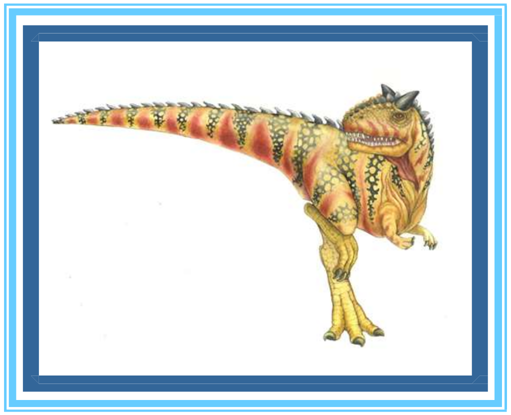
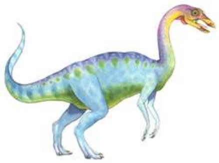
Chapter 8: Deadlocks



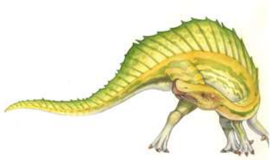
Outline

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock



Chapter Objectives

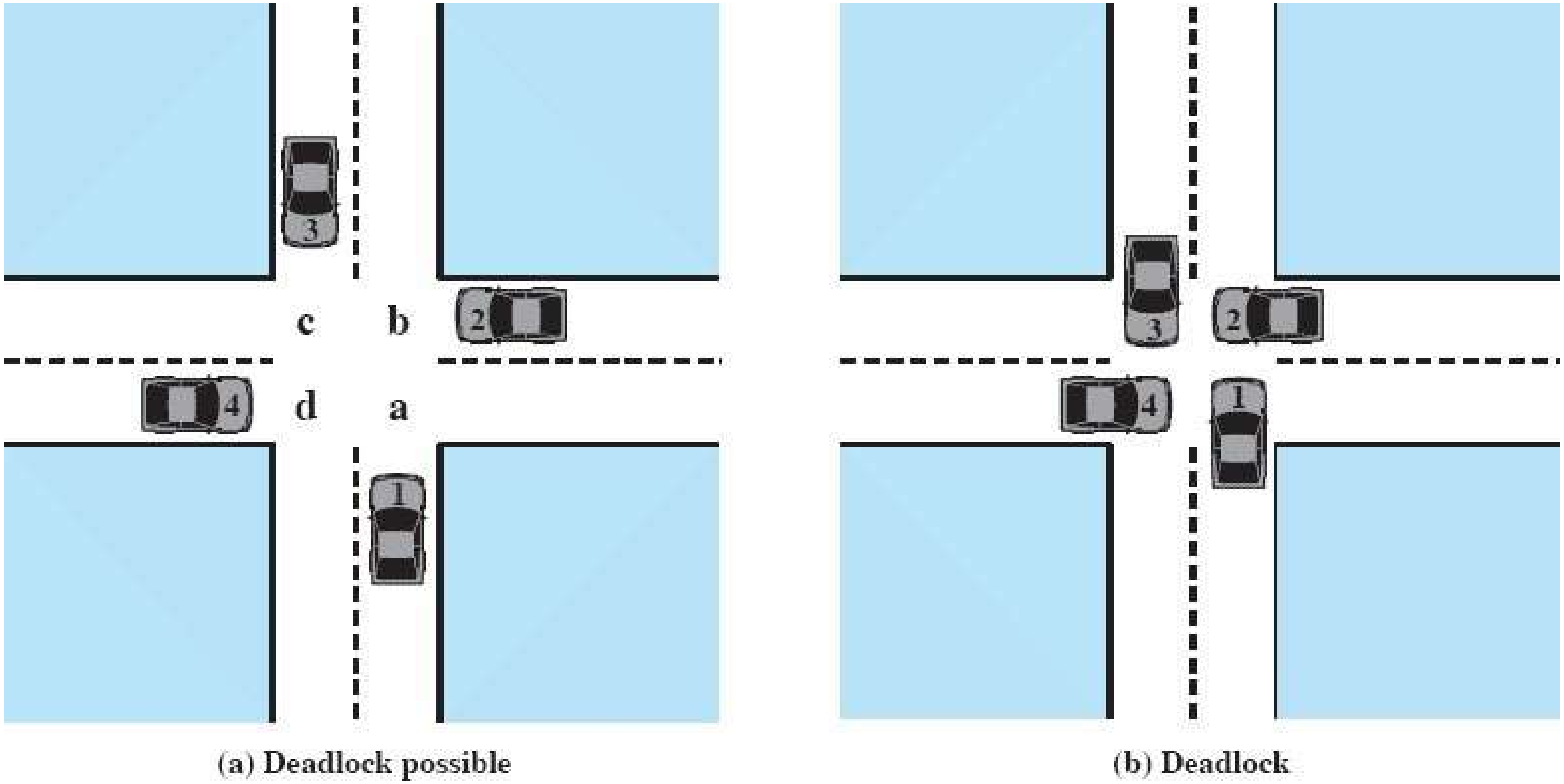
- Illustrate how deadlock can occur when mutex locks are used
- Define the four necessary conditions that characterize deadlock
- Identify a deadlock situation in a resource allocation graph
- Evaluate the four different approaches for preventing deadlocks
- Apply the banker’s algorithm for deadlock avoidance
- Apply the deadlock detection algorithm
- Evaluate approaches for recovering from deadlock



Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- No efficient solution
  - Most OSes do not prevent or deal with deadlocks
  - But such features will probably be added soon

Deadlock



(a) Deadlock possible

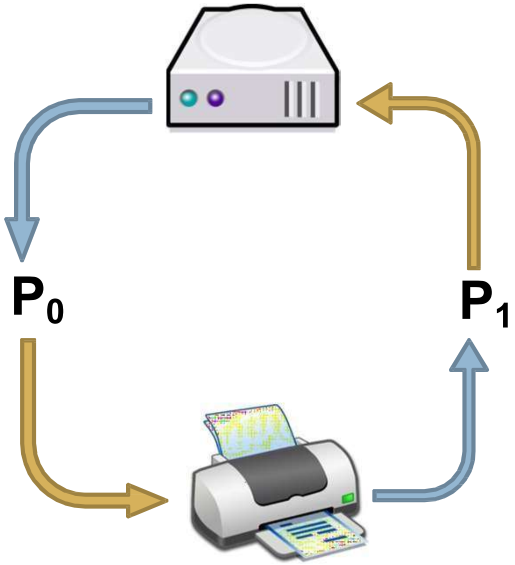
(b) Deadlock

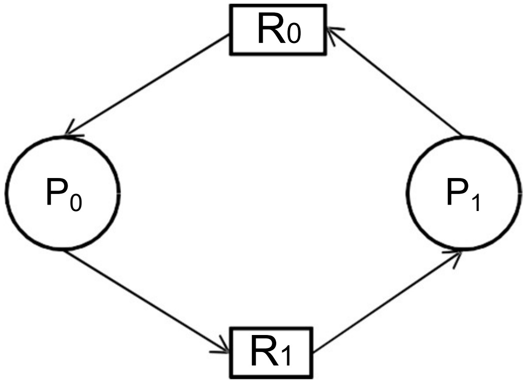
Figure 6.1 Illustration of Deadlock

Deadlock

Example:

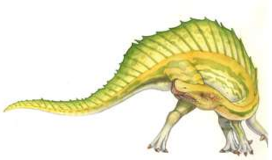
P_0	P_1
…	…
→ Request(Disk)	→ Request(Printer)
…	…
Request(Printer)	Request(Disk)
…	…
Release(Disk)	Release(Printer)
…	…
Release(Printer)	Release(Disk)
…	…





System Model

- System consists of resources
- Resource types R_1, R_2, . . ., 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**
    - If the request cannot be granted immediately, then the process must wait until it can acquire the resource.
  - **use**
  - **release**



