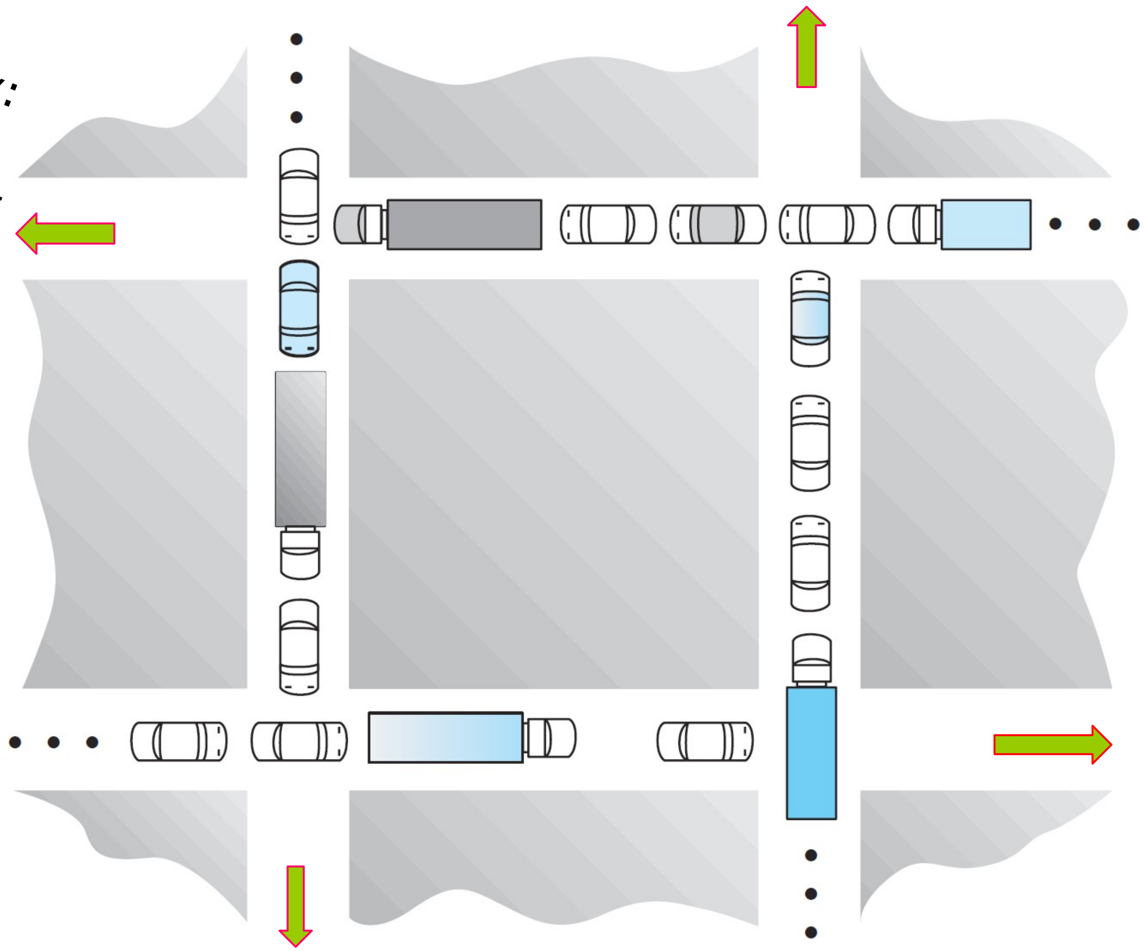
Chapter 8: Deadlocks

Traffic deadlock

Analogy:

Threads – Vehicles

Resources – roads



Deadlocked:

Every vehicle at the junction is waiting for the other to move, which is also blocked

Chapter Objectives

1. System Model
2. Deadlock in Multithreaded Applications
3. Deadlock Characterization
4. Illustrate how deadlock can occur when mutex locks are used
5. Define the four necessary conditions that characterize deadlock
6. Identify a deadlock situation in a resource allocation graph
7. Learn the methods for handling Deadlocks

Methods for handling Deadlocks:

1. Evaluate the four different approaches for **preventing deadlocks**
2. Apply the banker's algorithm for **deadlock avoidance**
3. Apply the deadlock **detection algorithm**
4. Evaluate approaches for **recovering from deadlock**

Deadlocks happen

Deadlocks happen in **multiprogramming** environment when **several threads** compete for a **finite number of resources**.

