Operating Systems and Systems Programming

Deadlock

The Content is prepared with the help of existing text books mentioned below:

References:

1. Silberschatz, Abraham, Peter B. Galvin, and Greg Gagne. *Operating system concepts with Java*. Wiley Publishing, 2009.
2. Stallings, William. *Operating Systems 5th Edition*. Pearson Education India, 2006.
3. Tannenbaum, Andrew S. "Modern Operating Systems, 2009."

Deadlock

Resources

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Resources

- Preemptable resources
  - can be taken away from a process with no ill effects
- Non-preemptable resources
  - will cause the process to fail if taken away

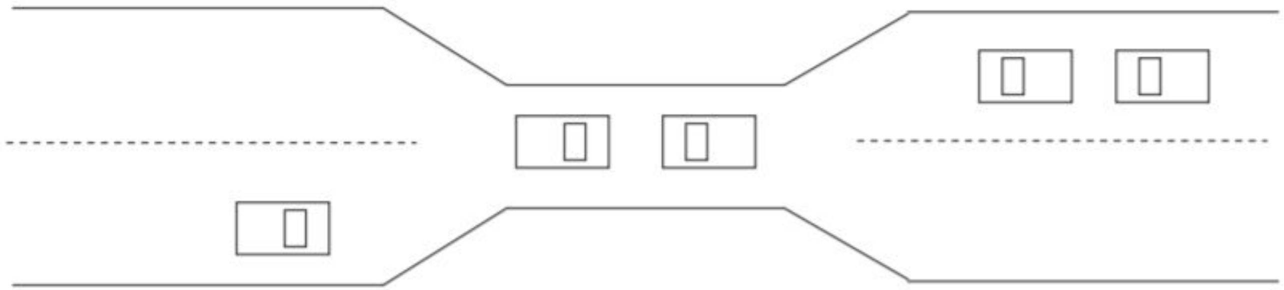
Resources

- Sequence of events required to use a resource
  - request the resource
  - use the resource
  - release the resource

- Must wait if request is denied
  - requesting process may be blocked
  - may fail with error code

DEADLOCKS

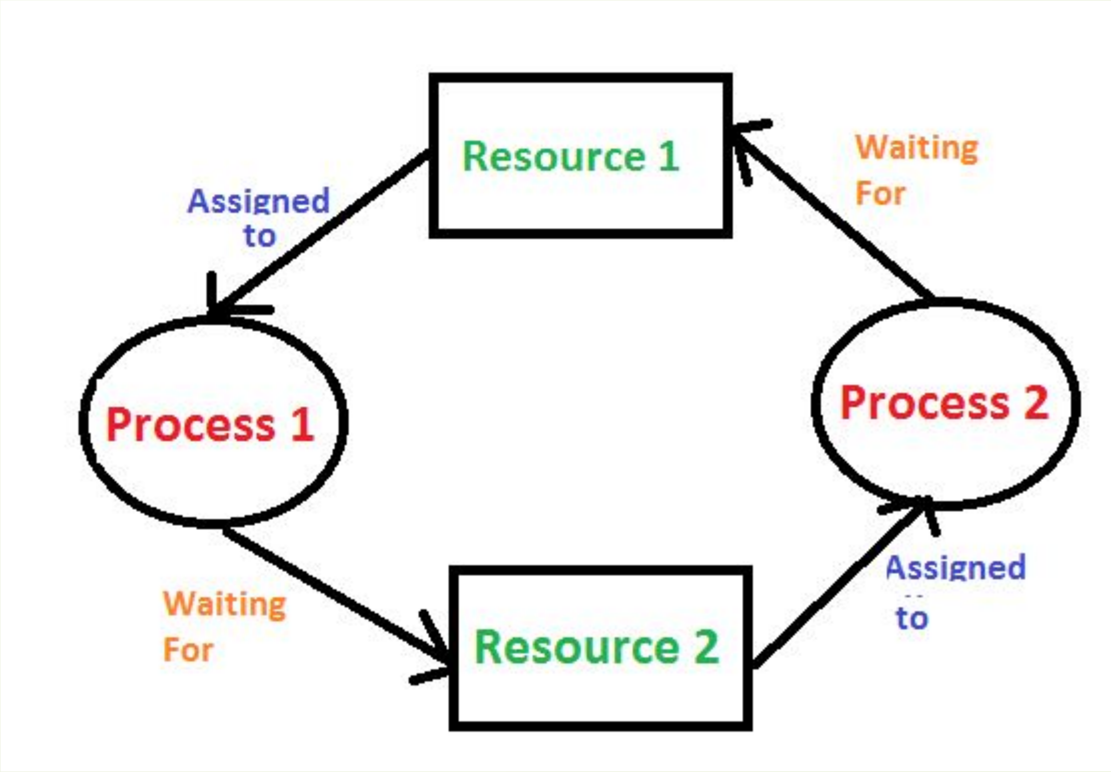
Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

