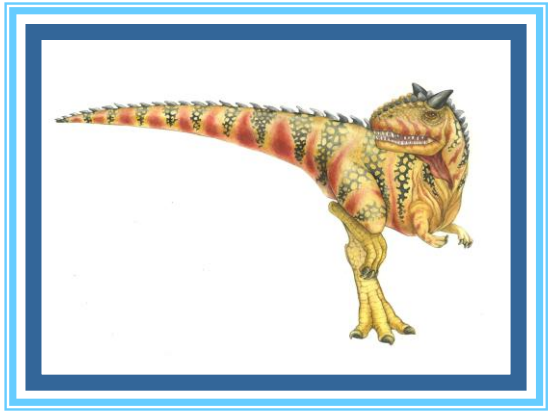
# Chapter 8: Deadlocks



# Chapter 8: Deadlocks

System Model

Deadlock in Multithreaded Applications

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

# System Model

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:

1. **request**
2. **use**
3. **release**

# Deadlock in Multithreaded Application

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

# Deadlock in Multithreaded Application

1st thread

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex); 1
   pthread_mutex_lock(&second_mutex); 2
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex); 2
   pthread_mutex_unlock(&first_mutex); 1

   pthread_exit(0);
}
```

2nd thread

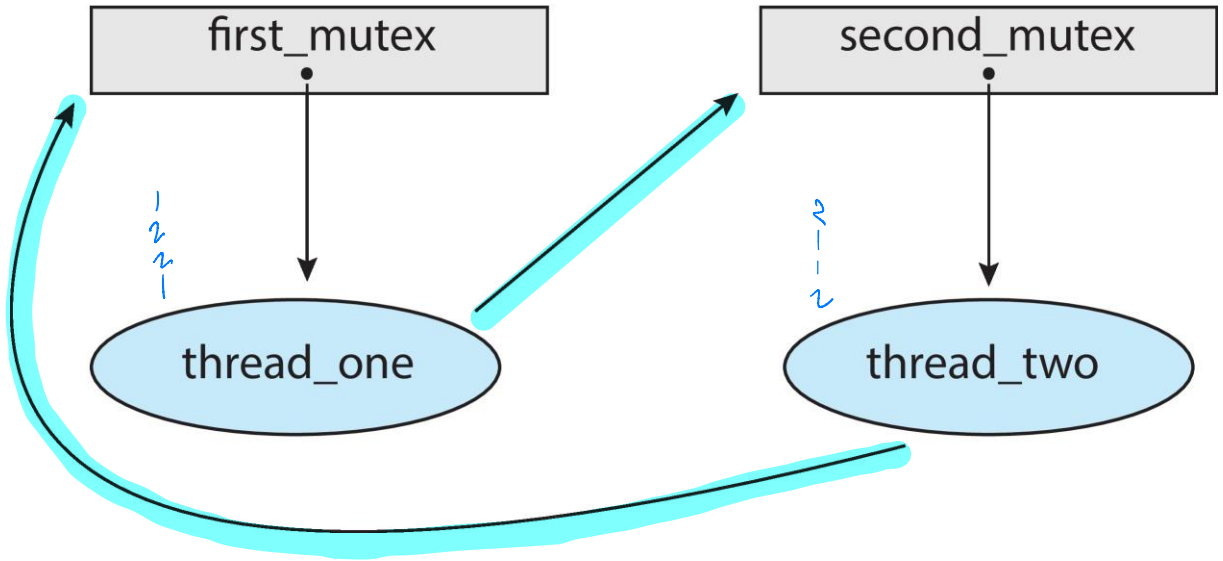
```
/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex); 2
   pthread_mutex_lock(&first_mutex); 1
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex); 1
   pthread_mutex_unlock(&second_mutex); 2