A set of threads is in a deadlocked state *when every thread in the set is waiting for an event that can be caused only by another thread in the set.* The events with which we are mainly concerned here are resource acquisition and release.

A thread requests resources; if the resources are not available at that time, the process enters a waiting state.

- Sometimes, a waiting thread is never again able to change state, because the resources it has requested are held by other waiting threads.
- This situation is called a **deadlock**.

System Resource Classification

System resources may be partitioned into several types (or **classes**)

Classification:

1. Each class consisting of some number of **identical instances**
2. If a thread requests an instance of a resource type, the allocation of ***any*** **instance** of the type should satisfy the request.
3. For example, a system may have two printers.
    1. These two printers may be defined to be in the same resource class if no one cares which printer prints which output.
4. If not, then the instances are not identical, and add another class for it.

Resource Classification

Each class consists of some number of identical instances.

- CPU cycles –
  - one or more instances: CPU Instance 1, CPU Instance 2, etc.
- Memory
  - one or more instances: Memory instance 1, Memory instance 2, etc.
- Semaphores
  - zero or more instances: Semaphore instance 1, instance 2, etc.
- Mutex locks
  - zero or more instances: Mutex locks instance 1, instance 2, etc.
- Files
  - zero or more instances: File instance 1, File instance 2 etc.
- I/O devices
  - zero or more instances:Printer Instance 1, Printer instance 2, etc.

System Model

- A thread may request as many resources as it requires to carry out its designated task.
- A thread must request a resource before using it and must release the resource after using it:

1. **Request**. The thread requests the resource.
   - If the request cannot be granted immediately (for example, if the resource is being used by another thread), then the requesting thread must wait until it can acquire the resource.
2. **Use**.
   - The thread can operate on the resource (for example, if the resource is a printer, the thread can print on the printer).
3. **Release**.
   - The thread releases the resource.

System Calls for resource request and release

- device
  - request() and release()
- file
  - open() and close()
- memory
  - allocate() and free()
- semaphores
  - wait() and signal()
- mutex lock
  - acquire() and release()

Deadlock in Multithreaded Applications

- In Figure 8.1 Deadlock Example, `thread_one` attempts to acquire the mutex locks in the order (1) `first_mutex`, (2) `second_mutex`.
- At the same time, `thread_two` attempts to acquire the mutex locks in the order (1) `second_mutex`, (2) `first_mutex`.
- Deadlock is possible if `thread_one` acquires `first_mutex` while `thread_two` acquires `second_mutex`.
- Deadlock will not occur if `thread_one` can acquire and release the mutex locks for `first_mutex` and `second_mutex` before `thread_two` attempts to acquire the locks.
- And, of course, the order in which the threads run depends on how they are scheduled by the CPU scheduler.

Livelock

Livelock is another form of liveness failure.

- Livelock is similar to deadlock. It is less common than deadlock
- Both prevent two or more threads from proceeding, but the threads are unable to proceed for different reasons.
- Deadlock occurs when every thread in a set is blocked waiting for an event that can be caused only by another thread in the set.
- Llivelock occurs when a thread continuously attempts an action that fails. They aren't blocked, but they aren't making any progress.
- Livelock typically occurs when threads retry failing operations at the same time.
- It thus can generally be avoided by having each thread retry the failing operation at random times.
- This is precisely the approach taken by Ethernet networks when a network collision occurs.

Livelock example

- Livelock can be illustrated with the Pthreads `pthread_mutex_trylock()` function, which attempts to acquire a mutex lock without blocking.
- The code example in Figure 8.2 rewrites the example from Figure 8.1 so that it now uses `pthread_mutex_trylock()`.
- This situation can lead to livelock if thread one acquires first mutex, followed by thread two acquiring second mutex.
- Each thread then invokes `pthread_mutex_trylock()`, which fails, releases their respective locks, and repeats the same actions indefinitely.

Deadlock Livelock

```
/* 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);
}
```

Figure 8.1 Deadlock example.

