Operating System

Deadlocks

CHAPTER OBJECTIVES

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks.
- To present a number of different methods for preventing or avoiding deadlocks

Why is deadlock handling important?

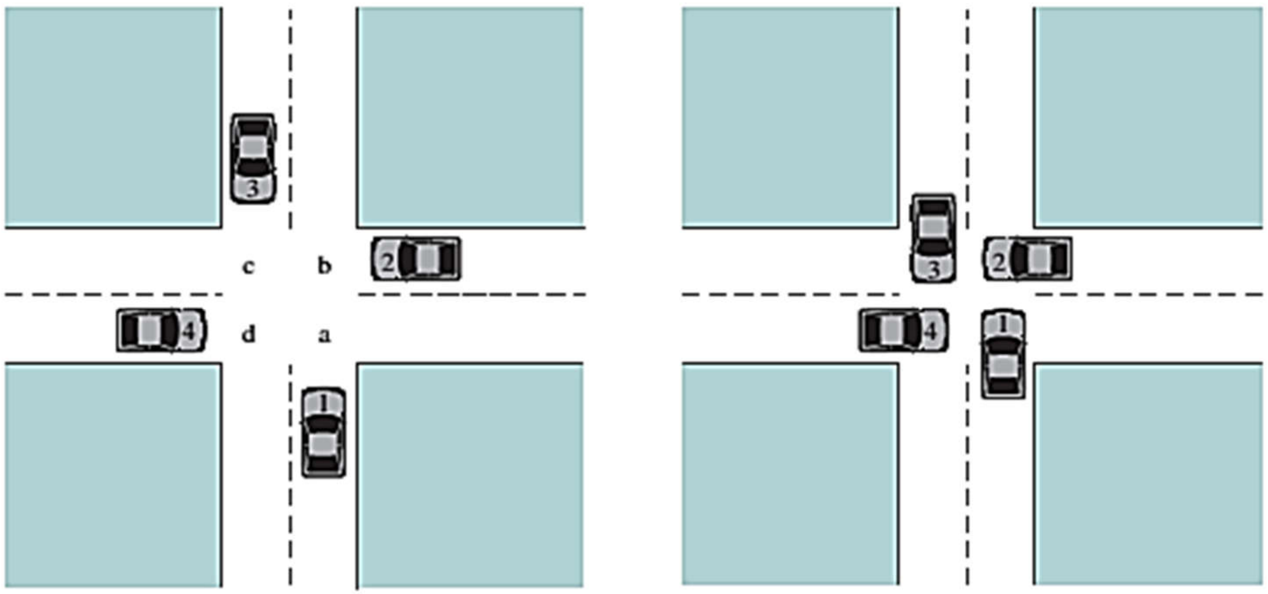
- Deadlocks prevent sets of concurrent processes from completing their tasks

Process P	**Process Q**
• • •	• • •
Get A	Get B
• • •	• • •
Get B	Get A
• • •	• • •
Release A	Release B
• • •	• • •
Release B	Release A
• • •	• • •

Deadlock

- Definition
  - To wait for a resource which is to acquired by another process that is waited for a resource of requesting process
  - Never finishing wait state
  - Circular dependencies between processes

