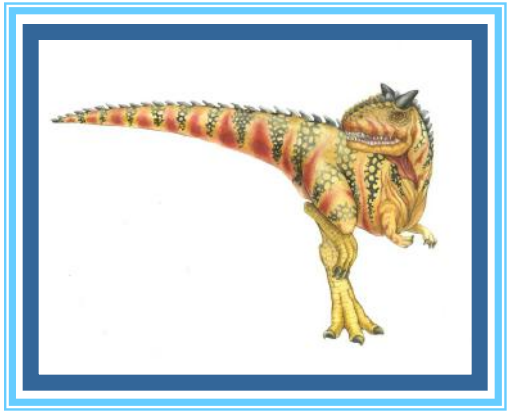
Chapter 8: Deadlocks



Outline

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

Chapter Objectives

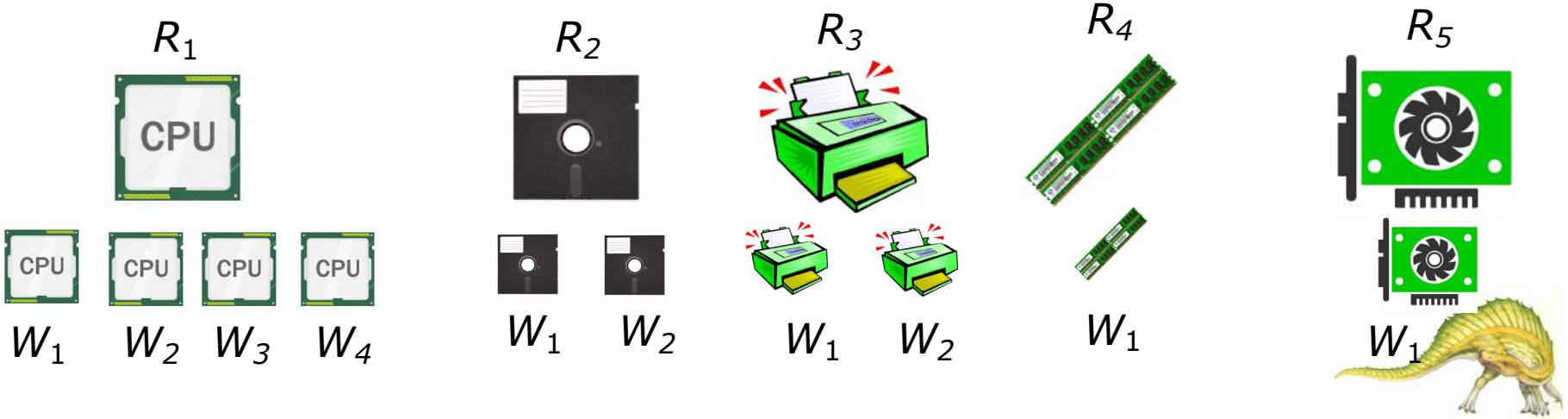
- Illustrate how deadlock can occur when mutex locks are used
- Define the four necessary conditions that characterize deadlock
- Identify a deadlock situation in a resource allocation graph
- Evaluate the four different approaches for preventing deadlocks
- Apply the banker's algorithm for deadlock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

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** Resources a process wants to use
  - **use** Resources a process is using
  - **release** Resources a process let free

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:
  - **request** Resources a process wants to use
  - **use** Resources a process is using
  - **release** Resources a process let free



Deadlock with Semaphores

- Data:
  - A semaphore s_1 initialized to 1
  - A semaphore s_2 initialized to 1
- Two threads T_1 and T_2
- T_1:

```
wait(s1)
wait(s2)
```

- T_2:

```
wait(s2)
wait(s1)
```

Deadlock with Semaphores

- Data:
  - A semaphore s_1 initialized to 1
  - A semaphore s_2 initialized to 1
- Two threads T_1 and T_2
- T_1:

  `wait(s1)`

  `wait(s2)`
- T_2:

  `wait(s2)`

  `wait(s1)`

What can happen here?

Deadlock with Semaphores

- Data:
  - A semaphore s_1 initialized to 1
  - A semaphore s_2 initialized to 1
- Two threads T_1 and T_2
- T_1:

```
wait(s1)
wait(s2)
```

- T_2:

```
wait(s2)
wait(s1)
```

What can happen here?

Option 1: Happy Ending!