   pthread_exit(0);
}
```

# Deadlock in Multithreaded Application

Deadlock is possible if thread 1 acquires **`first_mutex`** and thread 2 acquires **`second_mutex`**. Thread 1 then waits for **`second_mutex`** and thread 2 waits for **`first_mutex`**. ⇒ deadlock

Can be illustrated with a **resource allocation graph**:




# Deadlock Characterization

Deadlock can arise if 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 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 $P_0$.

# 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

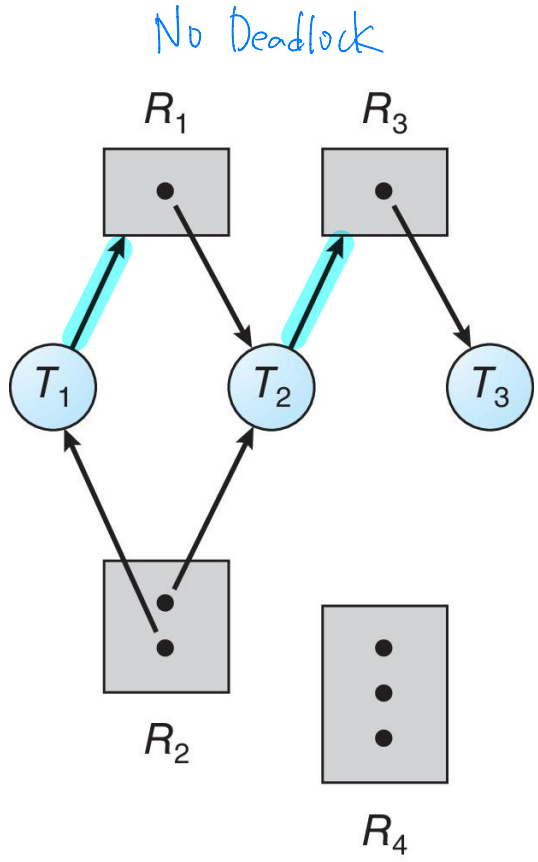
One instance of R3

Three instance of R4

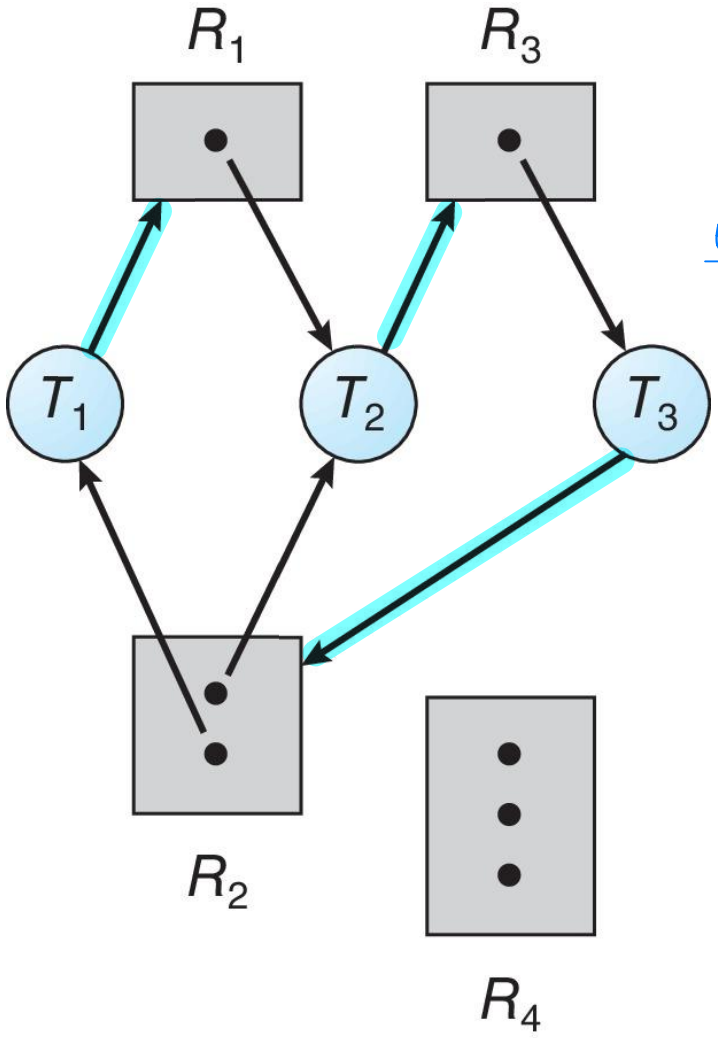
T1 holds one instance of R2 and is waiting for an instance of R1

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

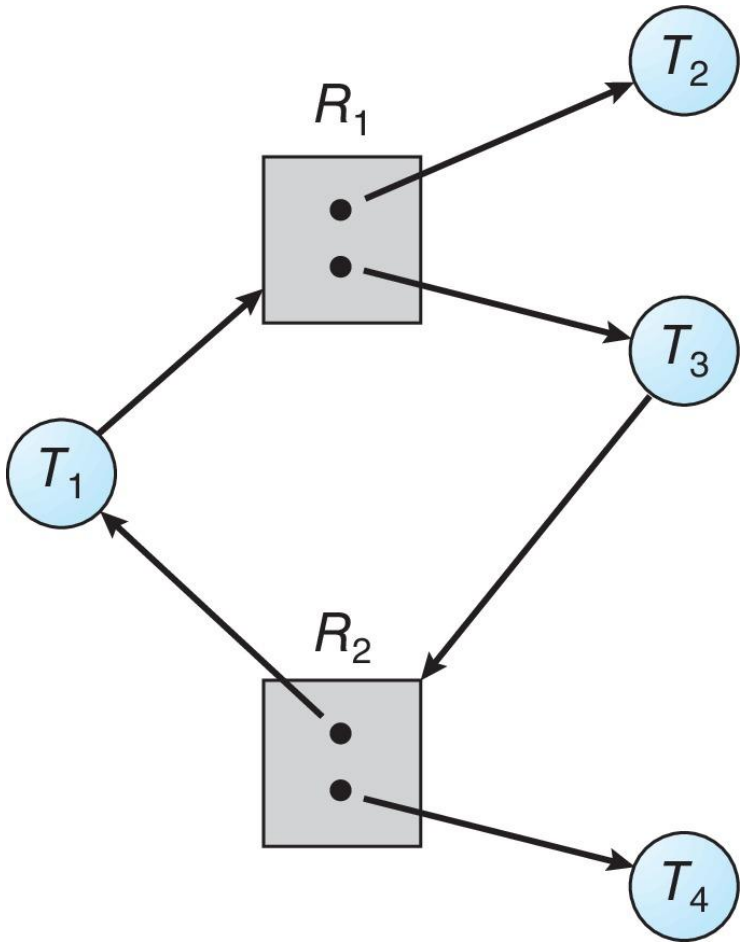



circular wait

# Graph With A Cycle But No Deadlock

$T_4 \rightarrow T_3 \rightarrow T_1$

$T_2 \rightarrow T_1 \rightarrow T_3$




# 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 that the system will *never* enter a deadlock state:

- Deadlock prevention
- Deadlock avoidance

Allow the system to enter a deadlock state and then recover

Ignore the problem and pretend that deadlocks never occur in the system.

# Deadlock Prevention ⇒ Severe performance degradation.

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 (Cont.)