Illustration of deadlock

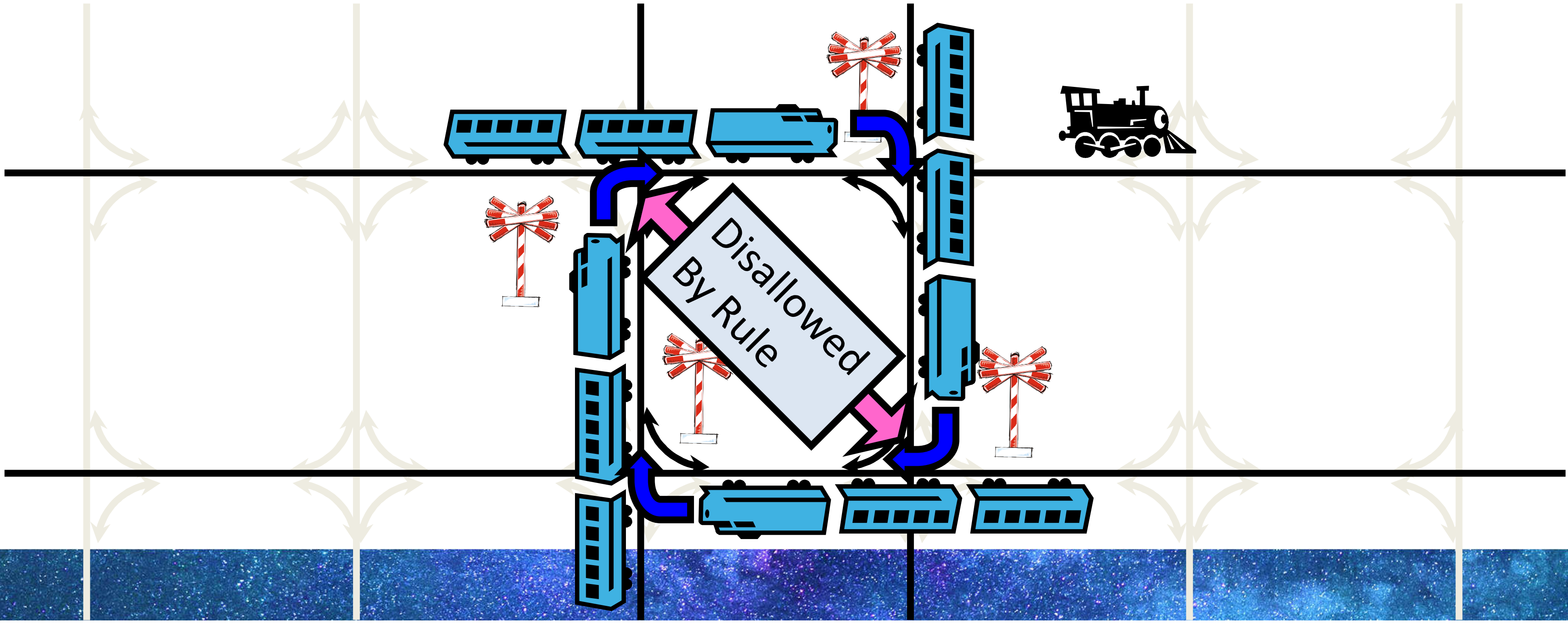


(a) Deadlock possible

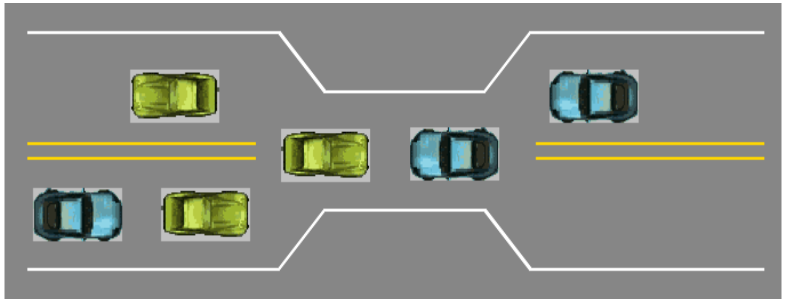
(b) Deadlock

Train Example (Wormhole-Routed Network)

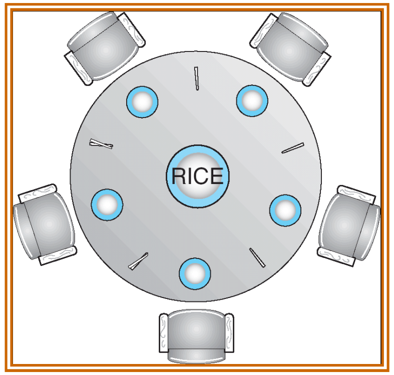
- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)