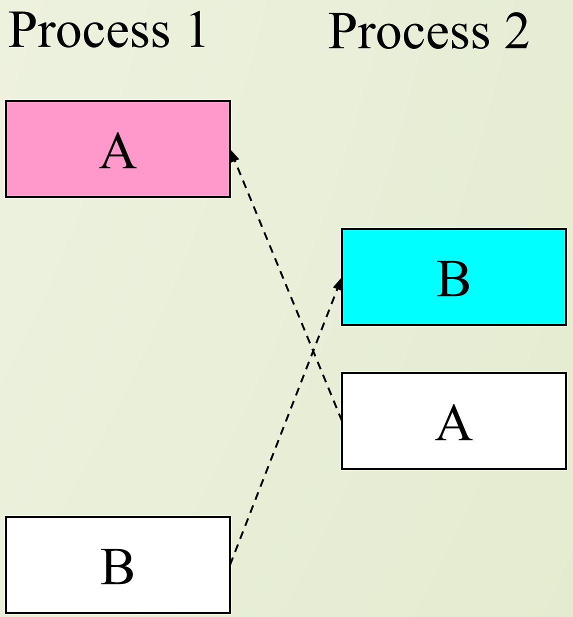
The Deadlock problem

In a computer system deadlocks arise when members of a group of processes which hold resources are blocked indefinitely from access to resources held by other processes within the group.



When do deadlocks happen?

- Suppose
  - Process 1 holds resource A and requests resource B
  - Process 2 holds B and requests A
  - Both can be blocked, with neither able to proceed
- Deadlocks occur when …
  - Processes are granted exclusive access to devices or software constructs (resources)
  - Each deadlocked process needs a resource held by another deadlocked process



DEADLOCK!

Reference: CS 1550, cs.pitt.edu

What is a deadlock?

- Formal definition:
  “A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.”
- Usually, the event is release of a currently held resource
- In deadlock, none of the processes can
  - Run
  - Release resources
  - Be awakened

Deadlock with Semaphores

- Data:
  - A semaphore **S1** initialized to 1
  - A semaphore **S2** initialized to 1
- Two processes P1 and P2
- **P1:**

  wait(s1)

  wait(s2)
- **P2:**

  wait(s2)

  wait(s1)

Four Conditions for Deadlock

Mutual exclusion condition

- each resource assigned to 1 process or is available

Hold and wait condition

- process holding resources can request additional

No preemption condition

- previously granted resources cannot forcibly taken away

Circular wait condition

- must be a circular chain of 2 or more processes
- each is waiting for resource held by next member of the chain

Four Conditions for Deadlock

Mutual exclusion condition

- each resource assigned to 1 process or is available

Hold and wait condition

- process holding resources can request additional

No preemption condition

- previously granted resources cannot forcibly taken away

Circular wait condition

- must be a circular chain of 2 or more processes
- each is waiting for resource held by next member of the chain