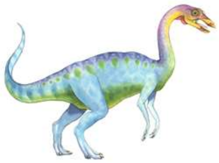
Deadlock with Semaphores

- 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)
```



Necessary Conditions for Deadlock

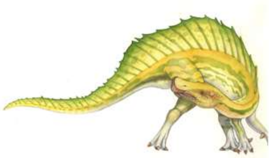
Deadlock can arise if **four** conditions hold simultaneously (*Coffman conditions*):

1. Mutual exclusion: only one process at a time can use a resource

2. Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes

3. No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task

- The first three conditions are **necessary** but not **sufficient** for a deadlock to exist



Necessary Conditions for Deadlock

Deadlock can arise if **four** conditions hold simultaneously (*Coffman conditions*):

4. Circular wait: there exists a set {P_0, P_1, …, P_n} of waiting processes such that

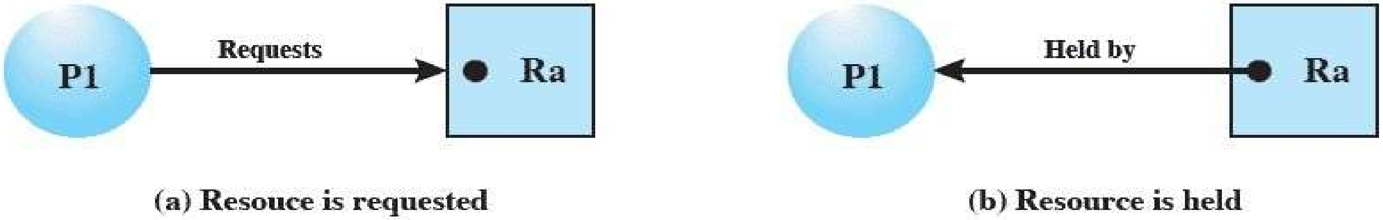
- P_0 is waiting for a resource that is held by P_1,
- P_1 is waiting for a resource that is held by P_2,
- …, P_{n-1} is waiting for a resource that is held by P_n,
- and P_n is waiting for a resource that is held by P_0.

- The first three conditions are **necessary** but not **sufficient** for a deadlock to exist
- For deadlock to actually take place, the fourth condition is required

Resource-Allocation Graph

A directed 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
- E is also portioned into two types:
  - **request edge** – directed edge $P_i \rightarrow R_j$
  - **assignment edge** – directed edge $R_j \rightarrow P_i$

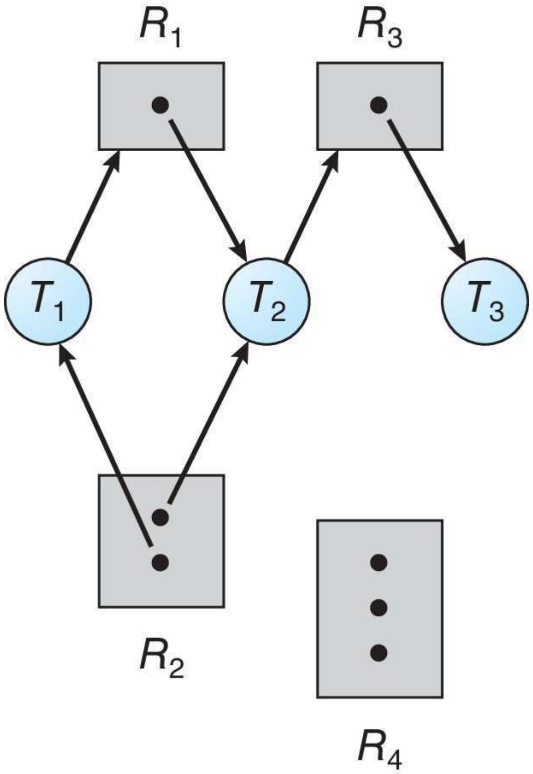


(a) Resouce is requested

(b) Resource is held

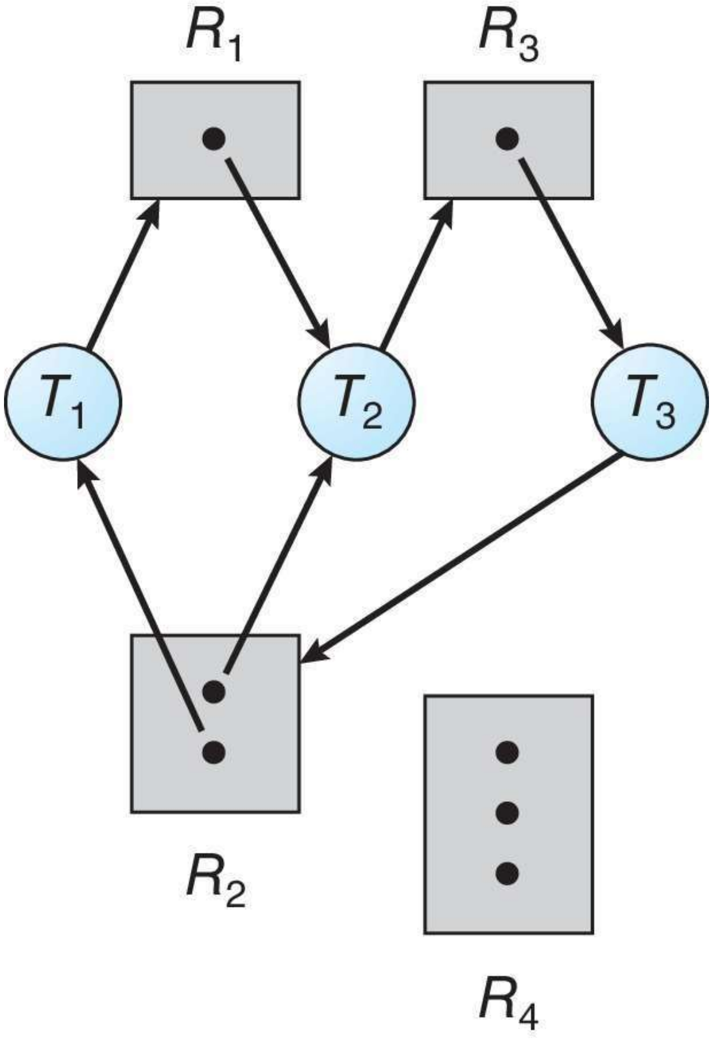
Resource Allocation Graph Example

- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4
- T_1 holds one instance of R_2 and is waiting for an instance of R_1
- T_2 holds one instance of R_1, one instance of R_2, and is waiting for an instance of R_3
- T_3 is holds one instance of R_3



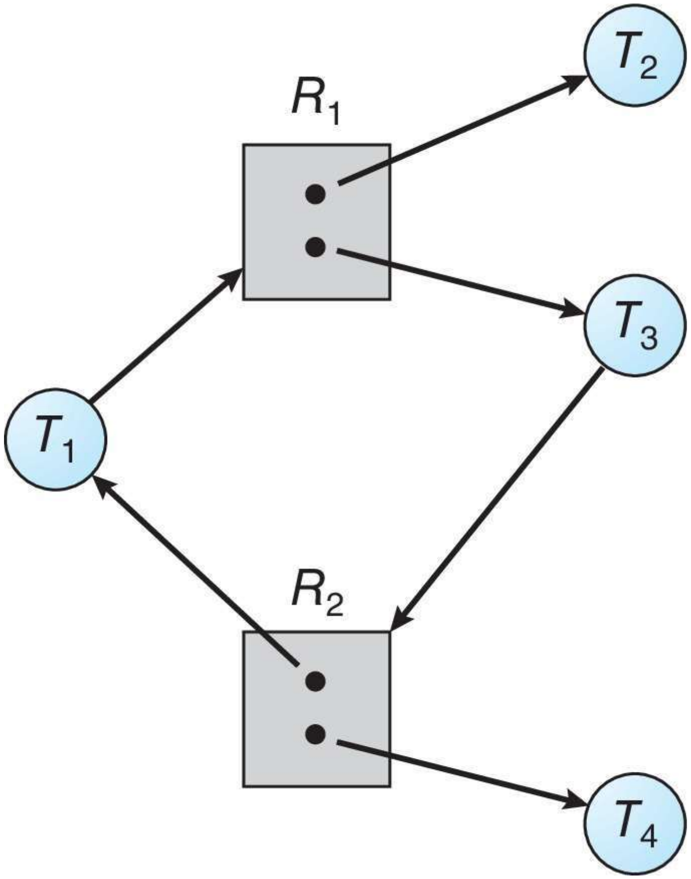
Deadlock?

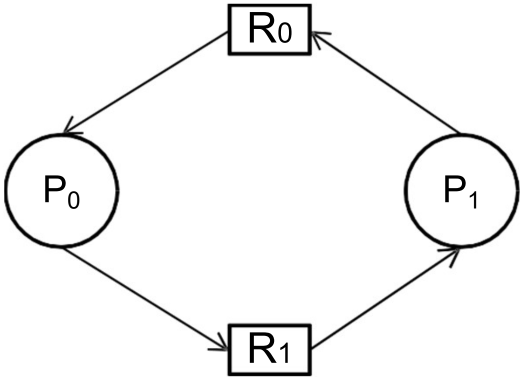
Resource Allocation Graph



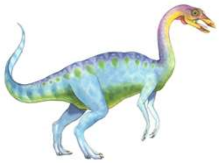
Deadlock?

Resource Allocation Graph