Deadlock



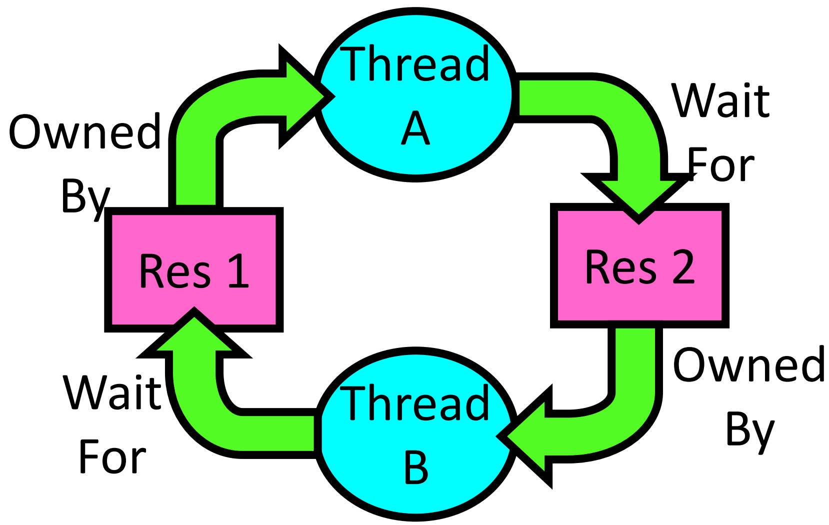
Dining Philosophers Problem

- Five chopsticks/Five philosophers
  – Free-for all: philosopher will grab any one they can
  – Need two chopsticks to eat
- What if all grab at same time?
  – Deadlock!
- How to fix deadlock?
  – Make one of them give up a chopstick
  – Eventually everyone will get chance to eat
- How to prevent deadlock?
  – Never let philosopher take last chopstick if no hungry philosopher has two chopsticks afterwards

Starvation vs Deadlock

- Starvation: thread waits indefinitely
  - Example, low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock: circular waiting for resources
  - Thread A owns Res 1 and is waiting for Res 2
    Thread B owns Res 2 and is waiting for Res 1



- Deadlock $\Rightarrow$ Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention

Necessary conditions

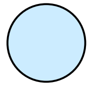
- Mutual exclusion
  - only one process at a time can use a resource
- Hold and wait
  - a process holding at least one resource is waiting to acquire additional resources held by other processes
- No preemption
  - a resource can be released only voluntarily by the process holding it, after that process has completed its task
- Circular wait
  - there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

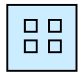
How to model deadlock?

- 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**

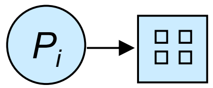
Resource-allocation graph

- Directed graph
  - Nodes = {Processes, Resources}
  - Edges = {Request edges: $P_i \rightarrow R_j$, Assignment edges: $R_j \rightarrow P_i$}
  - Process



