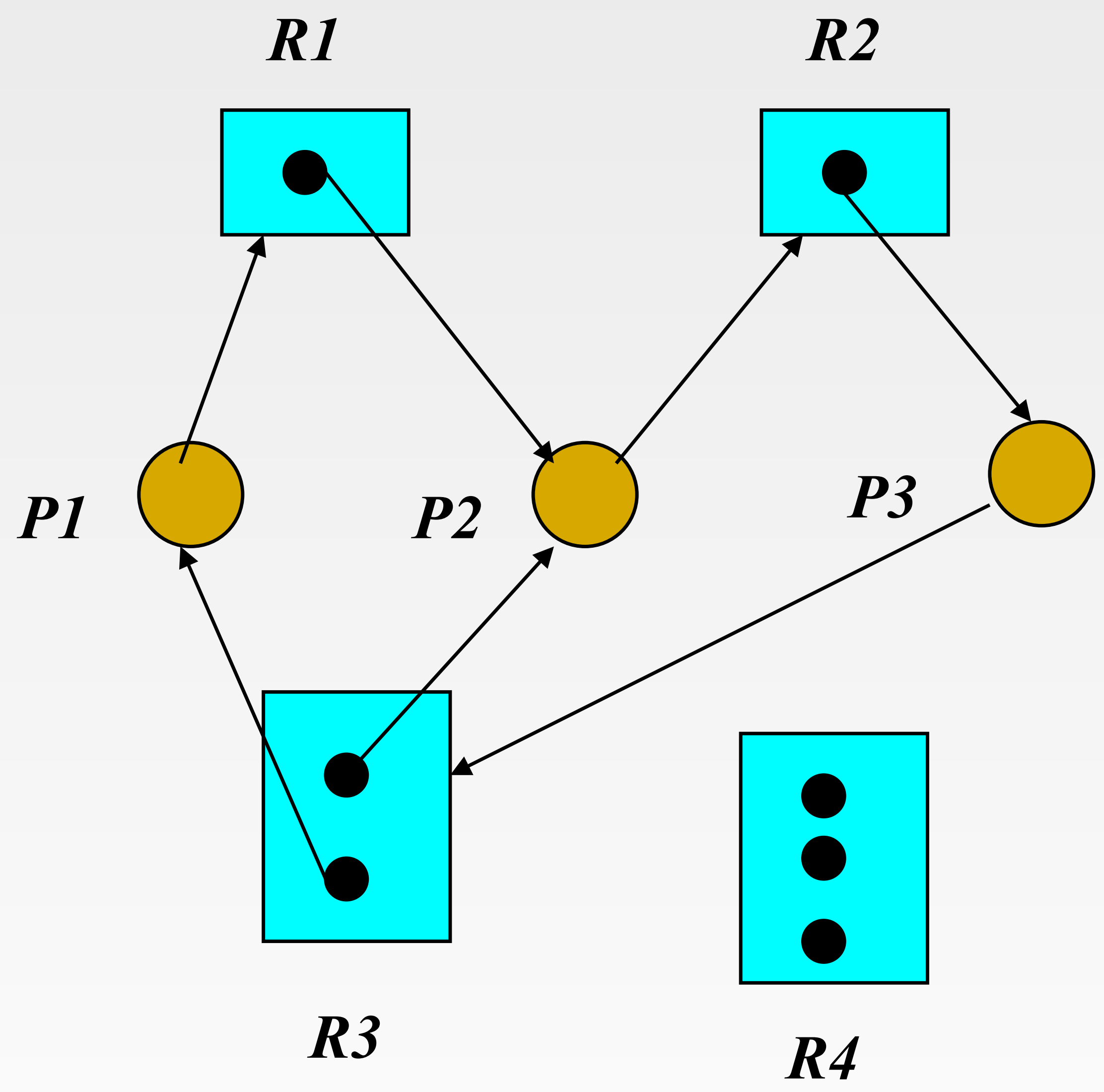
Graph with No Cycles



Graph with Cycles



Graph with Cycles and Deadlocks



Basic Facts

- 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

Methods for Handling Deadlocks

- Ensure never enter a deadlock state
- Allow to enter, detect it and recover
- Ignore, pretend never occur
  - used by many OS, e.g., UNIX

Deadlock Management

- Prevention
- Avoidance
- Detection
- Recovery

Deadlock Prevention

- If any one of the conditions for deadlock (with reusable resources) is denied, deadlock is impossible
- Restrain ways in which requests can be made
  - Mutual exclusion
    - Non-issue for sharable resources
    - Can not deny this for non-sharable resources

Deadlock Prevention (cont.)