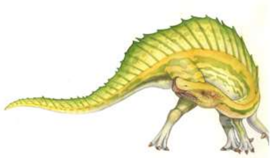


Deadlock?



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



Methods for Handling Deadlocks

- Ostrich
  - Ignore the problem and pretend that deadlocks **never** occur in the system
  - Used by most operating systems, including UNIX, Windows

- Ensure that the system will **never** enter a deadlock state:
  - Deadlock **prevention**
    - A set of methods for ensuring that at least one of the necessary conditions cannot hold
  - Deadlock **avoidance**
    - Tries to avoid deadlock by delaying the requests which may result in a deadlock.
    - Requires additional information concerning which resources a process will request and use during its life time.

- Allow the system to enter a deadlock state and then **detect** and **recover**
  - □ 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
  - In general, the first of the four listed conditions cannot be disallowed
  - If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS.

- **Hold and Wait**
  - must guarantee that whenever a process requests a resource, it does not hold any other resources

Deadlock Prevention

- **Hold and Wait (Cont.)**
  - 1. Require a process request all of its required resources at one time (e.g., at the beginning)
  - 2. Allow process to request resources only when the process has none
    - i.e., before a process can request any additional resources, it must release all the resources that it is currently allocated
  - Disadvantages
    - Low resource utilization
    - Starvation possible: a process that needs several popular resources may have to wait indefinitely

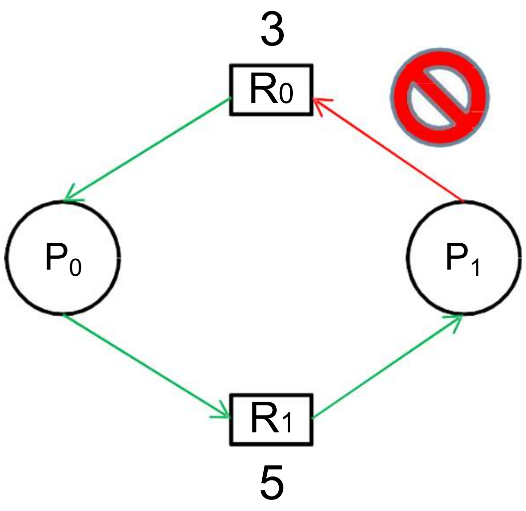
Deadlock Prevention

- **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 implicitly 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

Deadlock Prevention

- **No Preemption (Cont.)**
  - Disadvantage
    - It is often applied to resources whose state can be easily saved and restored later, such as CPU registers
    - It cannot generally be applied to such resources as printers and tape drives
- **Circular Wait**
  - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Prevention