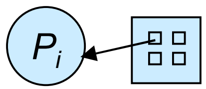


  - Resource Type with 4 instances



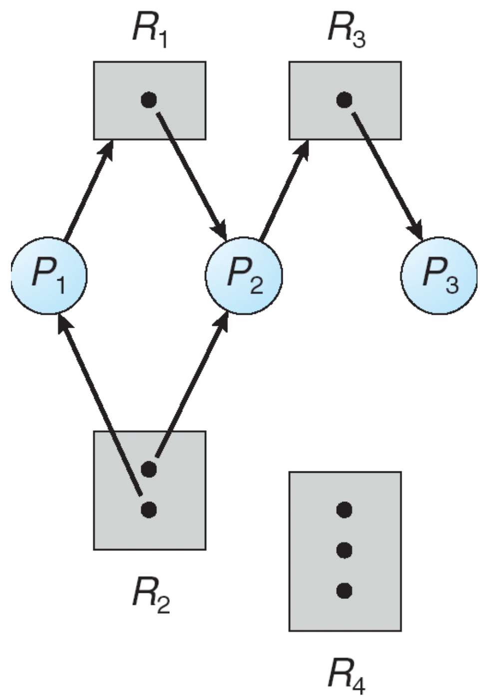
  - P_i requests instance of R_j

a directed edge $Pi \rightarrow Rj$ is called a **request edge**



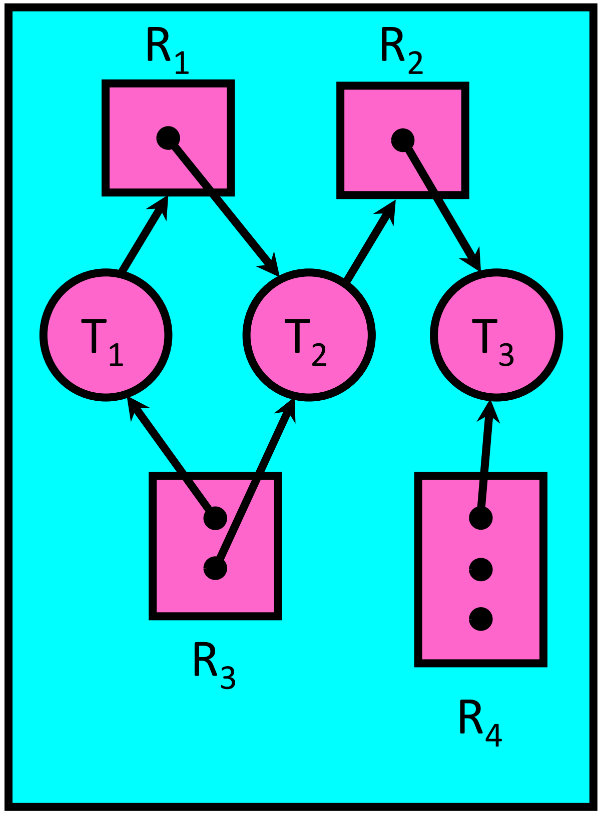
  - P_i is holding an instance of R_j

a directed edge $Rj \rightarrow Pi$ is called an **assignment edge**

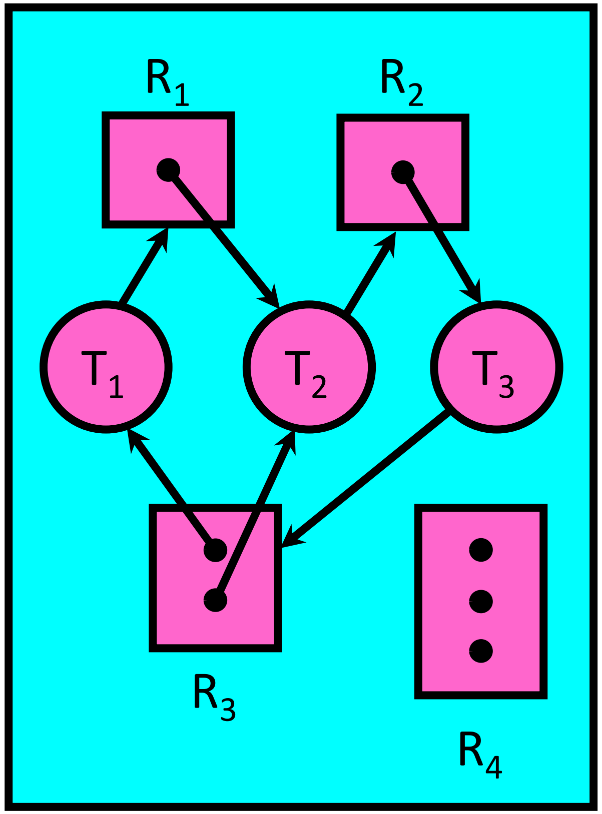


Resource-Allocation Graph Examples

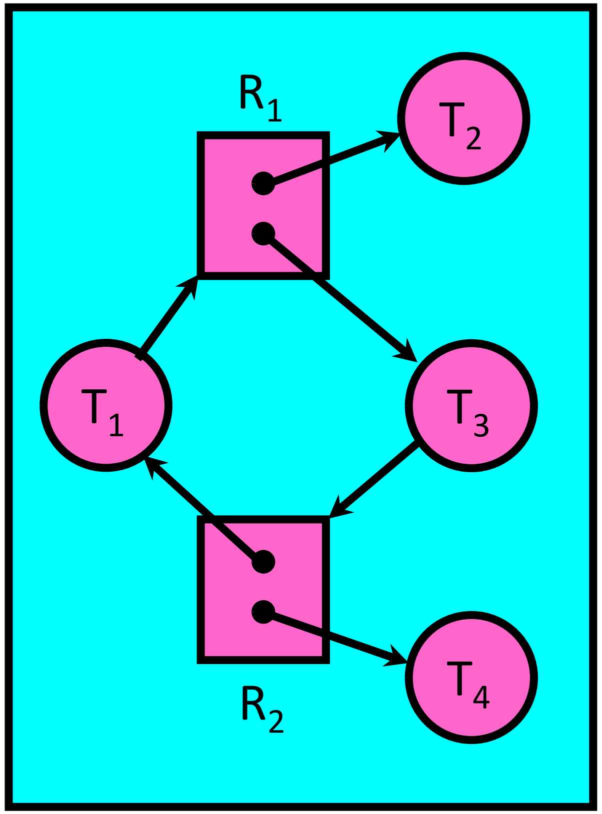
- Model:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph



Allocation Graph With Deadlock



Allocation Graph With Cycle, but No Deadlock

Deadlock illustration in RAG

- Having 1 resource in each resource type
  - Deadlock ⇔ existing a cycle
- Having multiple resources for at least one resource type
  - Deadlock is possible (not necessary) when existing a cycle
- No Cycle → No deadlock
- One cycle →
  - (one instance per resource type): deadlock
  - (multiple instances per resource type): possible of deadlock

How to handle deadlocks?