- Restrain ways in which requests can be made
  - Mutual exclusion
  - Hold and Wait
    - Guarantee that when a process requests a resource, it does not hold other resources
      - Force each process to acquire all the required resources at once. Process can not proceed until all resources have been acquired
      - Low resource utilization, starvation possible

Deadlock Prevention (cont.)

- No Preemption
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, the process releases the resources currently being held.
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources as well as the new ones that is requesting.

Deadlock Prevention (cont.)

- Circular wait
  - Impose a total ordering of all resource types
  - Require that processes request resources in increasing order of enumeration
  - If a resource of type N is held, process can only request resources of types > N

Deadlock Avoidance

- Set of resources, set of customers, banker
- Rules:
  - Each customer tells banker maximum number of resources it needs
  - Customer borrows resources from banker
  - Customer returns resources to banker
  - Customer eventually pays back loan
- Banker only lends resources if the system will be in a safe state after the loa

Deadlock Avoidance (cont.)

- Requires additional apriori information
  - Simplest and Most useful Mode
    - Maximum number of resources
  - Deadlock-avoidance algorithm
    - Resource-allocation state to ensure that there can never be a circular-wait condition
    - Resource allocation state
      - the number of available and allocated resources,
      - the maximum demands of the processes

Safe State

- When a process requests an available resource
  - If immediate allocation leaves the system in a safe state
- System in safe sate $\rightarrow$ there exists a safe sequence of all processes

Safe State (cont.)

- Sequence <P_1, … P_n> is safe
  - Each P_i, the requested resources satisfied by
    - Currently available resources +
    - Resources held by P_j ($j<i$)
  - If resources by P_j not available, P_i waits until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - P_i terminates, P_{i+1} obtains resources

Resource Allocation Graph Algorithm

- Used for deadlock avoidance when there is only one instance of each resource type
  - Claim edge: $P_i \rightarrow R_j$ indicates P_i may request resource R_j
    - represented by a dashed line (- - - -►)
  - Claim edge converts to request edge when a process requests a resource
  - When a resource is released by a process, assignment edge reconverts to claim edge
  - Resources must be claimed a priori in the system