- **Circular Wait (Cont.)**
  - Disadvantage: It is up to application developers to write programs that follow the ordering

Deadlock Avoidance

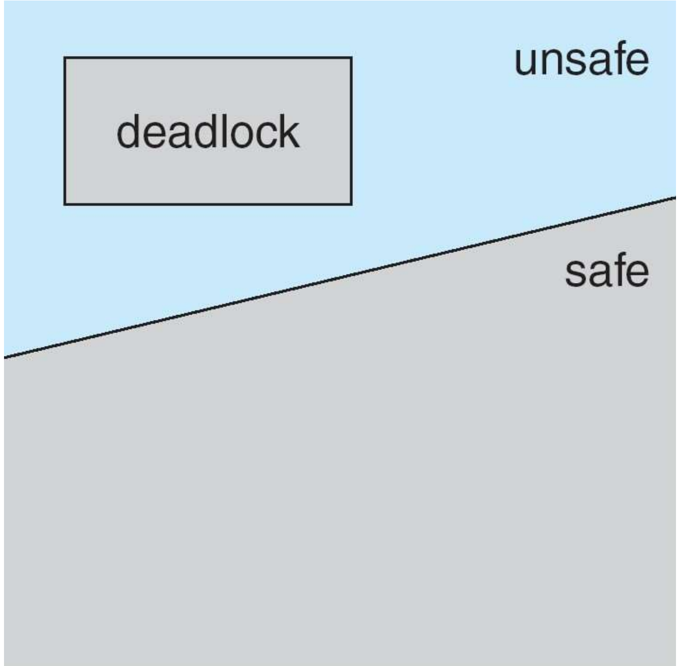
- In deadlock prevention, we constrain resource requests which leads to inefficient use of resources and inefficient execution of processes
- With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will potentially lead to a deadlock
- Deadlock avoidance thus requires knowledge of future process resource requests

Safe State

- A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock (i.e., all of the processes can be run to completion)
- 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 plus resources held by all the P_j, with $j < i$

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



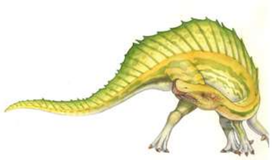
Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph

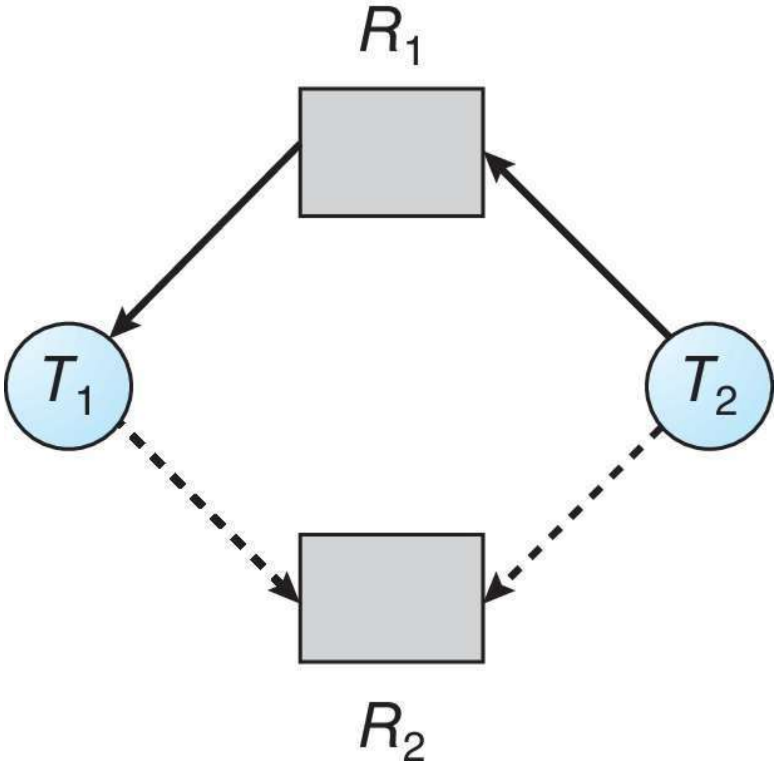
- Multiple instances of a resource type
  - Use the Banker’s Algorithm

Resource-Allocation Graph Scheme

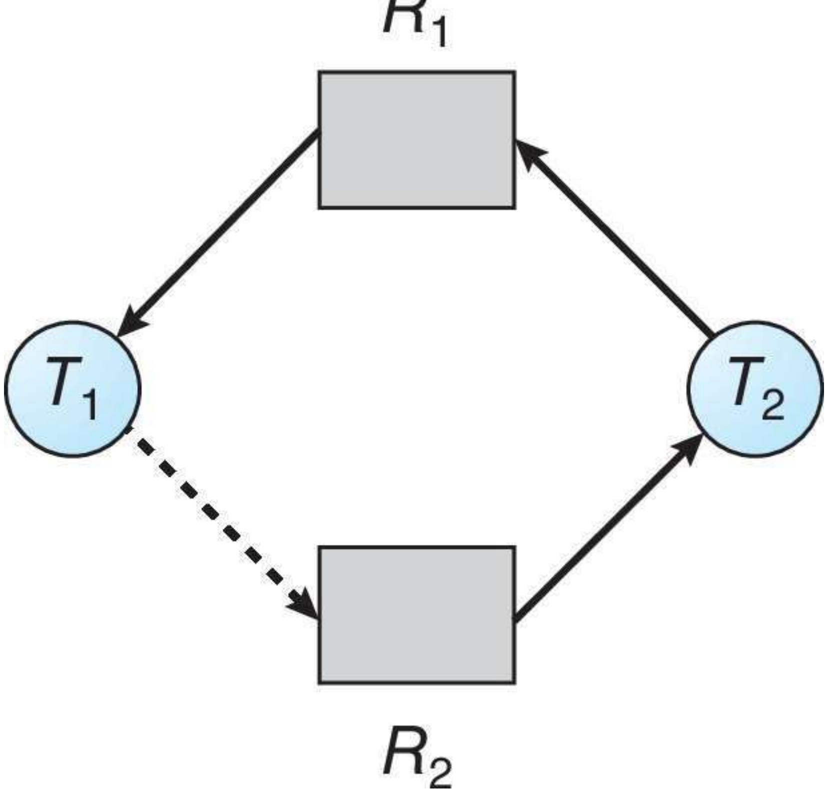
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system



Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Banker’s Algorithm

- The banker’s algorithm is used when we have Multiple instances of a resource type
  - For Single instance of a resource type, we can use resource-allocation graph (see Silberschatz)
- The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers
- A new process must declare the maximum number of instances of each resource type that it may need (weakness of this algorithm)

Data Structures for Banker’s Alg.

- These data structures encode the state of the resource-allocation system
- Let n be the number of processes, and m be the number of resources types
- Resource: Vector of length m indicates total amount of each resource in the system.
- Available: Vector of length m indicates the number of available resources of each type.
  - If available[j] = k, there are k instances of resource type R_j available

Data Structures for Banker's Alg.