- 1) Prevent or 2)avoid deadlocks
  - Deadlock prevention
    - To ensure at least one of necessary condition cannot hold.
  - Deadlock avoidance
    - Conservative: May current req make a deadlock in future (non safe state)?
- 3) Detect & recover deadlocks
  - 1st detect, next recover (how?)
- 4) Do nothing: Ostrich algorithm!
  - Modern OS: Windows, Linux

1) Deadlock prevention

- Prevent deadlock by missing one of:
  - Mutual exclusion
    - Make resources sharable: read-only files
    - Is not possible in all cases
  - Hold & wait
    - How?
      - Request all resources before execution
      - Request a resource if no others it have
    - Drawbacks
      - Underutilization of resources
      - Starvation
  - No preemption
    - 3 solutions exist = {self preemption, dest process preemption, save & switch resource status}
  - Circular wait
    - Request a resource in an increasing order of enumeration

Example of prevention of circular wait

- Example of ~circular wait
  - define a one-to-one function F: $R \rightarrow N$ where N is the set of natural numbers.
  - F(tape drive) = 1
  - F(disk drive) = 5
  - F(printer) = 12
- Protocol
  - After having resource type $\boldsymbol{R_i}$ request to resource type $\boldsymbol{R_j}$ is possible if $\boldsymbol{F(R_j)>F(R_i)}$
  - Otherwise, release all resource types $\boldsymbol{R_i}$ where $\boldsymbol{F(R_i)>=F(R_j)}$

For example

- Suppose the ordering of tapes, disks, and printers are 1, 5, and 12.
  - F(tape drive) = 1
  - F(disk drive) = 5
  - F(printer) = 12
- If a process holds a disk (5),
- it can only ask a printer (12) and cannot request a tape (1).
- process must release some lower order resources to request a lower order resource. To get tapes (1), a process must release its disk (5).

In this way, no deadlock is possible. Why?

Proof:

- Let the set of processes involved in the circular wait be $\{P0, P1, ..., Pn\}$,
  - where Pi is waiting for a resource Ri, which is held by process $Pi+1$.

- Then, since process $Pi+1$ is holding resource Ri while requesting resource $Ri+1$,
- we must have $F(Ri) < F(Ri+1)$ for all i. But this condition means that $F(R0)$

  $< F(R1) < ... < F(Rn) < F(R0)$
- By transitivity,
  $F(R0) < F(R0)$, which is impossible.

- Therefore, there can be no circular wait

2) Deadlock avoidance

- Prevention is bad (why?)
  - Resource underutilization
  - Reduced throughput
- Avoidance is good; How to avoid?
  - Required extra info
    - Available resources
    - Resources allocated to processes
    - Future request of processes (!)
  - Definition:
    - Safe state
  - Solutions
    - 1) Resource-allocation-graph algorithm (single instance resource type)
    - 2) Banker's algorithm (multiple instance resource type)

Safe state definition

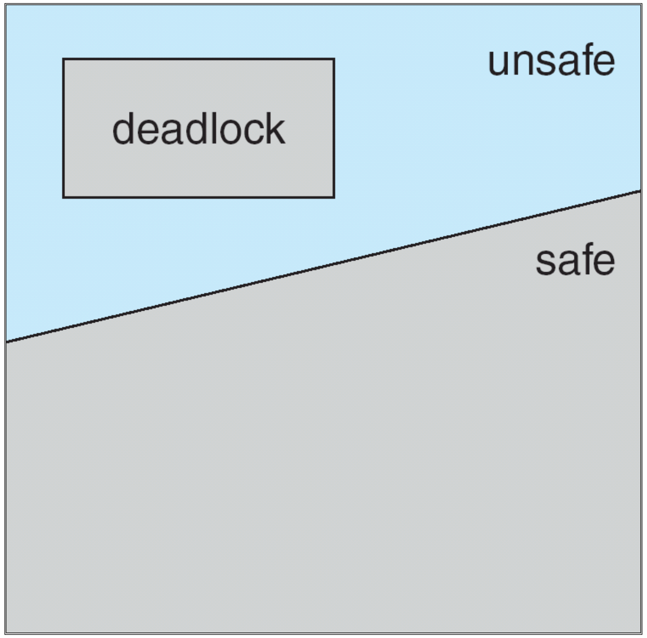
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

- Safe state: there exists a sequence $<P_1, P_2, \ldots, P_n>$ 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

Safe state definition

- 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 mode example (1)

- 3 processes: P_0, P_1, P_2
- 1 resource type: A (12)
- Snapshot at time T_0

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

- Safe mode sequence?