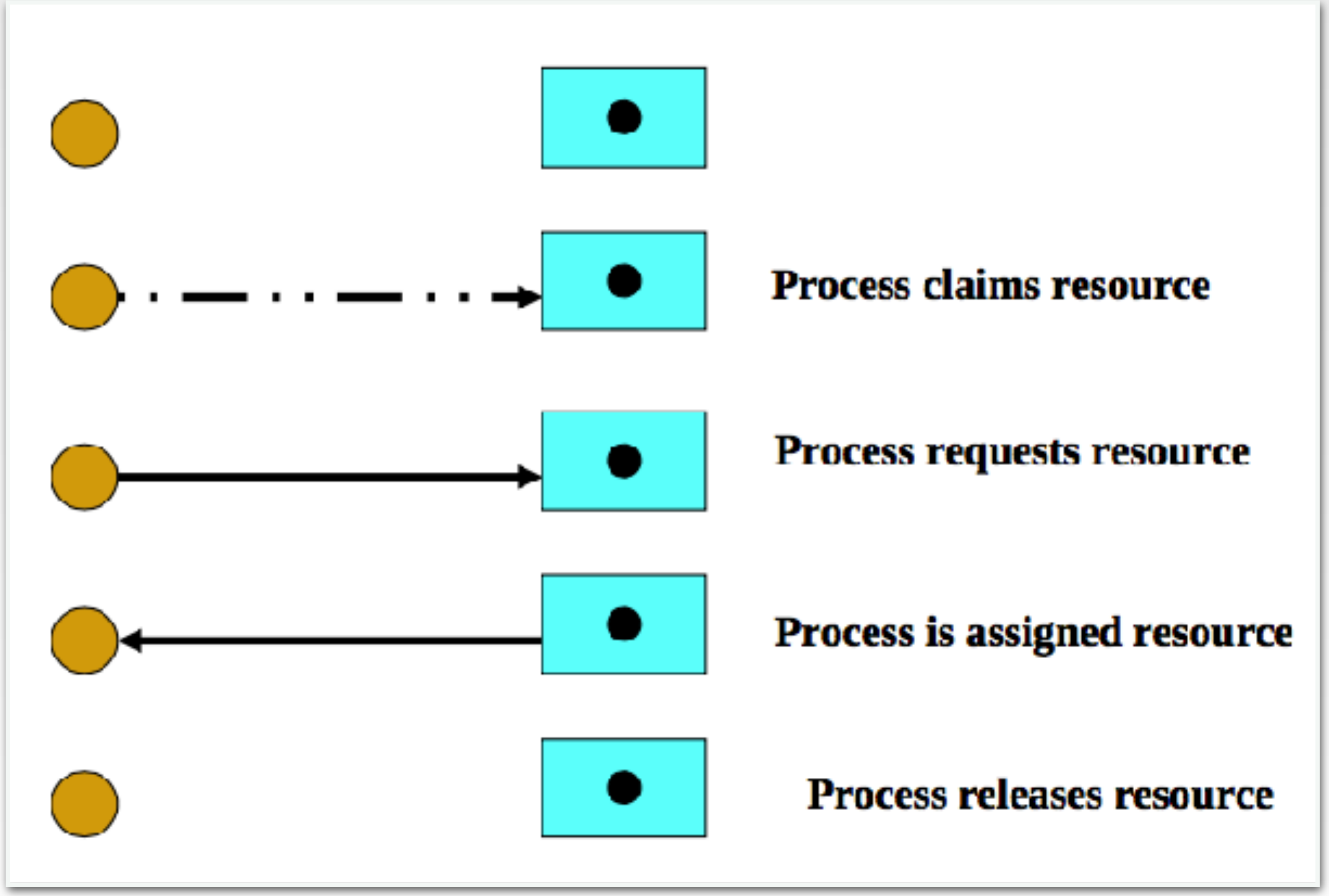
上海科技大学 ShanghaiTech University

Resource Allocation Graph Algorithm

- Used for deadlock avoidance when there is only one instance of each resource type
  - Claim edge: $P_i \rightarrow R_j$ indicates P_i may request resource R_j
    - represented by a dashed line (----►)
  - Claim edge converts to request edge when a process requests a resource