**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

# Circular Wait

Invalidating the circular wait condition is most common.

Simply assign each resource (i.e. mutex locks) a unique number.

Resources must be acquired in order.

If:

**first_mutex = 1**
**second_mutex = 5**

code for **thread_two** could not be written as follows:

must increasing order

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex); 5
   pthread_mutex_lock(&first_mutex);  1
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

# Deadlock Avoidance

maintain overall system performance

Requires that the system has some additional ***a priori*** information available

given in advance

- 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 $<P_1, P_2, \ldots, P_n>$ 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

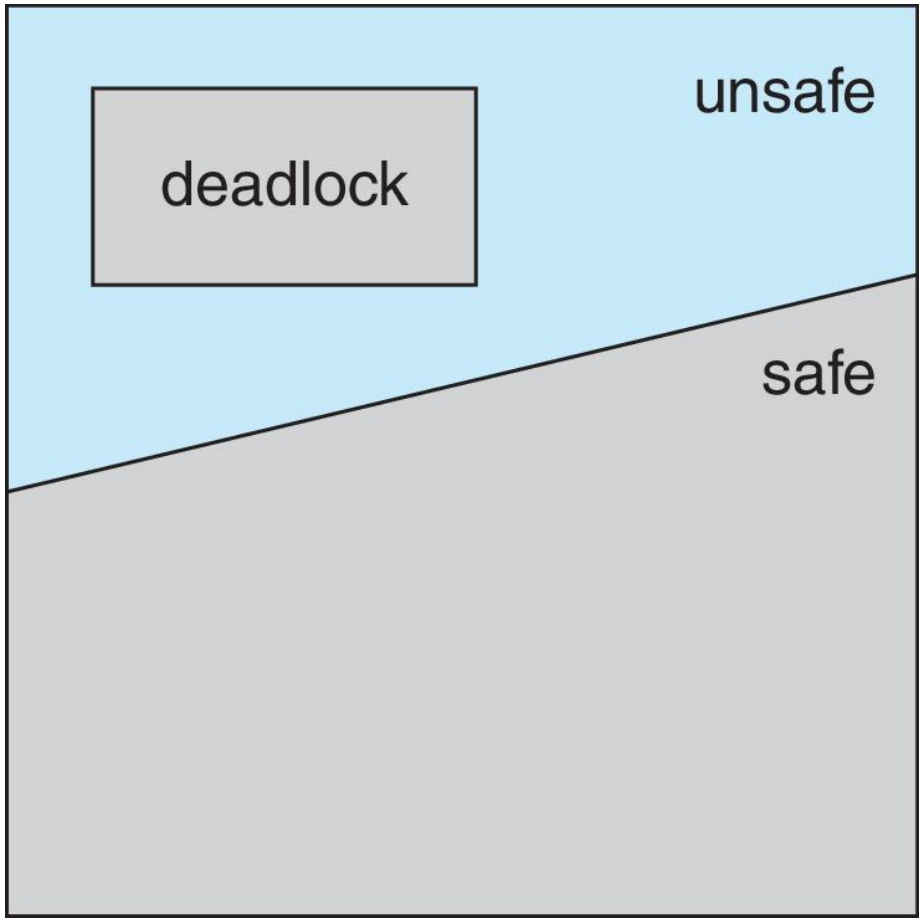
# Basic Facts

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, Unsafe, Deadlock 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

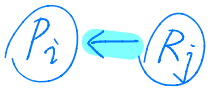
Single instance per resource

# Resource-Allocation Graph Scheme

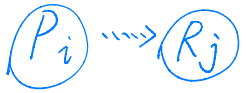
**Claim edge** $P_i \rightarrow R_j$ indicated that process $P_j$ may request resource $R_j$; represented by a dashed line


Claim edge converts to request edge when a process requests a resource (solid line)


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 Algorithm