Either T_1 or T_2 is very fast and obtains both locks for s_1 and s_2, and the other thread will block in the first wait.

You are lucky…
And you know it

Deadlock with Semaphores

- Data:
  - A semaphore s_1 initialized to 1
  - A semaphore s_2 initialized to 1
- Two threads T_1 and T_2
- T_1:

```
wait(s1)
wait(s2)
```

- T_2:

```
wait(s2)
wait(s1)
```

What can happen here?

Option 2: Deadlock Ending

Both T_1 and T_2 s_1 obtain the first lock, and they will be mutually blocked waiting forever, since both are waiting for the resource the other locked

You are screwed…
And you know it

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion**: only one thread at a time can use a resource
- **Hold and wait**: a thread holding at least one resource is waiting to acquire additional resources held by other threads
- **No preemption**: a resource can be released only voluntarily by the thread holding it, after that thread has completed its task
- **Circular wait**: there exists a set $\{T_0, T_1, \ldots, T_n\}$ of waiting threads such that T_0 is waiting for a resource that is held by T_1, T_1 is waiting for a resource that is held by T_2, …, T_{n-1} is waiting for a resource that is held by T_n, and T_n is waiting for a resource that is held by T_0.

Livelock

A variant of deadlock is *livelock.* It is similar to a deadlock, but instead of the processes being blocked, they are executing some action but without progressing because they cannot acquire a resource.

Data:

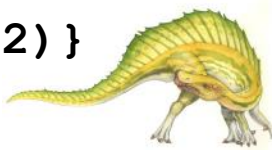
A semaphore s_1 initialized to 1

A semaphore s_2 initialized to 1

Two threads T_1 and T_2

```
T1: while (!done){
      wait(s1)
      if (try_lock(s2)){
       // do something
       unlock(&s2);
       unlock(&s1);
       done = 1;
      } else {unlock(&s1)}
    }
```

```
T1: while (!done){
      wait(s2)
      if (try_lock(s1)){
       // do something
       unlock(&s1);
       unlock(&s2);
       done = 1;
      } else {unlock(&s2)}
    }
```



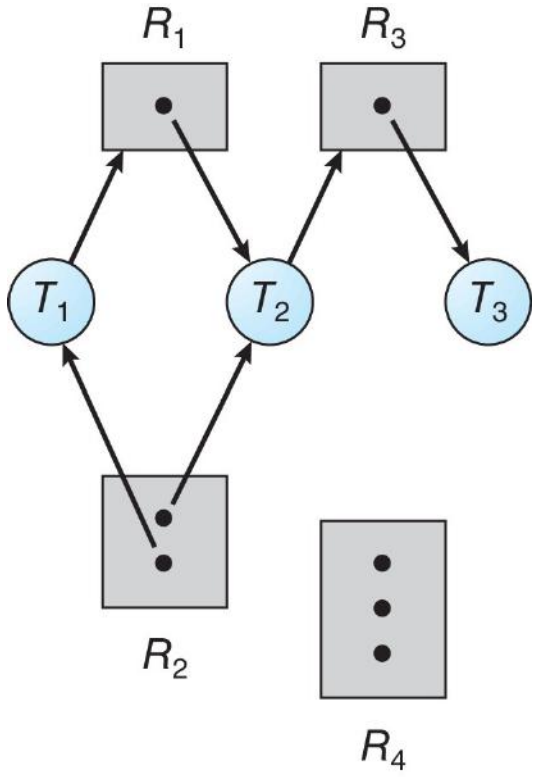
Resource-Allocation Graph

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

- V is partitioned into two types:
  - $T = \{T_1, T_2, \ldots, T_n\}$, the set consisting of all the threads 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 $T_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow T_i$