  - When a resource is released by a process, assignment edge reconverts to claim edge
  - Resources must be claimed a priori in the system

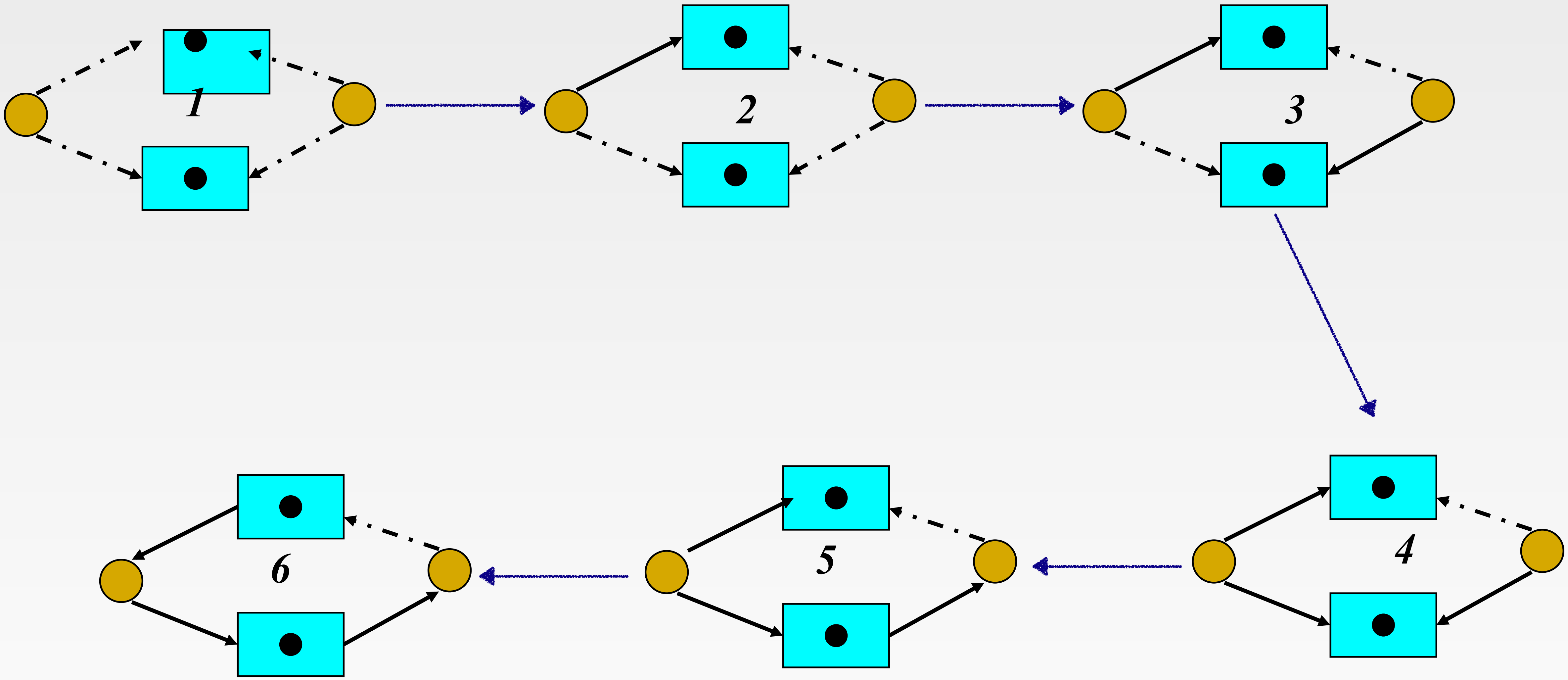
If request assignment does not result in the formation of a cycle in the resource allocation graph - safe state, else unsafe state.