Suppose that process $P_i$ requests a resource $R_j$

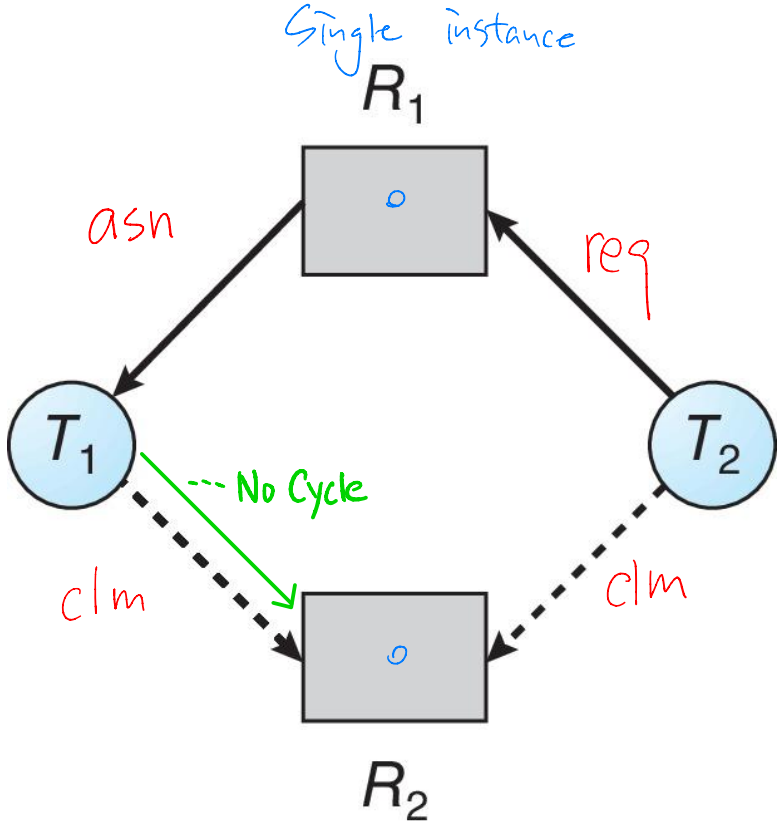
The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

$P_i \rightarrow R_j$

$\Downarrow$ No Cycle !

$P_i \leftarrow R_j$

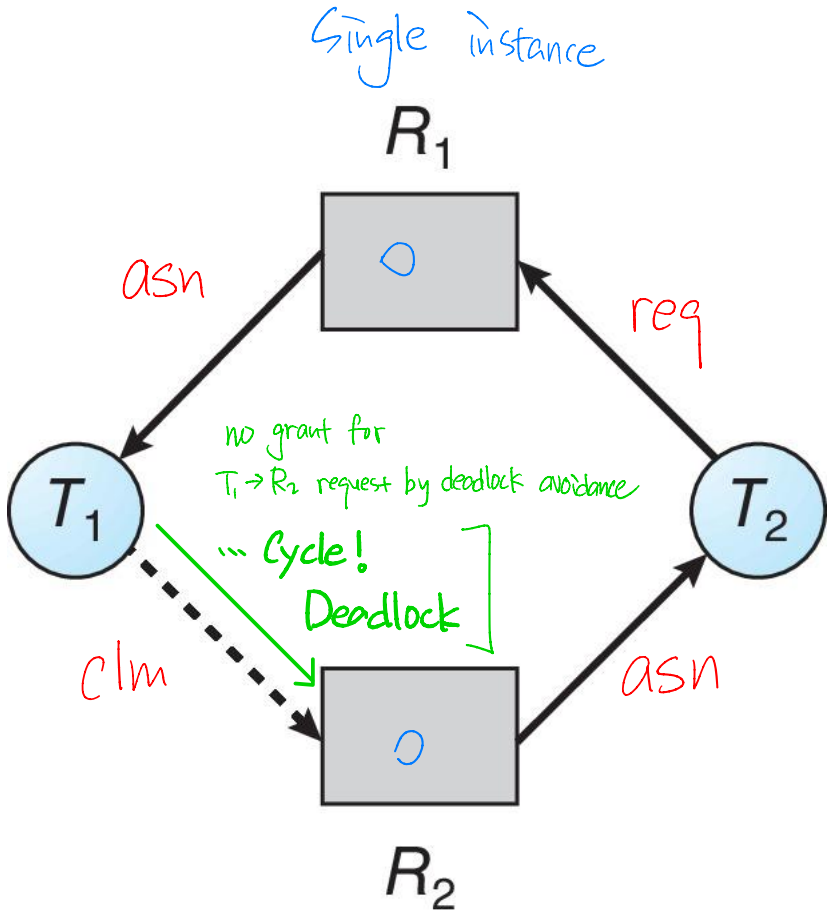
# Resource-Allocation Graph




No Cycle

consist of solid line (req, asn)

# Unsafe State In Resource-Allocation Graph




No Cycle

# 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

eventally

# Data Structures for the Banker's Algorithm

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

1. **Available**: Vector of length $m$. If available [$j$] = $k$, there are $k$ instances of resource type $R_j$ available

2. **Max**: $n$ x $m$ matrix. If $Max$ [$i,j$] = $k$, then process $P_i$ may request at most $k$ instances of resource type $R_j$

3. **Allocation**: $n$ x $m$ matrix. If Allocation[$i,j$] = $k$ then $P_i$ is currently allocated $k$ instances of $R_j$

4. **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

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

worst scenario : process request max instance
⇒ still safe ?

# 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*** → not finished process $P_i$

   (b) $Need_i \leq Work$ → we can allocate $need_i$ resource right now.

   vector-wise comparison

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

3. $Work = Work + Allocation_i$