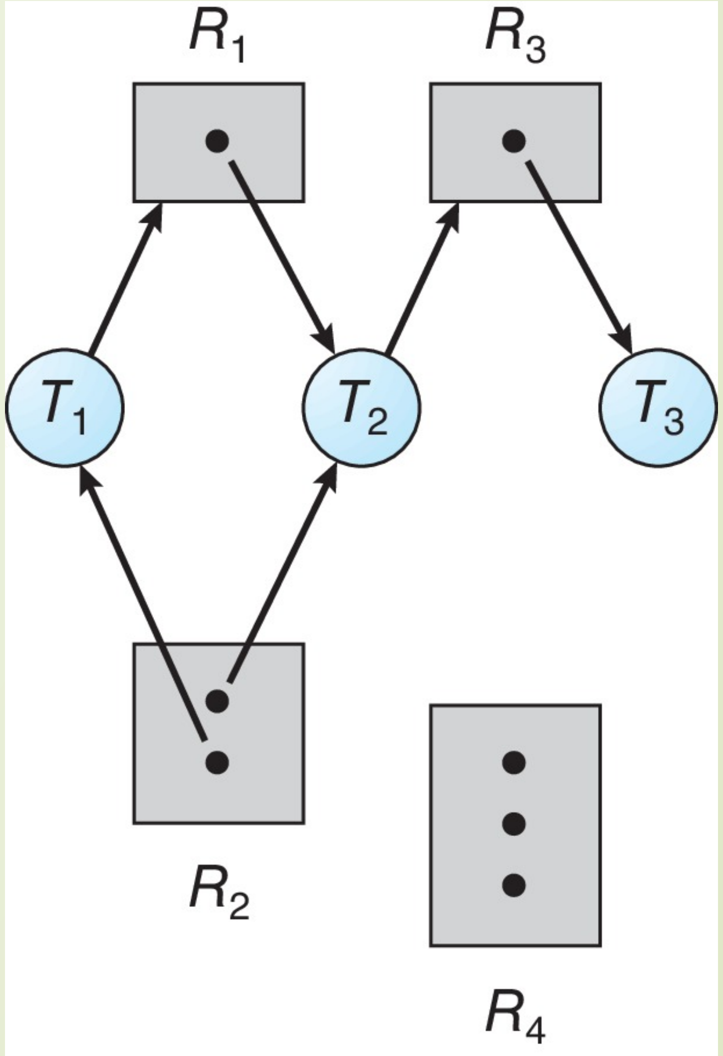
Resource-Allocation Graph

A set of vertices V and a set of edges E.

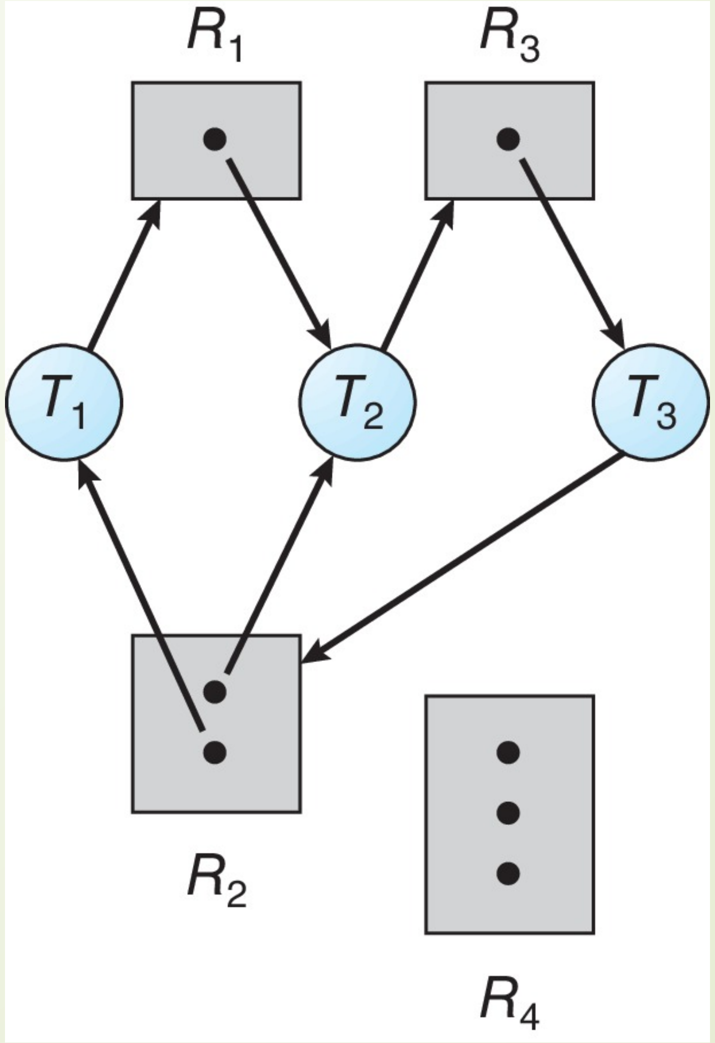
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource Allocation Graph Example