Claim Graph



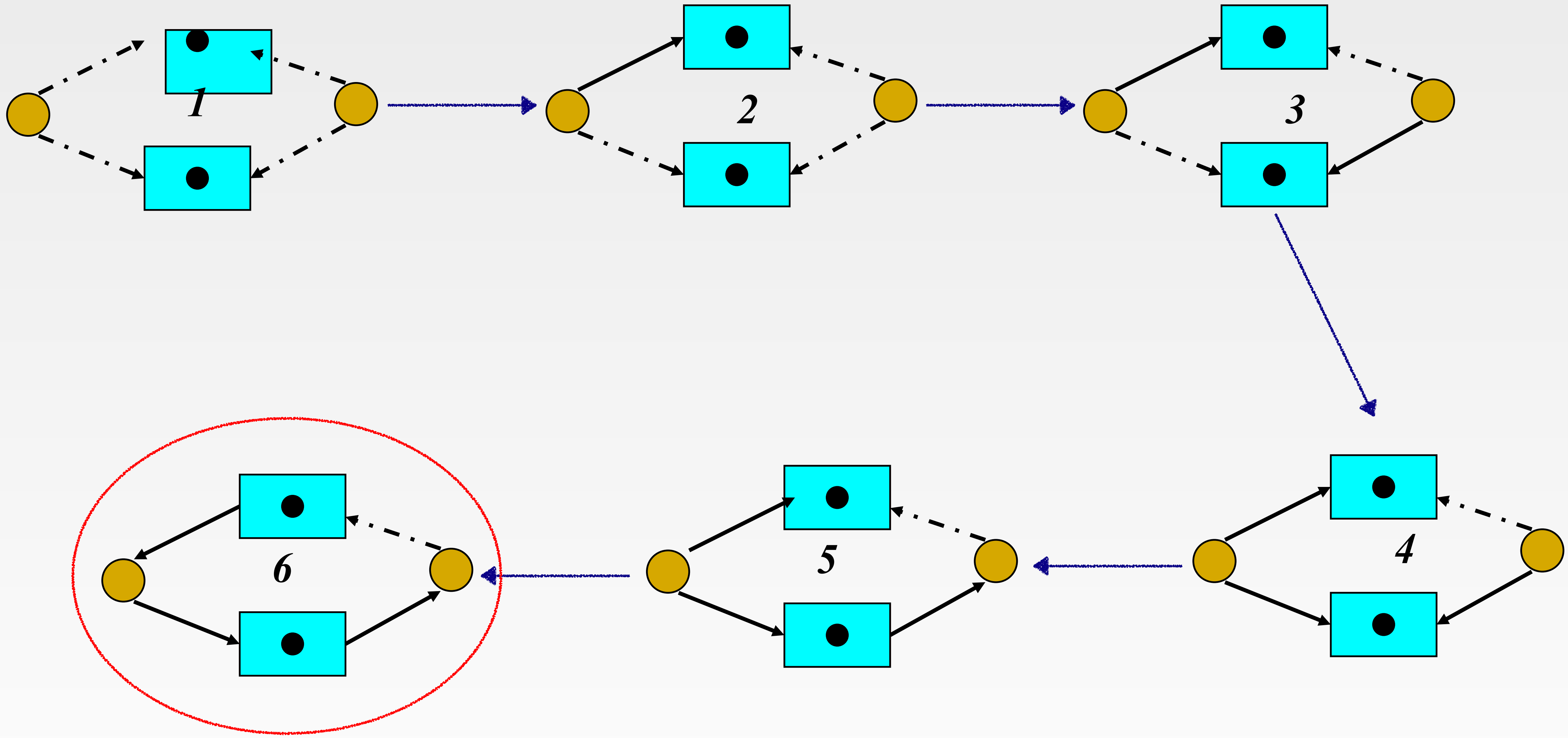
上海科技大学
ShanghaiTech University

Claim Graph



上海科技大学
ShanghaiTech University

Claim Graph



Possible Deadlock

Banker's Algorithm

- Multiple instances of each resource type
- MUST claim maximum use of each resource type
- 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

Data Structures for the Banker's Algorithm

- n: number of processes
- m: number of resource types
  - *Available*: vector of length m
    - Available[j] = k
  - *Max*: matrix (n*m)
    - Max[i,j] = k
  - *Allocation*: matrix (n*m)
    - Allocation[i,j]=k
  - *Need*: matrix (n*m)
    - Need[i,j]=k

Data Structures for the Banker's Algorithm

- n: number of processes
- m: number of resource types
  - *Available*: vector of length m
    - Available[j] = k
  - *Max*: matrix (n*m)
    - Max[i,j] = k
  - *Allocation*: matrix (n*m)
    - Allocation[i,j]=k
  - *Need*: matrix (n*m)
    - Need[i,j]=k

Need[i,j] = Max[i,j] - Allocation[i,j]

Banker's Algorithm - Safety Algorithm

1. Let *Work* and *Finish* :vectors of length m & n
   - Initially, Work = Available
   - Finish[i] = false for i = 0,1,2,…, n-1
2. Find i (P_i) such that both
   - Finish[i] == false
   - $Need_i$ <= Work

   If no such i exists, go to step 4
3. Work = Work+$Allocation_i$

   Finish[i] = true

   go to step 2
4. If Finish[i] == true for all i
   - then the system is in a safe state

Banker's Algorithm - Resource Request Algorithm

$Request_i$ request vector for P_i

$Request_i$ [j] ==k: process P_i wants k instances of resource type R_j

1. If $Request_i$ <= $Need_i$, go to step 2 Otherwise raise Error
2. If $Request_i$ <= $Available_i$, go to step 3 Otherwise Pi must wait
3. Pretend to allocate requested resources to Pi by modifying the state as:
   - Available = Available - $Request_i$
   - $Allocation_i$ = $Allocation_i$ + $Request_i$
   - $Need_i$ = $Need_i$ - $Request_i$

- If safe: resources are allocated to P_i
- If unsafe: P_i wait for $Request_i$ and state is restored

Banker's Algorithm - Resource Request Algorithm

Request ...

$Request_i$ ... type R_j

1. If Requ... ror
2. If Requ... ust wait
3. Preten... modifyi...

```
[Avail] = [FreeResources]
      Add all nodes to UNFINISHED
      do {
            done = true
         Foreach node in UNFINISHED {
            if ([Request_node] <= [Avail]) {
               remove node from UNFINISHED
               [Avail] = [Avail] + [Alloc_node]
               done = false
            }
         }
      } until(done)
```

   - $Available = Available - Request_i$
   - $Allocation_i = Allocation_i + Request_i$
   - $Need_i = Need_i - Request_i$

- If safe: resources are allocated to P_i
- If unsafe: P_i wait for $Request_i$ and state is restored

上海科技大学 ShanghaiTech University

Example: Banker's Algorithm

- 5 processes: P_0-P_4
- 3 resource types: A (10 instances), B (5 instance), C (7 instances)
- Snapshot at time T_0

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

Example (cont.)

- The content of the matrix *Need* is defined by *(Max-Allocation)*
- The system is in a safe state since the sequence <P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

	Need		
	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Example: P_1 requests (1,0,2)

- Check to see that $Request_1$ <= Available
  - ((1,0,2) <= (3,3,2)) $\rightarrow$ TRUE

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	2	3	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

Example (cont.)