   ***Finish*[*i*] = *true***

   go to step 2

   the resources are released after termination.

   $P_i$ execution finished,

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

   else, in unsafe 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
   no holding resource
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

by safety algorithm

# 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 $T_0$:

| | *Allocation* | *Max* | *Available* | Need = Max − Allocation |
|---|---|---|---|---|
| | *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 (Cont.)

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

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

Available

A B C

3 3 2

allocate [Need], release [Need] & [Allocation]

(3, 3, 2) → (2, 1, 0) → (5, 3, 2)

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

not unique sol

Available: (3,3,2) → (5,3,2) → (7,4,3) → (7,4,5) → (10,4,7)→ (10,5,7)

# Example: $P_1$ Request (1,0,2)

Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true) granted

initial = (7,7,2) − (1,0,2) = (2,3,0)

| | Allocation | Need | Available | |
|---|---|---|---|---|
| | A B C | A B C | A B C | |
| $P_0$ | 0 1 0 | 7 4 3 | 7 5 5 | (4) |
| $P_1$ | 3 0 2 | 0 2 0 | 5 3 2 | (1) |
| $P_2$ | 3 0 2 | 6 0 0 | 10 5 7 | (5) |
| $P_3$ | 2 1 1 | 0 1 1 | 7 4 3 | (2) |
| $P_4$ | 0 0 2 | 4 3 1 | 7 4 5 | (3) |

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: $P_1$ Request (1,0,2)

Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true)

| | Allocation | Need | Available |
|---|---|---|---|
| | A B C | A B C | A B C |
| $P_0$ | 0 3 0 | 7 2 3 | 2 1 0 |
| $P_1$ | 3 0 2 | 0 2 0 | 5 3 2 |
| $P_2$ | 3 0 2 | 6 0 0 | 7 4 3 |
| $P_3$ | 2 1 1 | 0 1 1 | 7 4 5 |
| $P_4$ | 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? No.

Can request for (0,2,0) by $P_0$ be granted? No.

# Deadlock Detection

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

# Single Instance of Each Resource Type

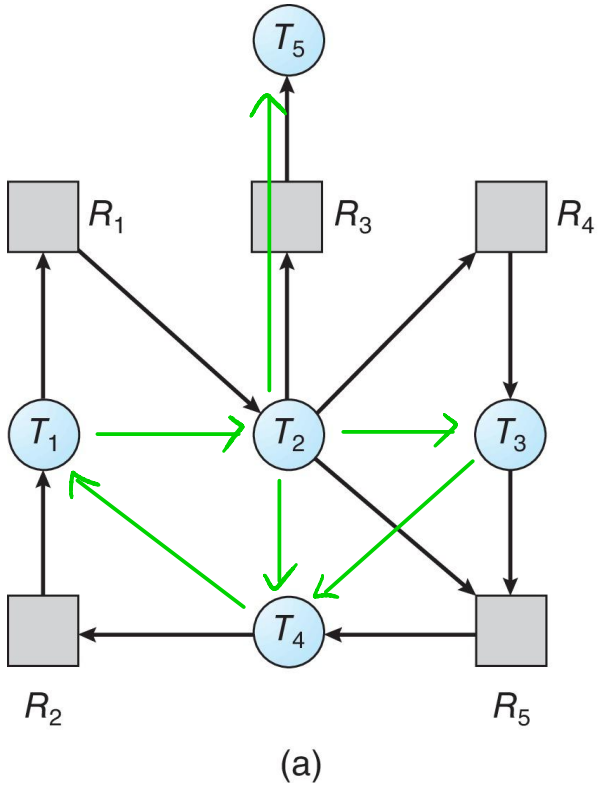
Maintain **wait-for** graph

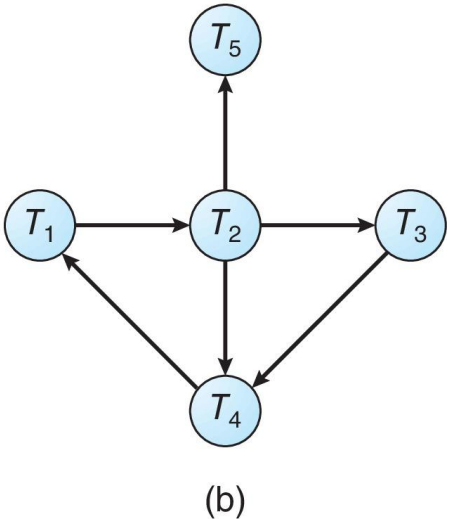
- Nodes are processes
- $P_i \rightarrow P_j$ if $P_i$ is waiting for $P_j$

Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock ⇒ then recovery!

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 $O(n^2)$

# 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 $R_j$.

No "Max" ⇒ no ability to decide grant of request like deadlock avoidance

# Detection Algorithm

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

2. Find an index ***i*** such that both: O(n) ] O(nxm)
   (a) ***Finish*[*i*] == *false***
   (b) ***Request*$_i$ ≤ *Work*** O(m)

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

# Detection Algorithm (Cont.)

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

   release resource after execution