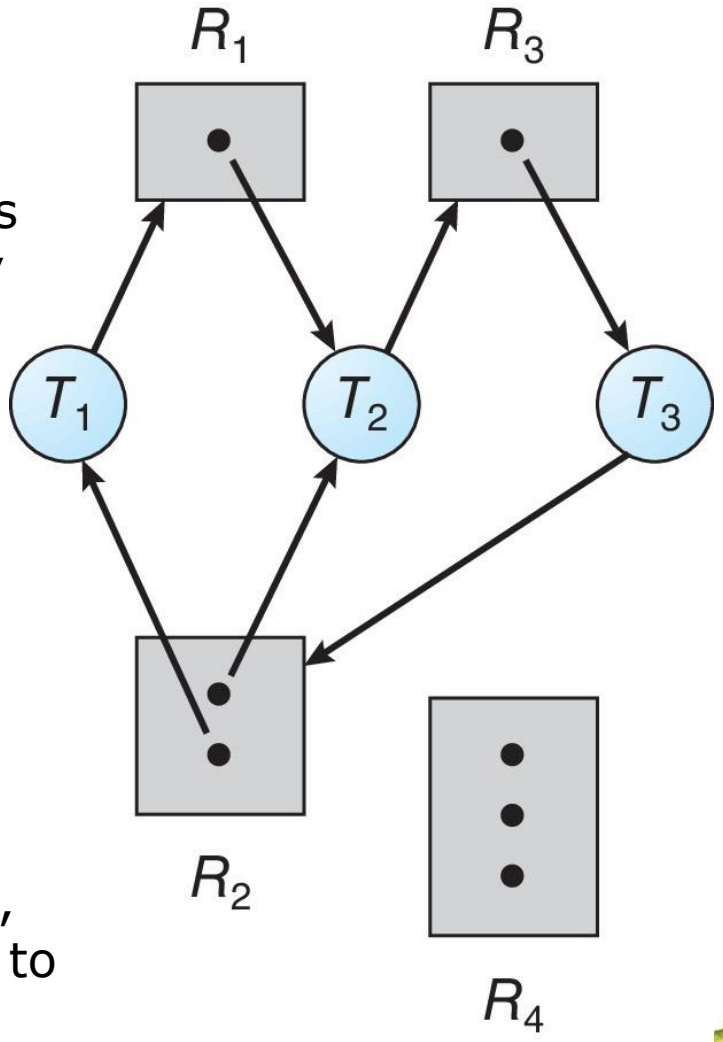
Resource Allocation Graph Example

- 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



Resource Allocation Graph with a Deadlock

In this system, T_1 is using R_2 and has requested R_1, which is being used by T_2. T_2 is also using R_2 and is waiting for R_3. Finally, T_3 is using R_3 and waiting for R_2….

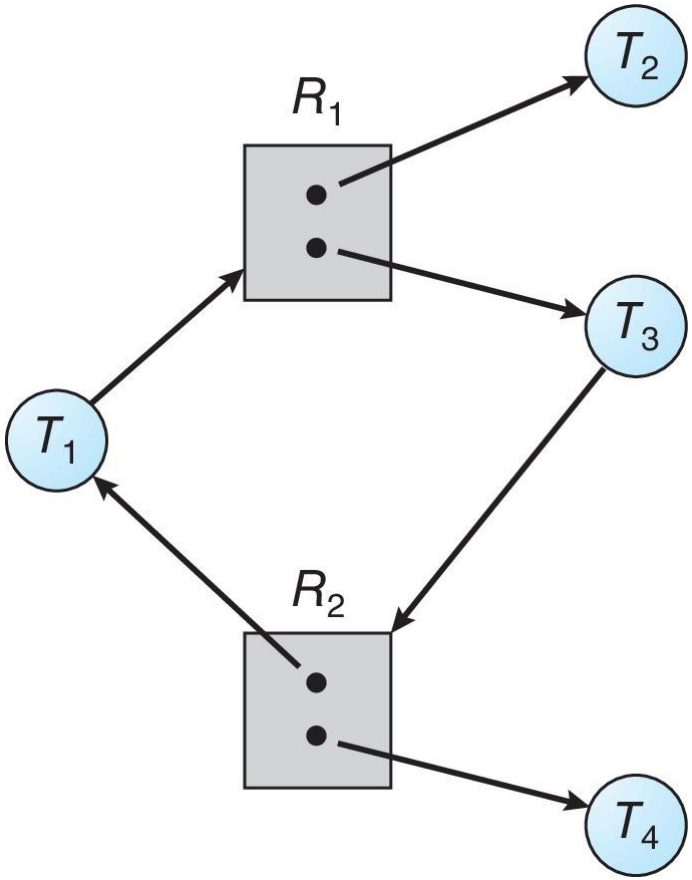


There is a loop T_1-R_1-T_2 -R_3-T_3 -R_2-T_1, and there are no resources available to satisfy all the requests (T_1 - R_1; T_2 – R_3; T_3 – R_2)

Graph with a Cycle But no Deadlock

In this other example, we can see that T_1 is waiting for R_1, which is being used by T_2 and T_3; and T_3 is waiting for R_2, which is being used by $T_1$and T_4.