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   int done = 0;

   while (!done) {
      pthread_mutex_lock(&first_mutex);
      if (pthread_mutex_trylock(&second_mutex)) {
         /**
          * Do some work
          */
         pthread_mutex_unlock(&second_mutex);
         pthread_mutex_unlock(&first_mutex);
         done = 1;
      }
      else
         pthread_mutex_unlock(&first_mutex);
   }

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   int done = 0;

   while (!done) {
      pthread_mutex_lock(&second_mutex);
      if (pthread_mutex_trylock(&first_mutex)) {
         /**
          * Do some work
          */
         pthread_mutex_unlock(&first_mutex);
         pthread_mutex_unlock(&second_mutex);
         done = 1;
      }
      else
         pthread_mutex_unlock(&second_mutex);
   }

   pthread_exit(0);
}
```

Figure 8.2 Livelock example.

Necessary Conditions

Deadlock can arise if **four conditions** **hold <u>simultaneously.</u>**

- **Mutual exclusion:** At least one resource must be held in a nonsharable mode; that is, only one thread at a time can use a resource. If another thread requests that resource, the requesting thread must be delayed until the resource has been released.

1. **Hold and wait:** A thread must be holding at least one resource and waiting to acquire additional resources currently held by other threads
2. **No preemption:** a resource can be released only voluntarily by the thread holding it, after that thread has completed its task
3. **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.

✓ **all four conditions must hold for a deadlock to occur**.

Resource-Allocation Graph

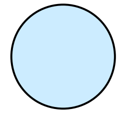
Deadlocks can be described more precisely in terms of a directed graph called a **system** **resource-allocation graph**.