$<P_1, P_0, P_2>$

Safe mode example (2)

- 3 processes: P_0, P_1, P_2
- 1 resource type: A (12)
- Snapshot at time T_0

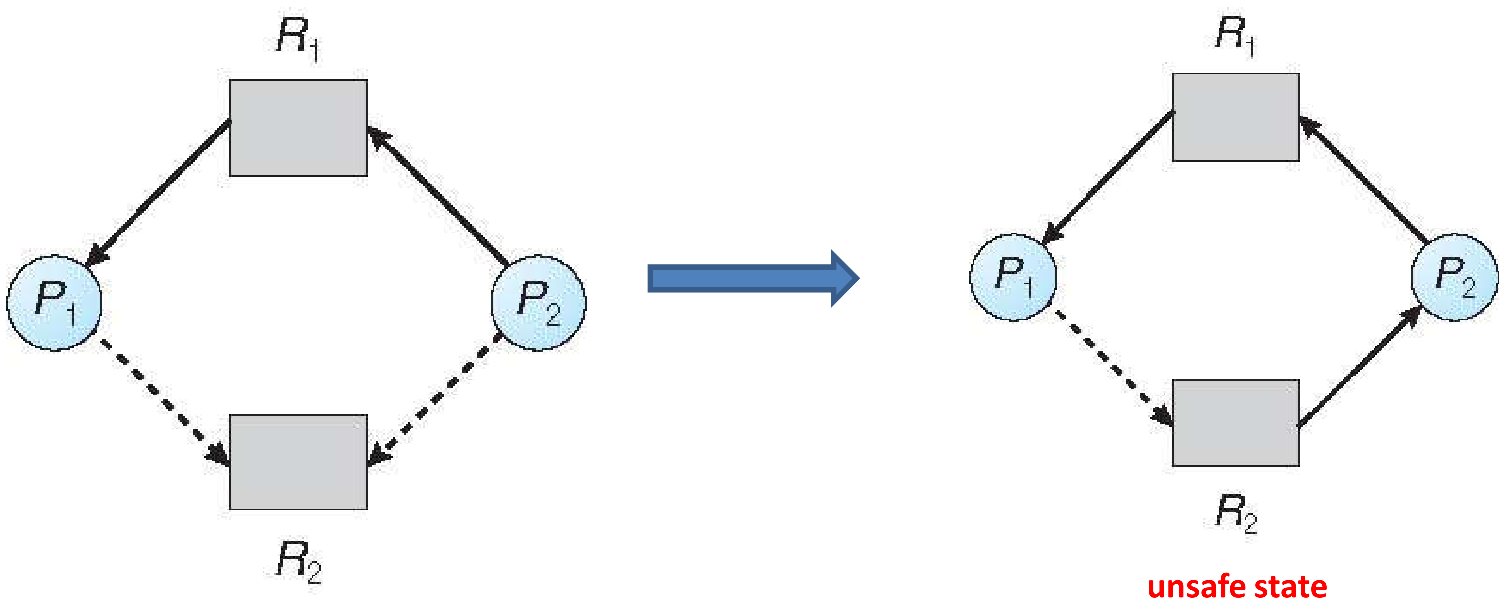
	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

- Suppose that, at time T1, process P2 requests and is allocated one more resource.
- Safe mode sequence?

no safe

2.1) Resource-allocation graph (RAG) algorithm

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j represented by a dashed line



2.1) Resource-allocation graph (RAG) algorithm

Suppose that process P_i requests a resource R_j

The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm for Preventing Deadlock

- Toward right idea:
  - State maximum (max) resource needs in advance
  - Allow particular process to proceed if:

    (available resources - #requested) $\geq$ max remaining that might be needed by any process

- Banker's algorithm:
  - Allocate resources dynamically
    - Evaluate each request and grant if some ordering of processes is still deadlock free afterward
    - Technique: pretend each request is granted, then run deadlock detection algorithm, substituting ($[Max_{node}]-[Alloc_{node}]$ <= [Avail]) for ($[Request_{node}]$ <= [Avail])
      Grant request if result is deadlock free (conservative!)