- Executing the safety algorithm shows that sequence <P_1, P_3, P_4, P_2, P_0> satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

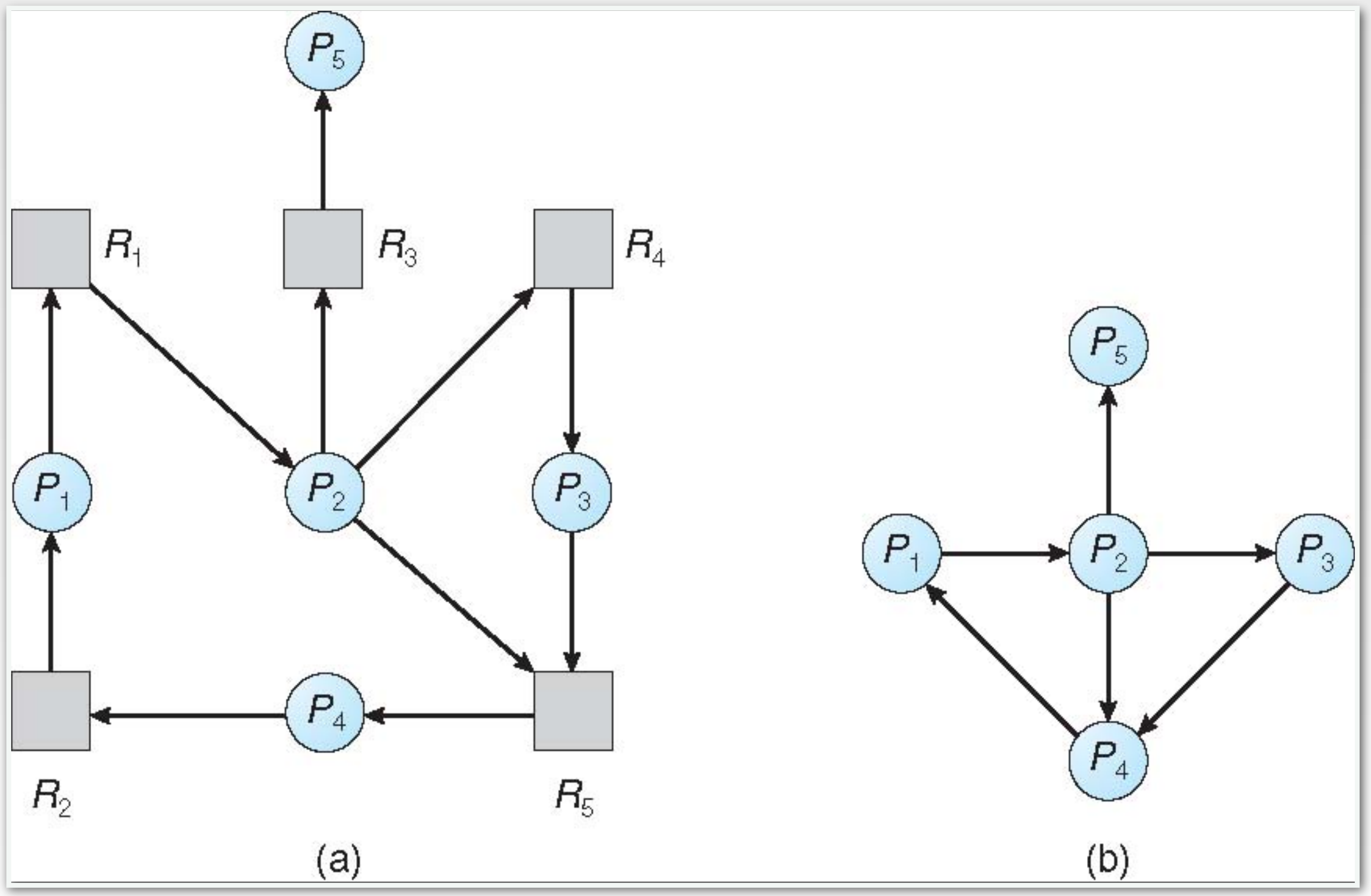
上海科技大学 ShanghaiTech University

Deadlock Detection

- Allow system to enter deadlock state
- Detection Algorithm
- Recovery Scheme

Single Instance of Each Resource Type

- Maintain wait-for graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph
- An algorithm to detect a cycle in a graph requires an order of n^2 operations
  - n: number of vertices in the graph