This graph consists of a set of **vertices *V*** and a set of edges *E*.

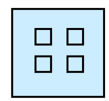
- V is partitioned into two types:
  - $T = \{T_1, T_2, \ldots, T_n\}$, the set consisting of all the threads in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $T_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow T_i$

Resource-Allocation Graph (Cont.)

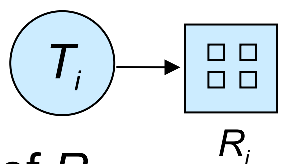
- Thread



- Resource Type with 4 instances

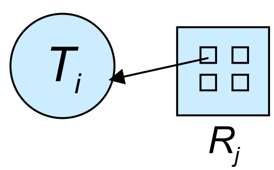


- T_i requests instance of R_j



When a thread *Ti* requests an instance of resource type *Rj* a request edge is inserted in the graph

- T_i is holding an instance of R_j



When request is granted, the request edge is transformed to an assignment edge.

When the process no longer needs the resource, it releases the resource and the assignment edge is deleted.

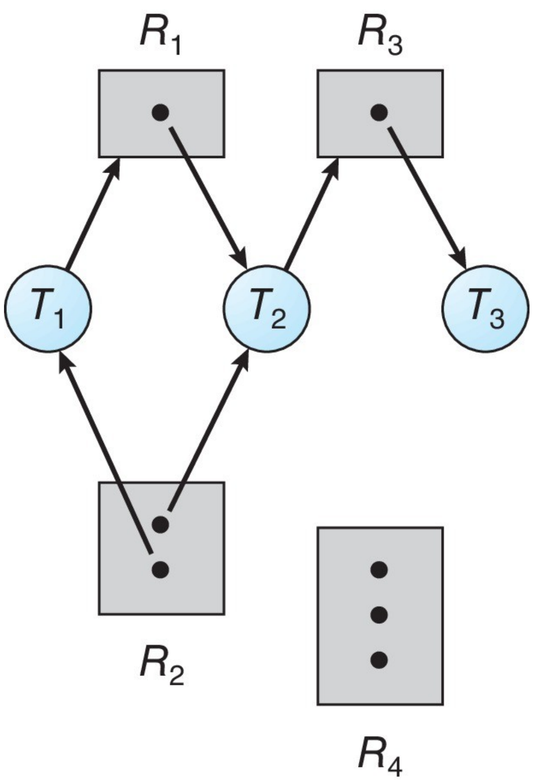
Resource Allocation Graph Example

$T = \{T1, T2, T3\}$

$R = \{R1, R2, R3, R4\}$

$E = \{T1 \rightarrow R1, T2 \rightarrow R3, R1 \rightarrow T2, R2 \rightarrow T2, R2 \rightarrow T1, R3 \rightarrow T3\}$

- Resource instances:
  - One instance of R1
  - Two instances of R2
  - One instance of R3
  - Three instance of R4
- Thread states:
  - 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



Cycle in a resource allocation graph

1. NO DEADLOCK:

- If the graph contains **no cycles, then no thread in the system deadlocked.**

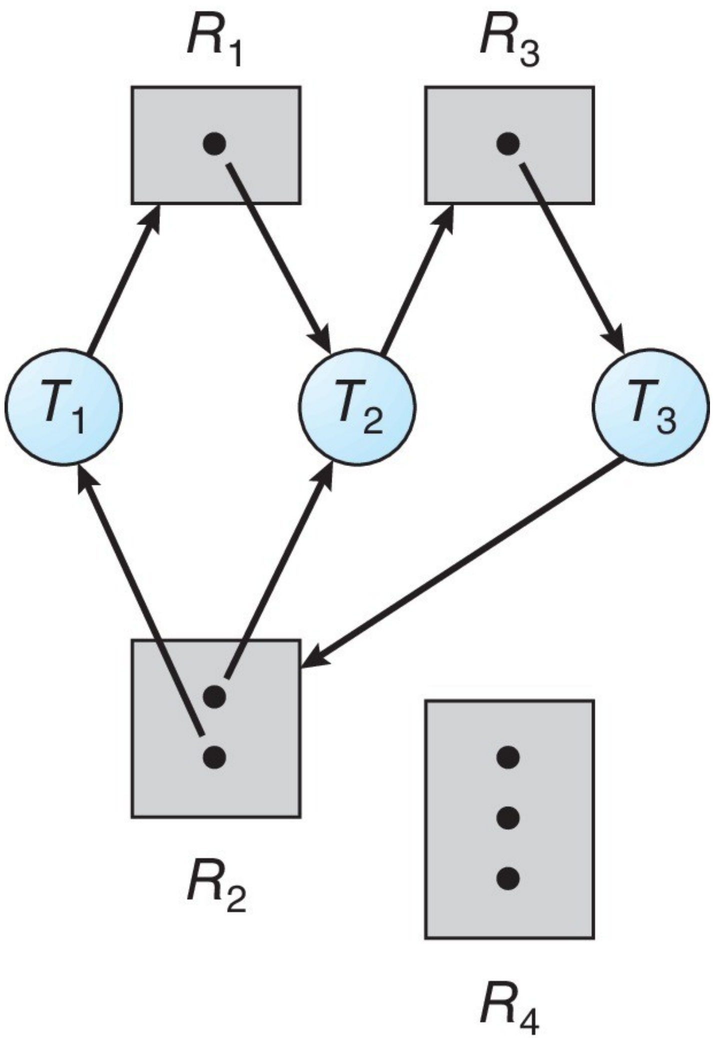
2. DEADLOCK:

- If the graph **contains a cycle, and** if the cycle involves only a set of **resource types, each** of which has **only a single instance**, then e**ach thread involved in the cycle is deadlocked**.

3. NOT NECESSARILY:

- If **each resource** type has **several instances, then a cycle does not necessarily** imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