- Max: n × m matrix defines the maximum demand of each process. If Max [i, j] = k, then process P_i may request at most k instances of resource type R_j
- Allocation: n × m matrix. If Allocation[i, j] = k then P_i is currently allocated k instances of R_j
- Need: n × 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]

Banker's Algorithm: an Example

Is the system safe?

$$\text{Resource} = \begin{array}{cccc} R_0 & R_1 & R_2 & R_3 \\ \hline \multicolumn{1}{|c|}{5} & \multicolumn{1}{c|}{7} & \multicolumn{1}{c|}{2} & \multicolumn{1}{c|}{3} \\ \hline \end{array} \qquad \text{Avail} = \begin{array}{cccc} R_0 & R_1 & R_2 & R_3 \\ \hline \multicolumn{1}{|c|}{1} & \multicolumn{1}{c|}{1} & \multicolumn{1}{c|}{0} & \multicolumn{1}{c|}{1} \\ \hline \end{array}$$

	R_0	R_1	R_2	R_3
P_0	1	0	2	3
P_1	2	3	1	0
P_2	0	2	1	1
P_3	4	3	1	1
P_4	2	2	2	1

Max

	R_0	R_1	R_2	R_3
P_0	1	0	0	2
P_1	1	2	0	0
P_2	0	1	1	0
P_3	2	3	1	0
P_4	0	0	0	0

Allocation

	R_0	R_1	R_2	R_3
P_0	0	0	2	1
P_1	1	1	1	0
P_2	0	1	0	1
P_3	2	0	0	1
P_4	2	2	2	1

Need

$$(1,1,0,1) \xrightarrow{P_2} (1,2,1,1) \xrightarrow{P_1} (2,4,1,1) \xrightarrow{P_3} (4,7,2,1) \xrightarrow{P_0} (5,7,2,3) \xrightarrow{P_4} (5,7,2,3)$$ ✓

Banker’s Algorithm: an Example

P_4 asks for an instance of R_0, is it allocated?

Resource =

R_0	R_1	R_2	R_3
5	7	2	3

Avail=

R_0	R_1	R_2	R_3
0	1	0	1

	R_0	R_1	R_2	R_3
P_0	1	0	2	3
P_1	2	3	1	0
P_2	0	2	1	1
P_3	4	3	1	1
P_4	2	2	2	1

Max

	R_0	R_1	R_2	R_3
P_0	1	0	0	2
P_1	1	2	0	0
P_2	0	1	1	0
P_3	2	3	1	0
P_4	**1**	0	0	0

Allocation

	R_0	R_1	R_2	R_3
P_0	0	0	2	1
P_1	1	1	1	0
P_2	0	1	0	1
P_3	2	0	0	1
P_4	**1**	2	2	1

Need

$(0,1,0,1) \xrightarrow{P_2} (0,2,1,1)$ 🚫

Banker's Algorithm: an Example

P_1 asks for an instance of R_0, is it allocated?

Resource =

R_0	R_1	R_2	R_3
5	7	2	3

Avail=

R_0	R_1	R_2	R_3
0	1	0	1

	R_0	R_1	R_2	R_3
P_0	1	0	2	3
P_1	2	3	1	0
P_2	0	2	1	1
P_3	4	3	1	1
P_4	2	2	2	1

Max

	R_0	R_1	R_2	R_3
P_0	1	0	0	2
P_1	**2**	2	0	0
P_2	0	1	1	0
P_3	2	3	1	0
P_4	0	0	0	0

Allocation

	R_0	R_1	R_2	R_3
P_0	0	0	2	1
P_1	**0**	1	1	0
P_2	0	1	0	1
P_3	2	0	0	1
P_4	2	2	2	1

Need