P5
R1
R3
R4
P1
P2
P3
R2
P4
R5
(a)
P5
P1
P2
P3
P4
(b)

Several Instances of A Resource Type

- Data Structures
  - *Available*: Vector of length *m*. If *Available*[*j*] = *k*, there are *k* instances of resource type *Rj* available.
  - *Allocation*: $n \times m$ matrix. If *Allocation*[*i,j*] = *k*, then process *Pi* is currently allocated *k* instances of resource type *Rj*.
  - *Request* : An $n \times m$ matrix indicates the current request of each process. If *Request* [*i,j*] = *k*, then process *Pi* is requesting *k* more instances of resource type *Rj* .

Deadlock Detection Algorithm

- Step 1: Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize
  - *Work* := *Available*
  - For *i* = 1,2,…,*n*, if *Allocation*(*i*) ≠ 0, then *Finish*[*i*] := *false*, otherwise *Finish*[*i*] := *true*.

Step 2: Find an index *i* such that both:

- *Finish*[*i*] = *false*
- *Request* (*i*) ≤ *Work*
- If no such *i* exists, go to step 4.

Deadlock Detection Algorithm (cont.)

- Step 3: *Work* := *Work* + Allocation(*i*)
  - *Finish*[*i*] := *true*
  - go to step 2
- Step 4: If *Finish*[*i*] = *false* for some *i*, $1 \leq i \leq n$, then the system is in a deadlock state. Moreover, if *Finish*[*i*] = *false*, then *Pi* is deadlocked.
- Algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlock stack

上海科技大学 ShanghaiTech University

Example of Detection Algorithm

- 5 processes: P_0-P_4
- 3 resource types: A (7 instances), B (2 instance), C (6 instances)
- Snapshot at time T_0 <P_0, P_2, P_3, P_1, P_4> will result in Finish[i] = true for all i

Example of Detection Algorithm

- 5 processes: P_0-P_4
- 3 resource types: A (7 instances), B (2 instance), C (6 instances)
- Snapshot at time T_0 <P
result in Finish[i] = true

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

Example (cont.)

- P_2 requests an additional instance of type C
- State of system
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes' requests
  - Deadlock exists, consisting of P_0, P_2, P_3, P_1, and P_4

上海科技大学
ShanghaiTech University

Example (cont.)

- P_2 requests an additional instance of type C
- State of system
  - Can reclaim resources held by pr insufficient resources to fulfill othe requests
  - Deadlock exists, consisting of P_0, I and P_4

	Request		
	A	B	C
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	0	0	2

上海科技大学
ShanghaiTech University

Detection-Algorithm Usage

- When and how often to invoke depend on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - One for each disjoint cycle

Detection-Algorithm Use (cont.)

- How often:
  - Every time a request for allocation cannot be granted immediately
    - Allows us to detect set of deadlocked processes and process that “caused” deadlock. Extra overhead
    - Every hour or whenever CPU utilization drops
  - With arbitrary invocation there may be many cycles in the resource graph and we would not be to tell which of the many deadlocked processes “caused” the deadlocked

Recovery from Deadlock

- Process Termination
  - Abort all deadlocked processes
  - Abort one process at a time, until eliminated
  - Abort order options
    - Priority of the process
    - Computing time (has computed, to complete)
    - Resources used
    - Resources needed to complete
    - Resources will be needed to be terminated
    - Is process interactive or batch?

Recovery from Deadlock (cont.)

- Resource Preemption
  - Selecting a victim- minimize cost
  - Rollback
    - Return to some safe state
    - Restart process from that state
  - Starvation
    - Same process may always be picked as a victim
    - Include number of rollback in cost factor

上海科技大学 ShanghaiTech University

Combined approach to deadlock handling

- Combination of 3 basic approaches
  - Prevention
  - Avoidance
  - Detection
  - Allowing the use of the optimal approach for each class of resources in the system

Combined approach (cont.)

- Partition resources into hierarchically ordered classes
  - Use more appropriate technique for handling deadlocks within each class

上海科技大学 ShanghaiTech University

Summary

- Starvation vs. Deadlock
  - process waits indefinitely
  - circular waiting for resources
- Four conditions for deadlocks
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait

Summary (cont.)

- Techniques for addressing Deadlock
  - Allow system to enter deadlock then recover
  - Ensure that system never enter deadlock
  - Ignore the problem and pretend deadlock never occur in system