Resource Allocation Graph With A Deadlock



2 cycles :

R2 → T1 → R1 → T2 → R3 → T3 → R2

R2 → T2 → R3 → T3 → R2

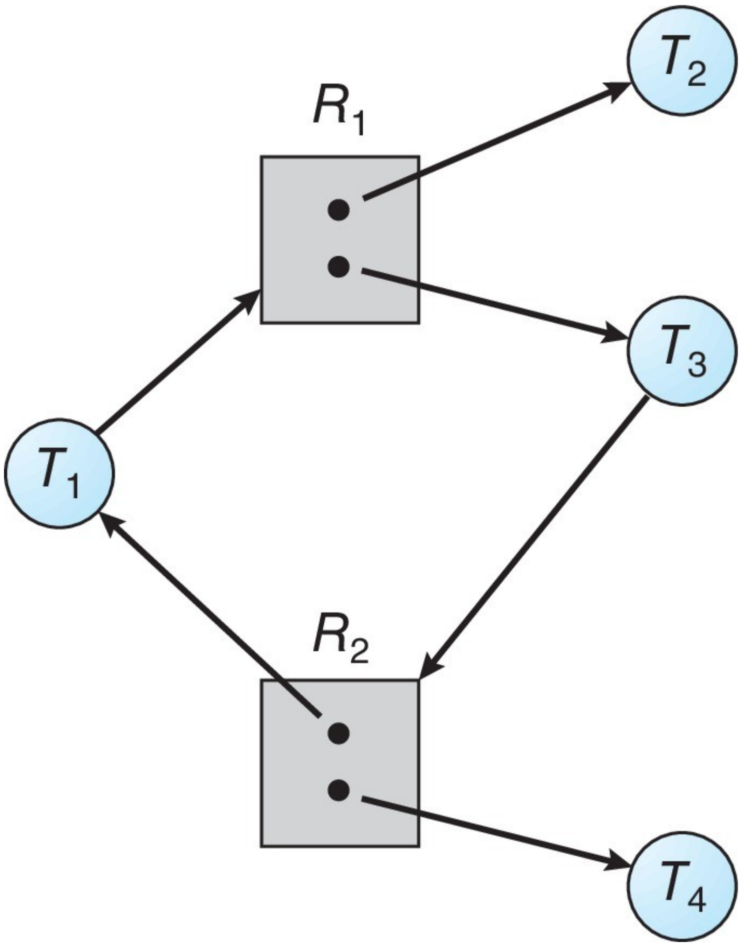
Threads *T*1, *T*2, and *T*3 are deadlocked.

T2 is waiting for the resource *R*3, which is held by *T*3.

T3 is waiting for either thread *T*1 or thread T2 to release resource *R*2.

In addition, T1 is waiting for T2 to release resource *R*1.

Graph With A Cycle But No Deadlock



we also have a cycle:
$T1 \rightarrow R1 \rightarrow T3 \rightarrow R2 \rightarrow T1$

There is no deadlock.

Thread $T4$ may release its instance of resource type $R2$.

That resource can then be allocated to $T3$, breaking the cycle.

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

- 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
  - solution used by most operating systems, including Linux and Windows.
  - It is then up to the application developer to write programs that handle deadlocks.
- Ignore the problem and pretend that deadlocks never occur in the system.

1. 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
  - **cannot prevent deadlocks by denying the mutual-exclusion** condition, because some resources are intrinsically nonsharable.

- **Hold and Wait** – must guarantee that whenever a thread **requests a resource, it does not hold** any other resources
  - Require a thread 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** –
  - If a thread 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 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
- **Circular Wait** – impose a total ordering of all resource types, and require that each thread 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 follow the increasing order:

Request first_mutex,

then second_mutex