$$(0,1,0,1) \xrightarrow{P_2} (0,2,1,1) \xrightarrow{P_1} (2,4,1,1) \xrightarrow{P_3} (4,7,2,1) \xrightarrow{P_0} (5,7,2,3) \xrightarrow{P_4} (5,7,2,3)$$ ✓

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) $Need_i \leq Work$

   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

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:

$Available = Available – Request;$

$Allocation_i = Allocation_i + Request_i;$

$Need_i = Need_i – Request_i;$

- *If safe* → *the resources are allocated to* P_i
- *If unsafe* → P_i *must wait, and the oldresource-allocation state is restored*

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

Banker’s Algorithm Disadvantages

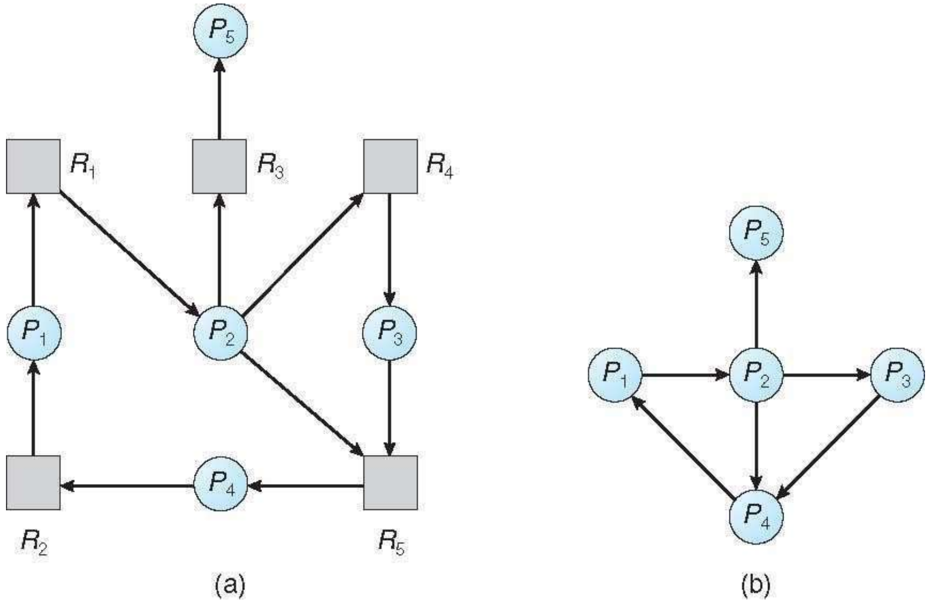
- Maximum resource requirement must be stated in advance
- There must be a fixed number of resources to allocate
- The algorithm is very conservative
  - It limits access to resources

Deadlock Detection and Recovery

- If a system does not employ either a deadlock prevention or a deadlock avoidance algorithm, then a deadlock situation may occur
- Then, the system must provide
  - An algorithm that examines the state of the system to determine whether a deadlock has occurred
  - An algorithm to recover from the deadlock

Deadlock Detection
Single Instance of Each Resource Type

- Use resource-allocation graph or a variant of it, called a wait-for graph
- Maintain **wait-for** graph
  - Nodes are processes
  - $\boldsymbol{P_i \rightarrow P_j}$ if $\boldsymbol{P_i}$ is waiting for $\boldsymbol{P_j}$
- 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 $\boldsymbol{n^2}$ operations, where $\boldsymbol{n}$ is the number of vertices in the graph



Resource-Allocation Graph

Corresponding wait-for graph

Deadlock Detection

- Several instances of each resource type
  - The algorithm is very similar to the banker's algorithm, with two main differences
    - The matrices Max and Need are replaced with an $n \times m$ matrix Request 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
    - If a process has a row in the Allocation matrix of all zeros, then it is marked as finished.
  - If there are unmarked processes at the end of the algorithm, these processes are deadlocked