There is a loop T_1-R_1-T_3 -R_2-T_1, However in this case there is no deadlock because both T_2 and T_3 can break it when they finish and they release their resources.



Exercise

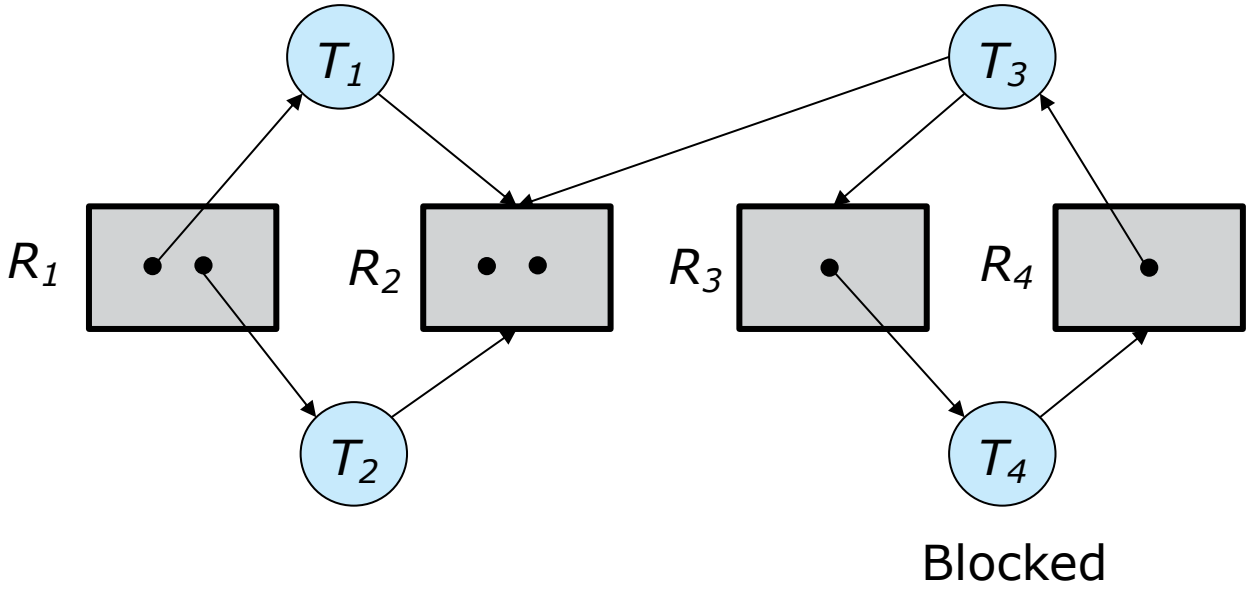
Given the following specification:

- Two instances of R_1
- Two instances of R_2
- One instance of R_3
- One instance of R_4
- T_1 holds one instance of R_1 and is waiting for an instance of R_2
- T_2 holds one instance of R_1, and is waiting for an instance of R_2
- T_3 is waiting for an instance of R_2 and R_3, and holds an instance of R_4
- T_4 is waiting for an instance of R_4 and holds an instance of R_3

Answer the following questions:

1. Is the system deadlocked?
2. Is there any blocked process?
3. Why is not deadlocked?

Solution



The system is not deadlocked. But T_3 and T_4 are blocked. The system can be slowed down depending whether T_3 obtains the lock for R_2 or not, since T_1 and T_2 will only have access to one instance of R_2

Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ignore the problem and pretend that deadlocks never occur in the system.
  - This is followed by Linux and Windows
- Allow the system to enter a deadlock state and then recover.
  - This is done by certain systems like databases
- Ensure that the system will **never** enter a deadlock state:
  - Deadlock prevention – Constrain how requests are done to avoid that the 4 deadlock conditions are satisfied simultaneously.
  - Deadlock avoidance – The system requires information on the resources that threads will request and use during their lifetime

Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

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

Deadlock Prevention (Cont.)

- **No Preemption (1)**:
  - A thread release resources when cannot acquire some others that it needs: 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 (or released) resources are added to the list of resources for which the thread is waiting
  - Thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Deadlock Prevention (Cont.)

- **No Preemption (2)**:
  - When thread T_i requests non-available resources check if they are hold by a thread *Tj* waiting for additional [non-available] resources.
  - In that case, preempt the resources from T_j *and allocate them to* T_i.
  - If the resources T_i requests are not available then T_i will wait, and its resources can be potentially preempted by another thread T_k.