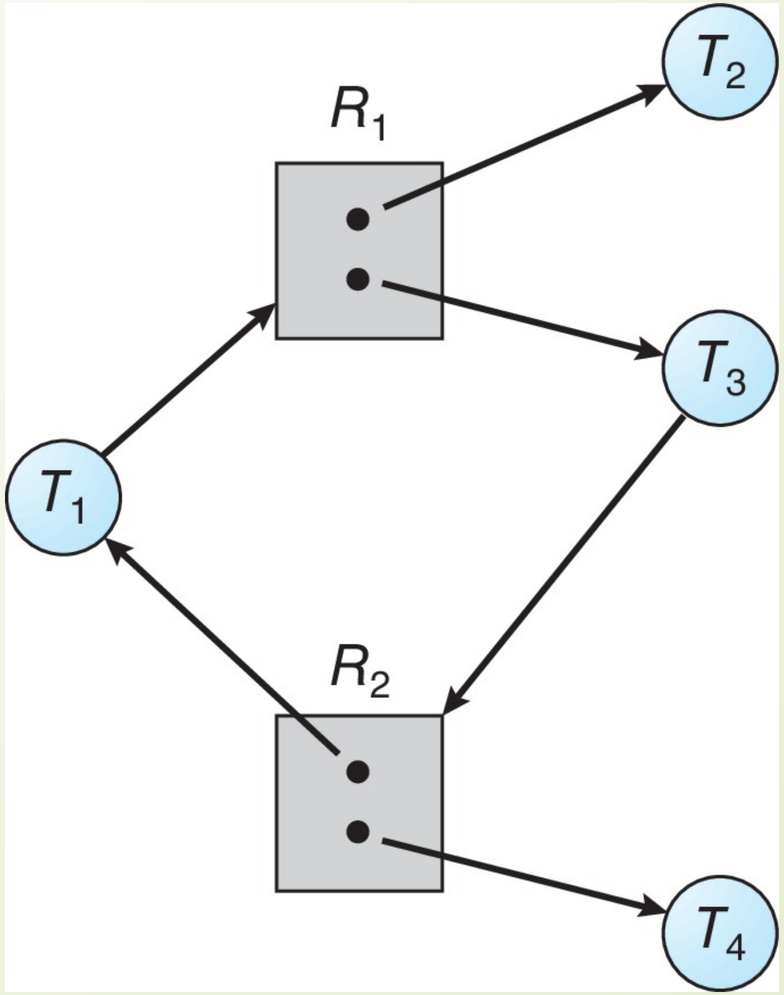
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3



Resource Allocation Graph with a Deadlock



Graph with a Cycle But no Deadlock



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

Dealing with Deadlock

- Three general approaches exist for dealing with deadlock:

Prevent Deadlock

- adopt a policy that eliminates one of the conditions

Avoid Deadlock

- make the appropriate dynamic choices based on the current state of resource allocation