Deadlock Detection: an Example

- Is there a deadlock in the system?

Resource =

R_0	R_1	R_2	R_3	R_4
2	1	1	2	1

Avail =

R_0	R_1	R_2	R_3	R_4
0	0	0	0	1

	R_0	R_1	R_2	R_3	R_4
P_0	1	0	1	1	0
P_1	1	1	0	0	0
P_2	0	0	0	1	0
P_3	0	0	0	0	0

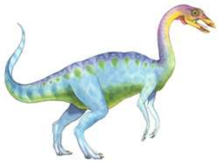
Allocation

	R_0	R_1	R_2	R_3	R_4
P_0	0	1	0	0	1
P_1	0	0	1	0	1
✓ P_2	0	0	0	0	1
✓ P_3	1	0	1	0	1

Request

$(0,0,0,0,1) \xrightarrow{P_2} (0,0,0,1,1)$ 🚫

Yes, P_0 and P_1 are deadlocked

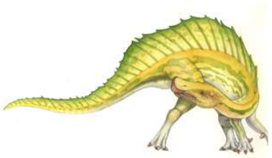


Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:
   a) ***Work* = *Available***
   b) For ***i* = 1,2, …, *n***, if ***Allocation***$_i$ **< 0**, then
      Finish*[i] = *false; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:
   a) ***Finish*[*i*] == *false***
   b) ***Request***$_i$ **< *Work***

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



Detection Algorithm (Cont.)

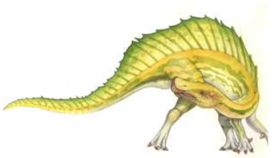
3. ***Work* = *Work* + *Allocation*$_i$**

   Finish*[*i*] = *true

   go to step 2

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



Deadlock Recovery

- Abort all deadlocked processes
  - One of the most common solution adopted in operating systems
  - Expensive solution
- Back up each deadlocked process to some previously defined checkpoint, and restart all process
  - Original deadlock may occur again
  - System requires rollback and restart mechanisms

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.

Deadlock Recovery

- Successively abort deadlocked processes until deadlock no longer exists
  - Incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked
- Successively preempt resources until deadlock no longer exists

Deadlock Recovery

- The last two methods requires some selection criteria to choose the victim process
- Choose the process with the
  - least amount of processor time consumed so far
  - least amount of output produced so far
  - most estimated time remaining
  - least total resources allocated so far
  - lowest priority

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

End of Chapter 8