Deadlock Prevention (Cont.)

- **Circular Wait:**
  - Impose a total ordering of all resource types, and require that each thread requests resources in an increasing order of enumeration
    - A total order is a set of elements (resources) together with binary relation (represented by $\leq$) indicating for each pair of elements (x,y) in the set that one element precedes (is smaller) the other.

      Who can give me some examples of total orders?

Deadlock Prevention (Cont.)

- **Circular Wait:**
  - Impose a total ordering of all resource types, and require that each thread requests resources in an increasing order of enumeration
    - A total order is a set of elements (resources) together with binary relation (represented by $\leq$) indicating for each pair of elements (x,y) in the set that one element precedes (is smaller) the other.
      - Examples of total orders are natural numbers, integers, real numbers, Booleans.

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:

```
/* 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);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

Circular Wait

- It does not work when there is dynamic allocation of resources like in the following code

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);

   acquire(lock1);
      acquire(lock2);

         withdraw(from, amount);
         deposit(to, amount);

      release(lock2);
   release(lock1);
}
```

Deadlock Avoidance

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

- Simplest and most useful model requires that each thread 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 thread 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 $<T_1, T_2, \ldots, T_n>$ of ALL the threads in the systems such that for each T_j, the resources that T_j can still request can be satisfied by currently available resources + resources held by all the T_i, with $i < j$
- That is:
  - If T_j resource needs are not immediately available, then T_j can wait until all T_i have finished
  - When T_i is finished, T_j can obtain needed resources, execute, return allocated resources, and terminate
  - When T_j terminates, T_{j+1} can obtain its needed resources, and so on

Safe State (Example)

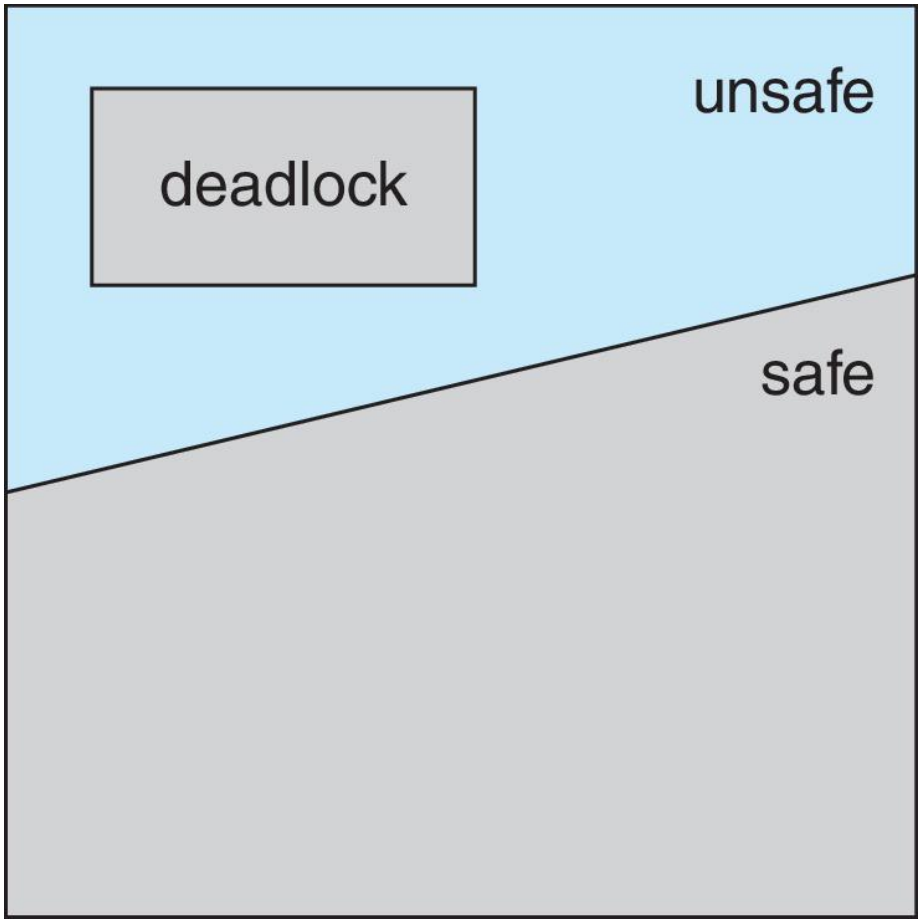
Let n = number of processes, and m = number of resources types.