Detect Recovery

- attempt to detect the presence of deadlock and take action to recover

Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

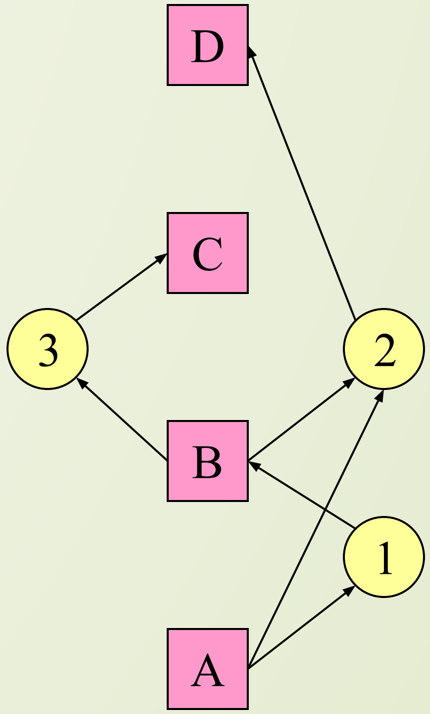
Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- **No Preemption:**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - 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 it is requesting
- **Circular Wait:**
  - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Attacking "circular wait"

- Assign an order to resources
- Always acquire resources in numerical order
  - Need not acquire them all at once!
- Circular wait is prevented
  - A process holding resource *n* can't wait for resource *m* if $m < n$
  - No way to complete a cycle
    - Place processes above the highest resource they hold and below any they're requesting
    - All arrows point up!



Reference: CS 1550, cs.pitt.edu

Deadlock Avoidance

Requires that the system has some additional a priori information available

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

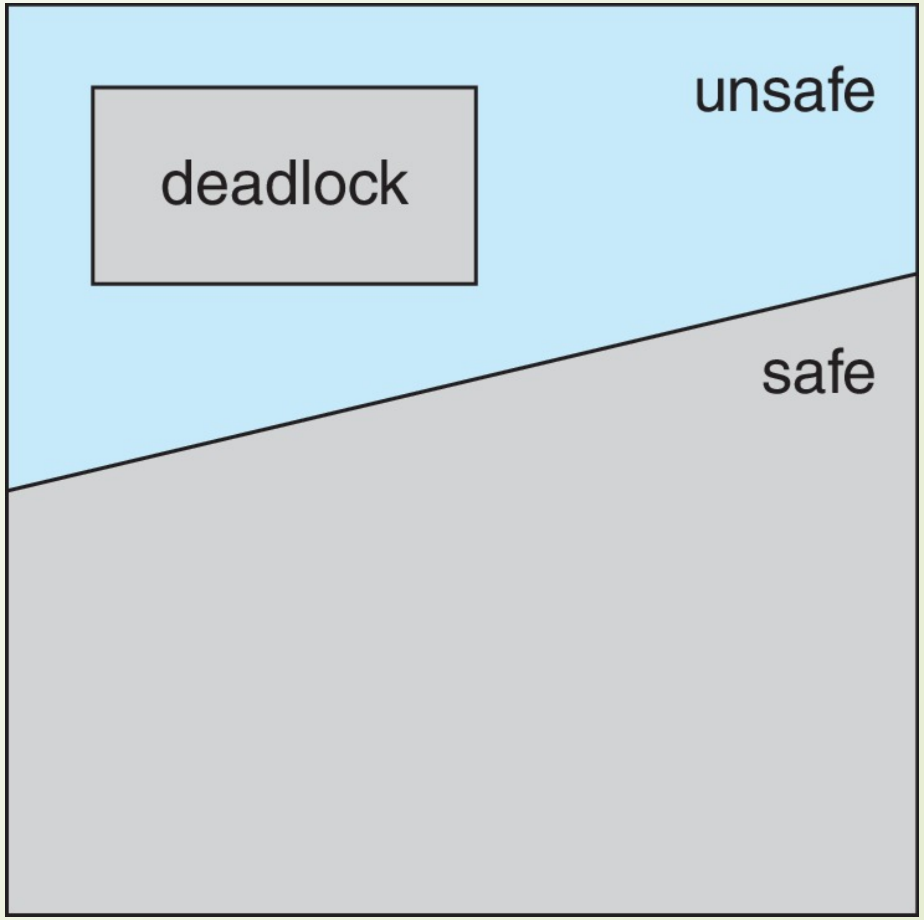
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $<P_1, P_2, ..., P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