```
/* 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);
}
```

2. 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 threads

Safe State

A state is *safe* if the system can allocate resources to each thread (up to its maximum) in some order and still avoid a deadlock.

A system is in a safe state only if there exists a **safe sequence**.

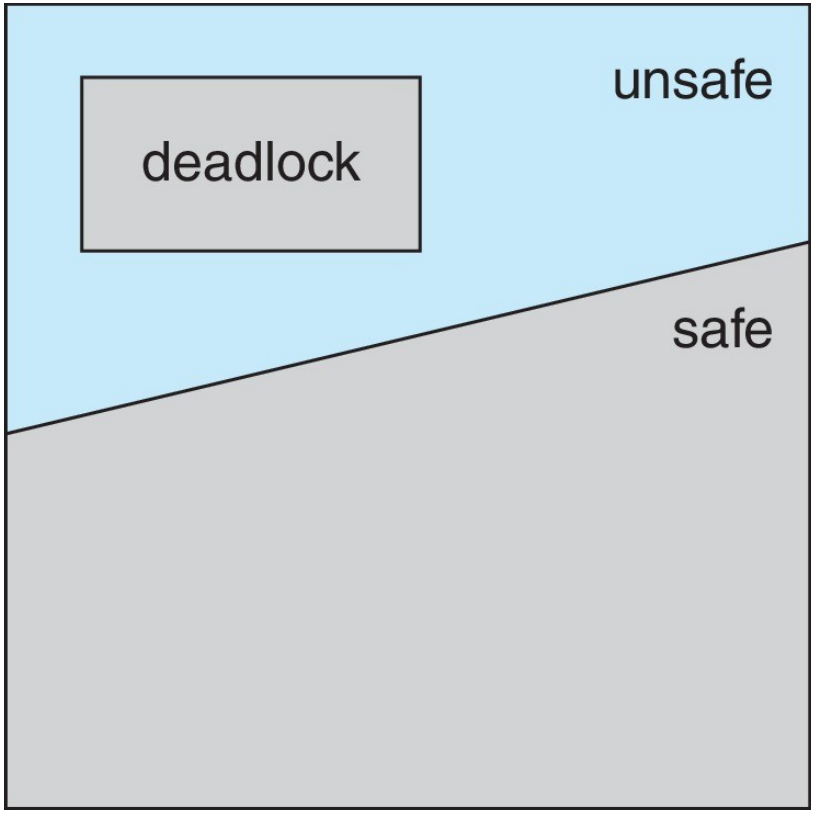
A sequence of threads <*T*1, *T*2, ..., *Tn*> is a safe sequence for the current allocation state if, for each *Ti*, the resource requests that *Ti* can still make, can be satisfied by the currently available resources plus the resources held by all *Tj*, with $j < i$.

That is:

- If the resources that *Ti* needs are not immediately available, then *Ti* can wait until all *Tj* have finished.
- When they have finished, *Ti* can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate.
- When *Ti* terminates, *Ti*+1 can obtain its needed resources, and so on.
- If no such sequence exists, then the system state is said to be *unsafe.*

Safe, Unsafe, Deadlock State

- A safe state is not a deadlocked state.
- Conversely, a deadlocked state is an unsafe state.
- Not all unsafe states are deadlocks
- An unsafe state ***may*** lead to a deadlock.
- As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states.
- In an unsafe state, the operating system cannot prevent threads from requesting resources in such away that a deadlock occurs. The behavior of the threads controls unsafe states.



Safe State illustration

Consider a system with twelve resources and three threads *T*0, *T*1, and *T*2.

Thread *T*0 requires ten resources, thread *T*1 may need four, and thread *T*2 may need nine resources.

Suppose that, at time t_0, thread *T*0 is holding five resources, thread *T*1 is holding two resources, and thread *T*2 is holding two resources. (Thus, there are three free resources.).

	Maximum Needs	*Allocation*	*Need*
T_0	10	5	5
T_1	4	2	2
T_2	9	2	7

At time t_0, the system is in a safe state. The sequence <*T*1, *T*0, *T*2> satisfies the safety condition.

- Thread *T*1 can immediately be allocated all its resources and then return them (the system will then have five available resources);
- Then thread *T*0 can get all its resources and return them (the system will then have ten available resources);
- Finally thread *T*2 can get all its resources and return them (the system will then have all twelve resources available).

Basic Facts