- **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 T_i may request at most k instances of resource type R_j.
  - $Max[i]$ = Vector of length m indicating that T_i may request at most k instances of resource type R_j, for j = 1 *length* $Max[i]$
- **Allocation**: n x m matrix. If Allocation[i,j] = k then T_i is currently allocated k instances of R_j
  - $Allocation[i]$ = Vector of length m indicating that T_i currently allocates k instances of resource type R_j, for j = 1 *length* $Allocation[i]$
- **Need**: n x m matrix. If $Need$[i,j] = k, then T_i may need k more instances of R_j to complete its task
  - $Need[i]$ = Vector of length m indicating that T_i currently needs k instances of resource type R_j, for j = 1 *length* $Need[i]$
  - $Need$ [i,j] = Max[i,j] – $Allocation$ [i,j]

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



Safe State (Example)

- A system S is composed by 12 resources $R_1 \cdots R_{12}$ (assume they are of the same type) and 3 threads T_1, T_2, T_3 and the following resource requirements for each thread:
  - T_1 requires a maximum of 10 resources
  - T_2 requires a maximum of 4 resources
  - T_2 requires a maximum of 9 resources
- Assume that at time t_i:
  - T_1 holds 5 resources (needs 5 more resources)
  - T_2 holds 2 resources (needs 2 more resources)
  - T_3 holds 2 resources (needs 7 more resources)

  Max = [10,4,9]; Allocation=[5,2,2]; Need = [5,2,7]

  Available[1] = Available[1] for t=0 - $\sum_{i=0}^{length\ Allocation} Allocation[i] = 12 - 9 = 3$

  Is safe t_i?

Safe State (Example)

- Max = [10,4,9]; Allocation=[5,2,2]; Need = [5,2,7] Available[1]=3
- The state at time t_i is safe since we can find that the sequence T_2, T_1, T_3 that is safe:
  - T_2 is using 2 resources, it takes the 2 resources it needs from the 3 available. When it finishes it frees the 4 resources and there are a total of 5 available resources.
  - T_1 now can take the 5 available resources and add it to the resources it had locked, making the total number of necessary resources it needed, 10. When it finishes it frees the 10 resources.
  - Finally, T_3 has enough resources available for itself, and the state is safe.

Safe State (Example)

- However, if in t_{i+1} T_3 requests one resource and it is granted, then the state is not safe anymore.

Check it out!!!

Safe State (Example)

- However, if in t_{i+1} T_3 requests one resource and it is granted, then the state is not safe anymore.
- Max = [10,4,9]; Allocation=[5,2,3]; Need = [5,2,6] Available[1]=2
- Then at t_{i+1} we have:
  - T_1 holds 5 resources (needs 5 resources)
  - T_2 holds 2 resources (needs 2 resources)
  - T_3 holds 3 resources (needs 6 resources)
  - Total resources used 10
  - Total resources available 2

But no sequence guarantees a safe state.

We discard sequences starting in T_1 and T_3, since there are only 2 available resources and they need 5 and 6 resources respectively.

T_2 can allocate the 2 available resources, but after finishing there will be a total of 4 available resources, which is not enough to execute either T_1 *or* T_3 and hence the state is unsafe