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

Safe, Unsafe, Deadlock State



Avoidance Algorithms

- Resource-allocation graph

- Banker’s Algorithm

Banker's Algorithm

- Multiple instances of resources
- Each process must a priori claim maximum use
- 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

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available
- **Max**: $n \times m$ matrix. If $Max\ [i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length m and n, respectively. Initialize:

   $$Work = Available$$

   $$Finish[i] = false \text{ for } i = 0, 1, \ldots, n-1$$

2. Find an i such that both:

   (a) $Finish[i] = false$

   (b) $Need_i \leq Work$

   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

Resource-Request Algorithm for Process P_i

$Request_i$ = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to P_i
- If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), B (5 instances), and C (7 instances)

- Snapshot at time T_0:

	<u>*Allocation*</u>	<u>*Max*</u>	<u>*Available*</u>
	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	

Example (Cont.)

- The content of the matrix *Need* is defined to be *Max – Allocation*

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

Example (Cont.)

- Snapshot at time T_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	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

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, $(1,0,2) \le (3,3,2) \Rightarrow$ true

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2>$ 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?

Example: consider a system with 5 processes (P_0 ... P_4) and 3 resources types (A(10) B(5) C(7))

resource-allocation state at time t_0:

Is the system in a safe state? If so, which sequence satisfies the safety criteria?

< P_1, P_3, P_4, P_2, P_0 >

Now suppose, P_1 requests an additional instance of A and 2 more instances of type C.

request[1] = (1,0,2)

1. check if request[1] <= need[i] (yes)
2. check if request[1] <= available[i] (yes)
3. do pretend updates to the state

Is the system in a safe state? If so, which sequence satisfies the safety criteria?

<P_1, P_3, P_4, P_0, P_2> **(The state matrix on the back slide)**

Hence, we immediately grant the request.

Will a request of (3,3,0) by P_4 be granted?

Will a request of (0,2,0) by P_0 be granted?

Process	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3	3	3	2
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			

Q1. Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) Initial state

Determination of a Safe State contd

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) **P2 runs to completion**

Determination of a Safe State Contd..

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) **P1 runs to completion**

Determination of a Safe State Contd..

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

(d) P3 runs to completion

Determination of an Unsafe State Contd..(Solve)

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) **Initial state**

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(b) **P1 requests one unit each of R1 and R3**

A computer system uses the Banker's Algorithm to deal with deadlocks. Its current state is shown in the tables below, where P0, P1, P2 are processes and R0, R1, R2 are resource types.

	Maximum Need				Current Allocation			Available		
	R0	R1	R2		R0	R1	R2	R0	R1	R2
P0	4	1	2	P0	1	0	2	2	2	0
P1	1	5	1	P1	0	3	1			
P2	1	2	3	P2	1	0	2			

(a) Show that the system can be in this state.

(b) What will the system do on a request by process P0 for one unit of resource type R1?