- A system can go froma safe state to an unsafe state.
- Suppose that, at time $t1$, thread $T2$ requests and is allocated one more resource. Only thread $T1$ can be allocated all its resources. When it returns them, the system will have only four available resources. The system is no longer in a safe 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.

Avoidance Algorithms

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

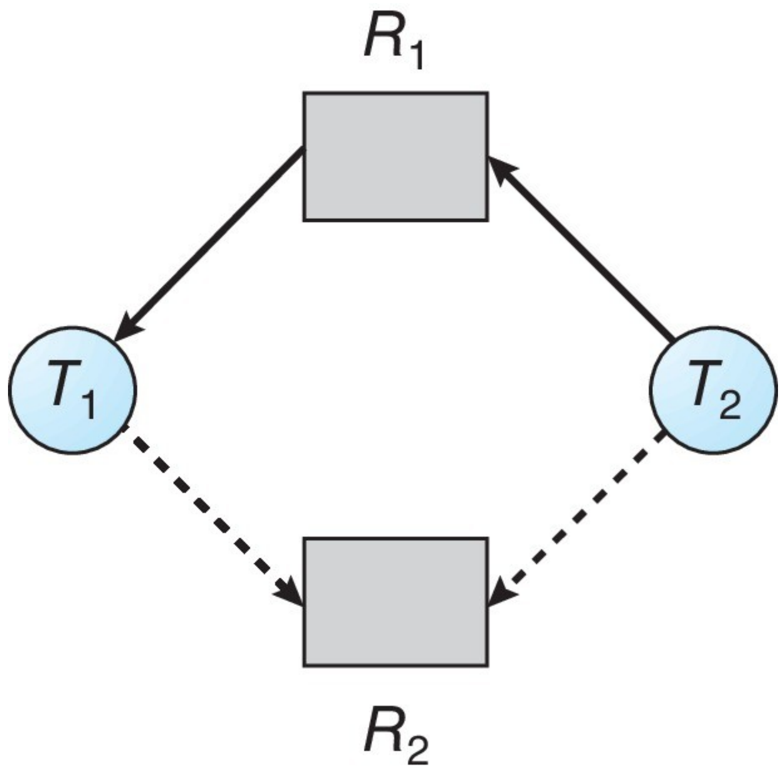
- Multiple instances of a resource type
  - Use the Banker's Algorithm

Resource-Allocation Graph Scheme

In addition to the request and assignment edges, we introduce a new type of edge, called a **claim edge,** represented by a dashed line

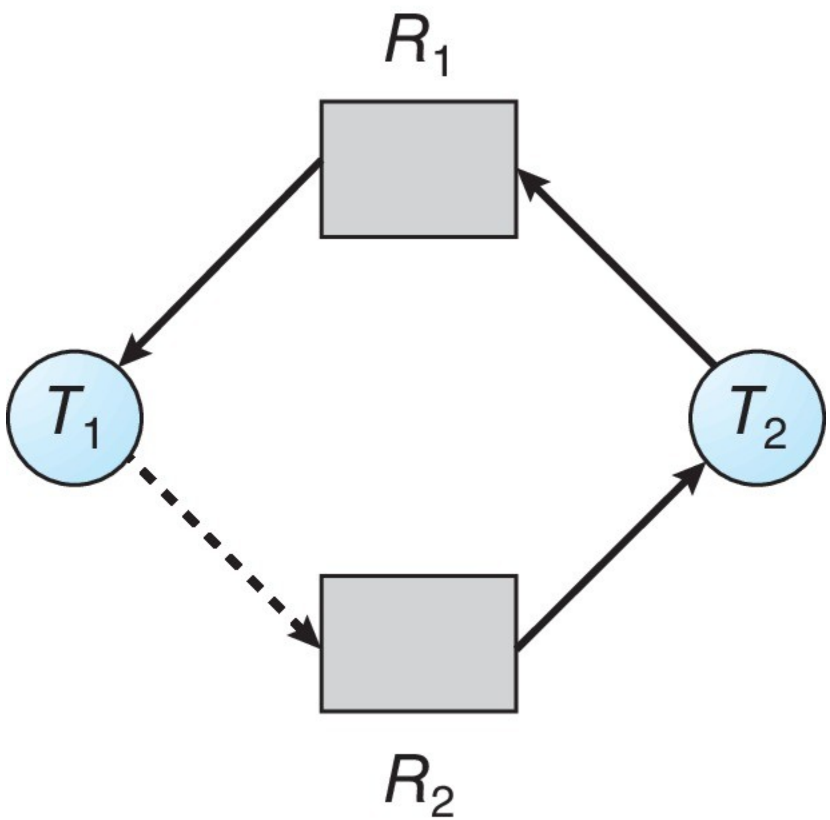
- **Claim edge** $T_i \rightarrow R_j$ indicated that thread T_j may request resource R_j at some time in the future.
- Claim edge converts to request edge when a thread requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the thread
- When a resource is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

Resource-Allocation Graph for Deadlock Avoidance



Now suppose that *Ti* requests resource *Rj*. The request can be granted only if converting the request edge *Ti* → *Rj* to an assignment edge *Rj* → *Ti* does not result in the formation of a cycle in the graph

Unsafe State In Resource-Allocation Graph



Although *R*2 is currently free, we cannot allocate it to *T*2, since this action will create a cycle in the graph. A cycle indicates that the system is in an unsafe state.

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

Banker's Algorithm

- Multiple instances of resources
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

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

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

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

Example of Banker's Algorithm

- 5 processes P_0 through P_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
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

Need is Max – Allocation

7-0 = 7 5-1=4 3-0=3
3-2 = 1 2-0=2 2-0=2
9-3 = 6 0-0=0 2-2=0
2-2 = 0 2-2=1 2-1=1
4-0 = 4 3-0=3 3-2=1

Example (Cont.)

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

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

- The system is in a safe state since 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 < $\boldsymbol{P_1}$, $\boldsymbol{P_3}$, $\boldsymbol{P_4}$, $\boldsymbol{P_0}$, $\boldsymbol{P_2}$> satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