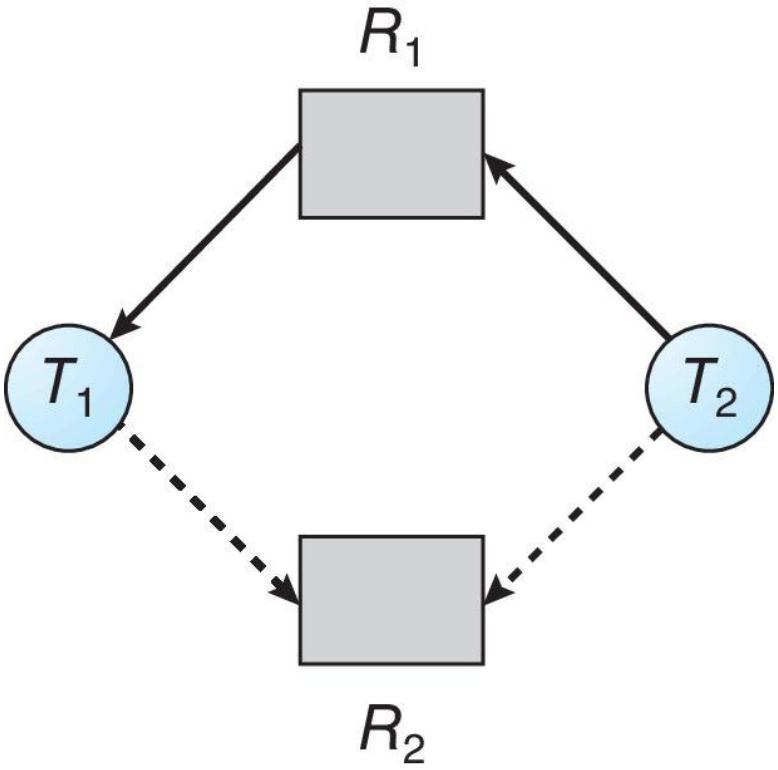
Avoidance Algorithms

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

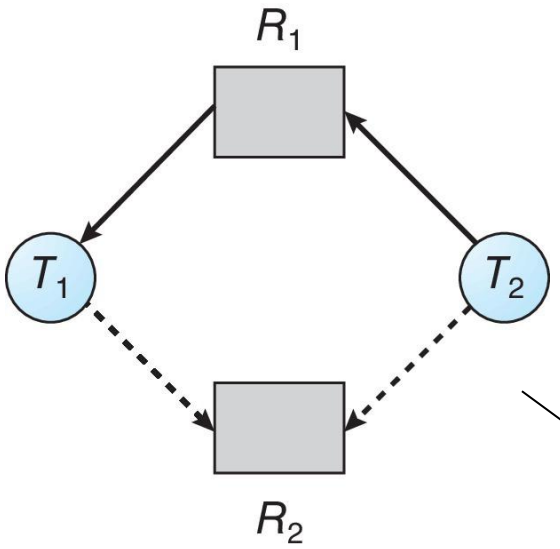
Resource-Allocation Graph



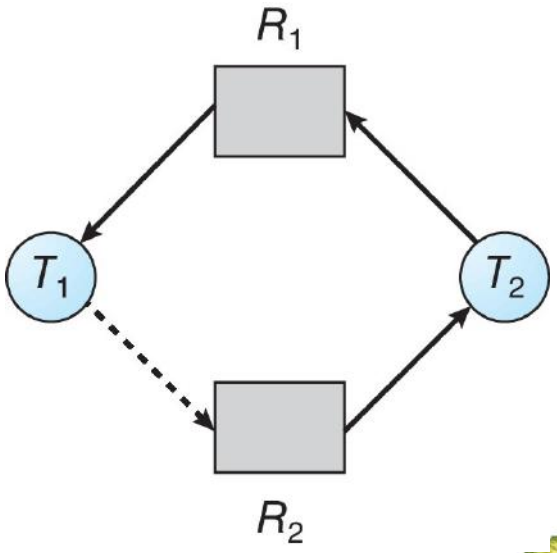
Resource-Allocation Graph Algorithm

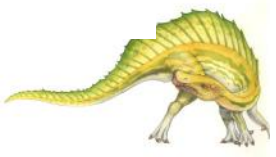
- Suppose that thread T_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph
- Checking a cycle is an operation with a complexity equals to the square of the number of vertices.

Unsafe State In Resource-Allocation Graph



T_2 requests R_2





Banker's Algorithm

- Multiple instances of resources
- Each thread must a priori claim maximum use
- When a thread requests a resource, it may have to wait
- When a thread 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 x m matrix. If Max [i,j] = k, then process T_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If Allocation[i,j] = k then T_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If $Need$[i,j] = k, then T_i may need k more instances of R_j to complete its task

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

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

The i satisfying this will be able to allocate the resources it needs.

3. $Work = Work + Allocation_i$

   Finish*[*i*] = *true

   go to step 2

This step simulates the execution of process i, releasing the resources that become ready for another process.

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

Realize that this is not the algorithm!! Why not? What happens? Exercise, based on this, construct the complete algorithm.