2.2) 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 x 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 $\text{Allocation}[i,j] = k$ 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

$Need\ [i,j] = Max[i,j] – Allocation\ [i,j]$

2.2) Banker’s algorithm(Cont.)

- Banker’s Algorithm:
  – An example of the Alloc[i,j] matrix:

Resources

Processes

$$
\begin{array}{cc}
 & \begin{array}{cccccc} & & \text{Column } j & & & \end{array} \\
\begin{array}{c} \\ \\ \\ \text{Row } i \\ \\ \\ \\ \\ \end{array} & \left[\begin{array}{cccccc}
 & & 1 & & & \\
 & & 7 & & & \\
 & & 12 & & & \\
8 & 0 & 2 & 17 & 0 & 1 \\
 & & 0 & & & \\
 & & 4 & & & \\
 & & 0 & & & \\
 & & 1 & & &
\end{array}\right]
\end{array}
$$

$Alloc_i$ is shorthand for row i of matrix Alloc[i,j], i.e. the resources allocated to process i

2.2) Banker's algorithm(Cont.)

- Banker's Algorithm:
  - Some terminology:
    - let X and Y be two vectors. Then we say $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all i.
    - Example:

$$V1 = \begin{bmatrix} 1 \\ 7 \\ 3 \\ 2 \end{bmatrix} \quad V2 = \begin{bmatrix} 0 \\ 3 \\ 2 \\ 1 \end{bmatrix} \quad V3 = \begin{bmatrix} 0 \\ 10 \\ 2 \\ 1 \end{bmatrix}$$

then $V2 \leq V1$, but $V3 \not\leq V1$, i.e. V3 is not less than or equal to V1

2.2) Banker's algorithm(Cont.)

- Example 3:
  - 3 resources (A,B,C) with total instances available (10,5,7)
  - 5 processes
  - At time t0, the allocated resources Alloc[i,j], Max needs Max[i,j], and Available resources Avail[j], are:

Alloc[i,j]

$$\begin{array}{c} \\ P0 \\ P1 \\ P2 \\ P3 \\ P4 \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix} \end{array}$$

Max[i,j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{bmatrix} \end{array}$$

Avail[j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 3 & 3 & 2 \end{bmatrix} \end{array}$$

Need[i,j]

$$\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{bmatrix} \end{array}$$

where Need[i,j] is computed given Alloc[i,j] and Max[i,j]

2.2) Banker's algorithm: Safety algorithm

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

    Work = Available

    Finish [i] = false for i = 0, 1, …, n- 1

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

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

    (b) ***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**

2.2) Banker's algorithm: Resource-request algorithm for process P_i

Request$_i$ = request vector for process ***Pi***
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
   else
   raise error condition (process has exceeded its maximum claim)
2. If $Request_i \leq Available$
   go to step 3
   else
   P_i must wait (resources are not available)
3. Pretend to allocate requested resources to ***Pi*** 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 Pi
If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored

Banker's algorithm example : safe

initialization

MAX

	R1	R2	R3
T1	3	2	2
T2	6	1	3
T3	3	1	4
T4	4	2	2

Allocation

	R1	R2	R3
T1	1	0	0
T2	6	1	2
T3	2	1	1
T4	0	0	2

Need

	R1	R2	R3
T1	2	2	2
T2	0	0	1
T3	1	0	3
T4	4	2	0

Max resources

R1	R2	R3
9	3	6

Available

R1	R2	R3
0	1	1

Banker's algorithm example : safe

T2

Max

	R1	R2	R3
T1	3	2	2
T2	0	0	0
T3	3	1	4
T4	4	2	2

Allocation

	R1	R2	R3
T1	1	0	0
T2	0	0	0
T3	2	1	1
T4	0	0	2

Need

	R1	R2	R3
T1	2	2	2
T2	0	0	0
T3	1	0	3
T4	4	2	0

R

R1	R2	R3
9	3	6

available

R1	R2	R3
6	2	3

Banker's algorithm example : safe

T1

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	3	1	4
T4	4	2	2

Max

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	2	1	1
T4	0	0	2

Allocation

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	1	0	3
T4	4	2	0

Need

R1	R2	R3
9	3	6

R

R1	R2	R3
7	2	3

available

Banker's algorithm example : safe

T3

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	4	2	2

Max

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	0	0	2

Allocation

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	4	2	0

available

R1	R2	R3
9	3	6

R

R1	R2	R3
9	3	4

need

Banker’s algorithm example : unsafe

intialization

Max

	R1	R2	R3
T1	3	2	2
T2	6	1	3
T3	3	1	4
T4	4	2	2

Allocation

	R1	R2	R3
T1	1	0	0
T2	5	1	1
T3	2	1	1
T4	0	0	2

need

	R1	R2	R3
T1	2	2	2
T2	1	0	2
T3	1	0	3
T4	4	2	0

R

R1	R2	R3
9	3	6

available

R1	R2	R3
1	1	2

Banker's algorithm example : unsafe

T1 request R1 and R3

	R1	R2	R3
T1	3	2	2
T2	6	1	3
T3	3	1	4
T4	4	2	2

Max

	R1	R2	R3
T1	2	0	1
T2	5	1	1
T3	2	1	1
T4	0	0	2

allocation

	R1	R2	R3
T1	1	2	1
T2	1	0	2
T3	1	0	3
T4	4	2	0

need

R1	R2	R3
9	3	6

R

R1	R2	R3
0	1	1

available

Banker's algorithm example

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

	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	

	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

Is the system safe?

Yes. The sequence <P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2)

Check that Request $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ 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

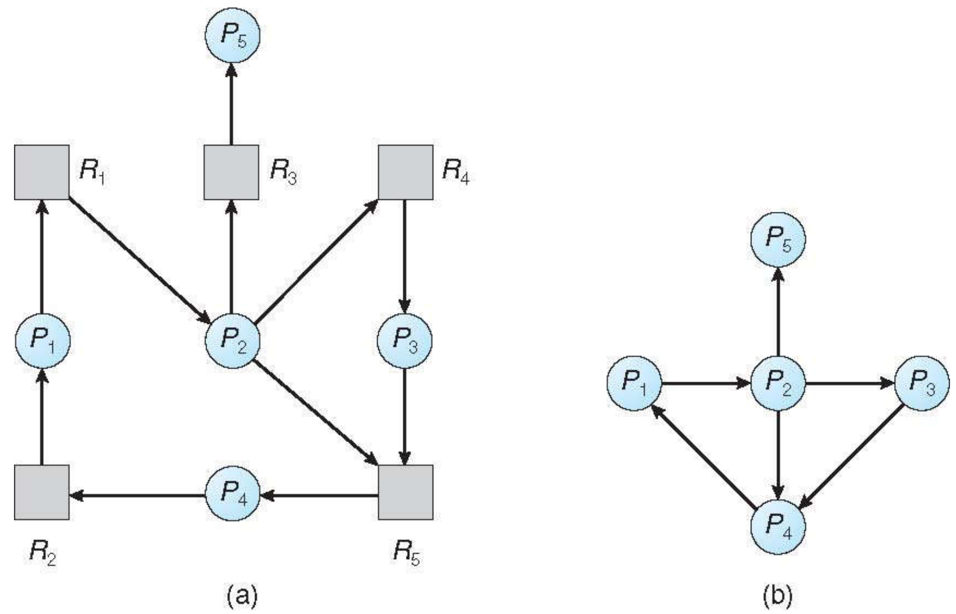
Can request for (3,3,0) by P_4 be granted?
Can request for (0,2,0) by P_0 be granted?

3) Deadlock detection & recovery

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

3.1) Resource-allocation graph and wait-for graph

- For single instance resource type:
  - Create wait-for graph from RAG
  - Periodically run cycle detector



Resource-Allocation Graph — Corresponding wait-for graph

3.2) Several instances of a resource type

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

Allocation: n x m matrix.

If Allocation[i,j] = k then P_i is currently allocated k instances of R_j

Request: n x m matrix.

If *Request*[i,j] = k, then P_i is requesting k more instances of R_j

3.2) Several instances of a resource type

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

   Work = Available

   for ***i = 0, 1, …, n- 1*** if ***$Allocation_i \neq$ 0*** then ***Finish [i] = false***

   else Finish [i] = true

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

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

   (b) ***$Request_i \leq Work$***

   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] == false*** for some *i*, then the system is in a **deadlock 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	

Deadlock?

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

	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	

	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

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?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. 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

Questions?