  - No. Resources are not available
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?
  - No. Resources available, but will result in unsafe state

3. Deadlock Detection

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

Deadlock detection algorithm – Single Instance

- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j

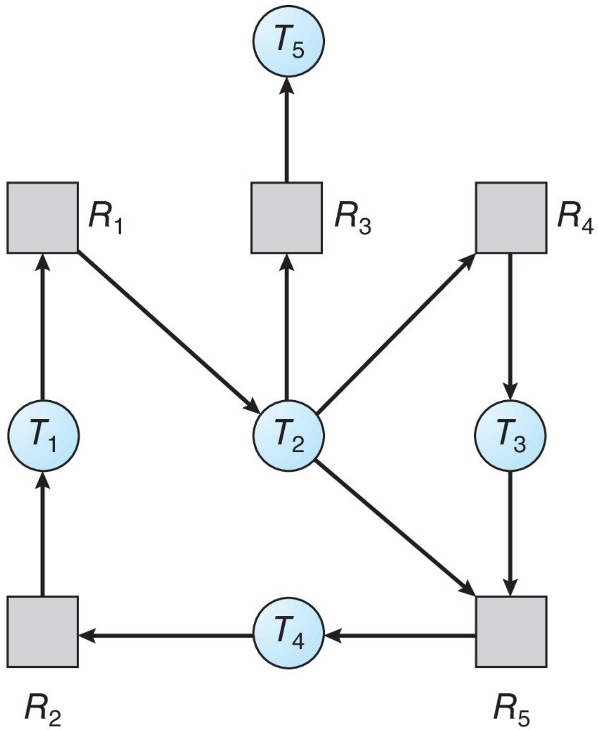
- Periodically invoke the algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

If all resources have only a single instance, define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for** graph.

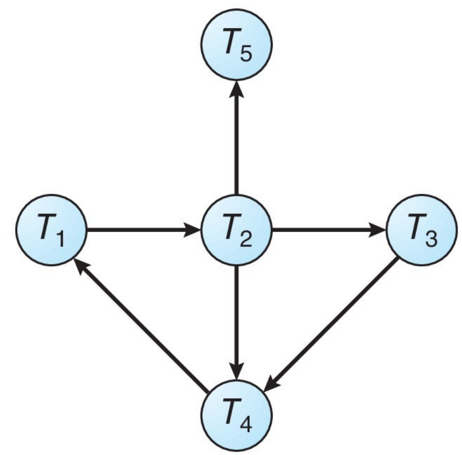
We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

- More precisely, an edge from *Pi* to *Pj* in a wait-for graph implies implies that process *Pi* is waiting for process *Pj* to release a resource that *Pi* needs.
- (An edge *Pi* → *Pj* exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges *Pi* → *Rq* and *Rq* → *Pj* for some resource *Rq)*

Resource-Allocation Graph and Wait-for Graph



(a)



(b)

Resource-Allocation Graph	Corresponding wait-for graph

Deadlock detection algorithm - Several Instances of a Resource Type

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

Detection Algorithm

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

   (a) ***Work = Available***

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

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

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

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

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

Detection Algorithm (Cont.)

3. $\boldsymbol{Work = Work + Allocation_i}$
   Finish*[*i*] = *true
   go to step 2

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

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

Example of Detection Algorithm

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

- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

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

Example (Cont.)

- P_2 requests an additional instance of type ***C***

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

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

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

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?

4. 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