Resource-Request Algorithm for Process P_i

$\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{T_i}$. If $\boldsymbol{Request_i[j] = k}$ then process $\boldsymbol{T_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$

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

$$\boldsymbol{Available = Available - Request_i;}$$

$$\boldsymbol{Allocation_i = Allocation_i + Request_i;}$$

$$\boldsymbol{Need_i = Need_i - Request_i;}$$

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

Example of Banker's Algorithm

- 5 threads T_0 through T_4;

  3 resource types:

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

- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 5 3	3 3 2
T_1	2 0 0	3 2 2	
T_2	3 0 2	9 0 2	
T_3	2 1 1	2 2 2	
T_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
T_0	7 4 3
T_1	1 2 2
T_2	6 0 0
T_3	0 1 1
T_4	4 3 1

- The system is in a safe state since the sequence $< T_1, T_3, T_4, T_2, T_0>$ satisfies safety criteria

Example: P_1 Request (1,0,2)

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

Updated state:

	Allocation	Need	Available
	A B C	A B C	A B C
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < T_1, T_3, T_4, T_0, T_2> satisfies safety requirement
- Can request for (3,3,0) by T_4 be granted?
- Can request for (0,2,0) by T_0 be granted?

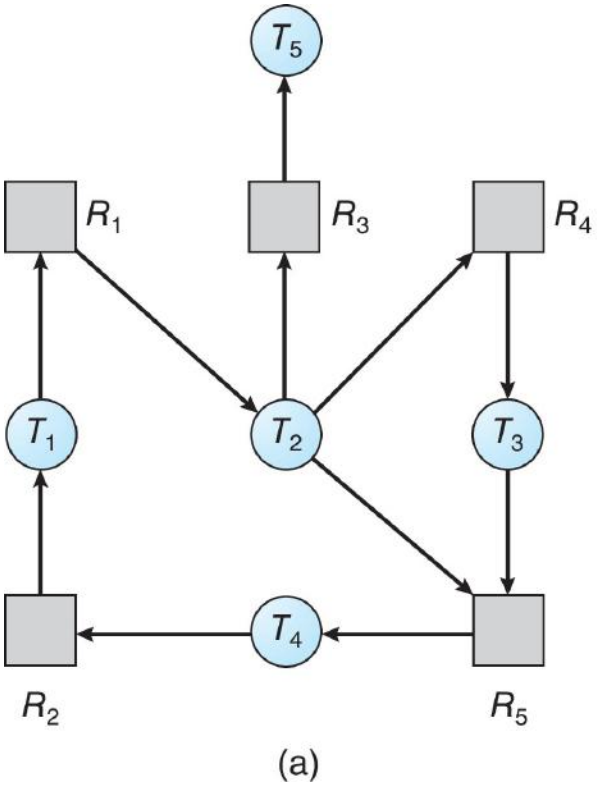
Deadlock Detection

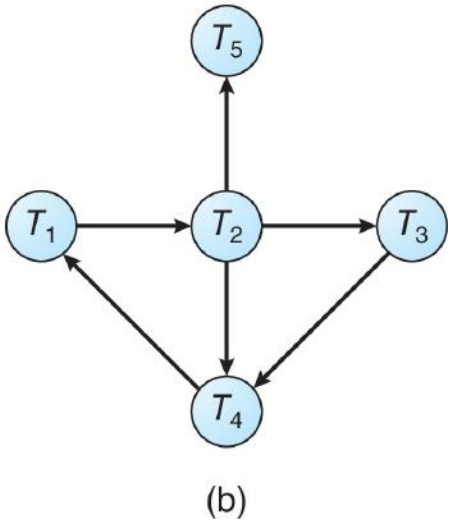
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are threads
  - $T_i \rightarrow T_j$ if T_i is waiting for T_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 thread.
- **Request**: An ***n*** **x** ***m*** matrix indicates the current request of each thread. If ***Request*** **[*i*][*j*] = *k***, then thread T_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 ***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
3. ***Work = Work + Allocation*$_i$**
   Finish*[*i*] = *true
   go to step 2
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 T_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 threads T_0 through T_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*
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 2	
T_2	3 0 3	0 0 0	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

- Sequence <T_0, T_2, T_3, T_1, T_4> will result in ***Finish[i] = true*** for all ***i***

Example (Cont.)

- T_2 requests an additional instance of type *C*

	Request
	A B C
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

- State of system?
  - Can reclaim resources held by thread T_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes T_1, T_2, T_3, and T_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 threads “caused” the deadlock.

Recovery from Deadlock: Process Termination

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

Recovery from Deadlock: Resource Preemption

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

End of Chapter 8