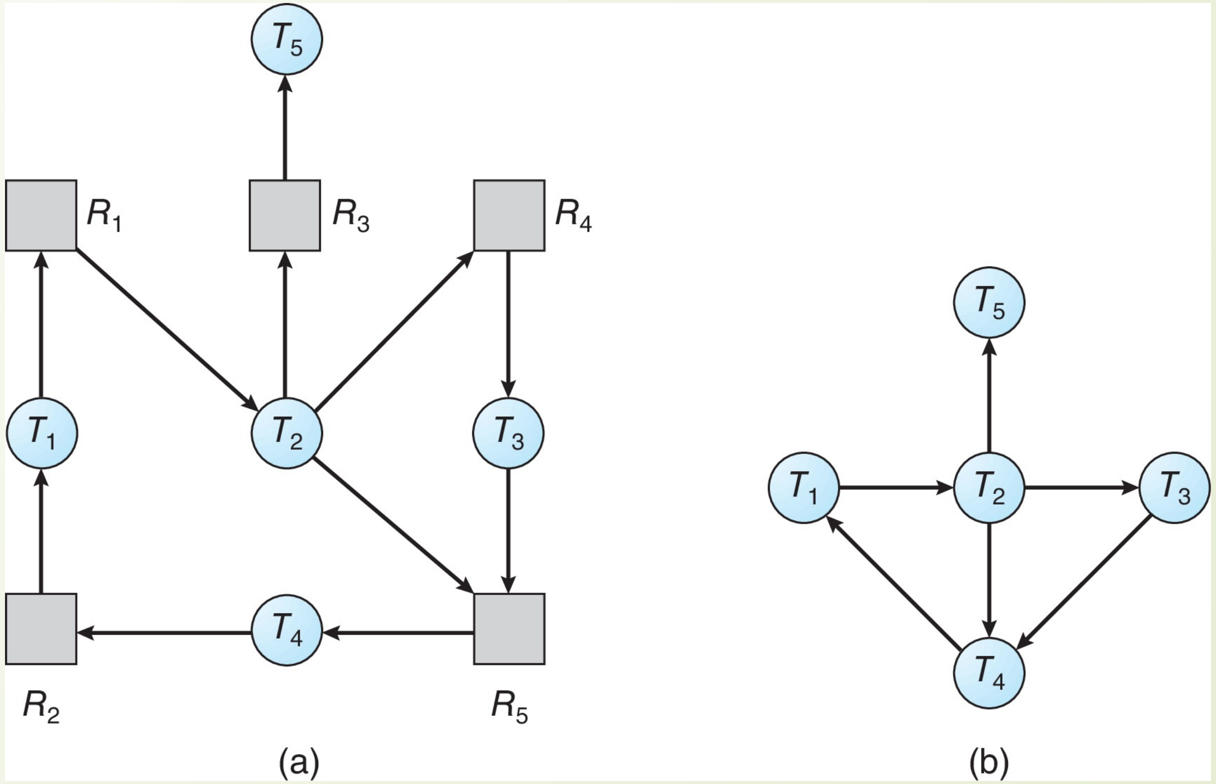
Deadlock Detection

- 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. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph

Several Instances of a Resource Type

- **Available**: A vector of length ***m*** indicates the number of available resources of each type
- **Allocation**: An ***n*** **x** ***m*** matrix defines the number of resources of each type currently allocated to each process
- **Request**: An ***n*** **x** ***m*** matrix indicates the current request of each process.
  - If ***Request*** **[*i*][*j*]** = ***k***, then process P_i is requesting ***k*** more instances of resource type R_j.

Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:

   a) ***Work = Available***

   b) For $\boldsymbol{i = 1,2, \ldots, n}$, if $\boldsymbol{Allocation_i \neq 0}$, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

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

   a) ***Finish*[*i*] == *false***

   b) $\boldsymbol{Request_i \leq Work}$

   If no such ***i*** exists, go to step 4

Detection Algorithm (Cont.)

3. ***Work = Work + Allocation**$_i$*
   Finish*[*i*] = *true
   go to step 2

4. If ***Finish[i]* == *false***, for some ***i***, $1 \leq \boldsymbol{i} \leq \boldsymbol{n}$, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then $\boldsymbol{P_i}$ is deadlocked

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
- Snapshot at time T_0:

	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	

- Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in ***Finish[i] = true*** for all ***i***

Example (Cont.)

- P_2 requests an additional instance of type C

	Request
	$A\ B\ C$
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

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

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Question: Deadlock Detection (Solve)

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

Advantages and Disadvantages

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

Thank you!!!!

References:

1. Silberschatz, Abraham, Peter B. Galvin, and Greg Gagne. *Operating system concepts with Java*. Wiley Publishing, 2009.
2. Stallings, William. *Operating Systems 5th Edition*. Pearson Education India, 2006.
3. Tannenbaum, Andrew S. "Modern Operating Systems, 2009."