   $O(n)$ $O(1)$ ] $O(n)$

4. If ***Finish[i] == false***, for some ***i***, $1 \le i \le n$, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then ***P***$_i$ is deadlocked

   $O(n)$

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

Step 2 x Step 3

$\Rightarrow O(n \times m) \times O(n) = O(m \cdot n^2)$

# 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 | 0 1 0 |
| $P_1$ | 2 0 0 | 2 0 2 | | 7 2 4 |
| $P_2$ | 3 0 3 | 0 0 0 | | 3 1 3 |
| $P_3$ | 2 1 1 | 1 0 0 | | 5 2 4 |
| $P_4$ | 0 0 2 | 0 0 2 | | 7 2 6 |

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

order is not unique

# 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 | $P_0$ | 0 0 0 | 0 0 0 |
| $P_1$ | 2 0 0 | $P_1$ | 2 0 2 | |
| $P_2$ | 3 0 3 | $P_2$ | 0 0 1 | |
| $P_3$ | 2 1 1 | $P_3$ | 1 0 0 | |
| $P_4$ | 0 0 2 | $P_4$ | 0 0 2 | |

0 1 0

0→1

Deadlock

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.

# 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 low ⋯ abort
2. How long process has computed, and how much longer to completion short ⋯ abort
3. Resources the process has used many ⋯ abort
4. Resources process needs to complete many ⋯ abort
5. How many processes will need to be terminated minimize
6. Is process interactive or batch? batch ⋯ abort

# Recovery from Deadlock: Resource Preemption

better

**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 ⇒ aging

# End of Chapter 8