Computer and Operating Systems (COS)

Lecture 10

Overview of the contents

- Deadlock characterization
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection
- Recovery from deadlock

Deadlocks

In a multi-threaded multi-programming environment, several threads fight for a limited number of resources.

A thread requests resources. If the resources are not available at that time, the thread enters a waiting state. Sometimes, *a waiting thread can never again change state, because the resources it has requested are held by other waiting threads*. This situation is called a **deadlock**.

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

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

Hold and wait: a thread holds at least one resource while waiting for additional resources held by other threads.

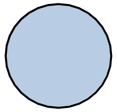
No preemption: a resource is only released voluntarily by the thread that holds it when it has completed its task.

Circular wait: if there exists a set {T0, T1,…, Tn} of waiting threads such that T0 waits for a resource held by T1, T1 waits for a resource held by T2,…, Tn – 1 waits for a resource held by Tn and Tn waiting for a resource held by T0.

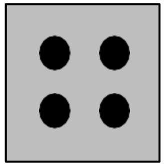
Resource allocation graph

Symbol explanation

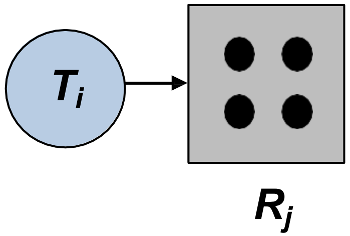
A Thread



A Resource (with 4 instances)

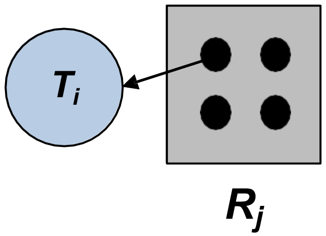


T_i Requests an instance of R_j



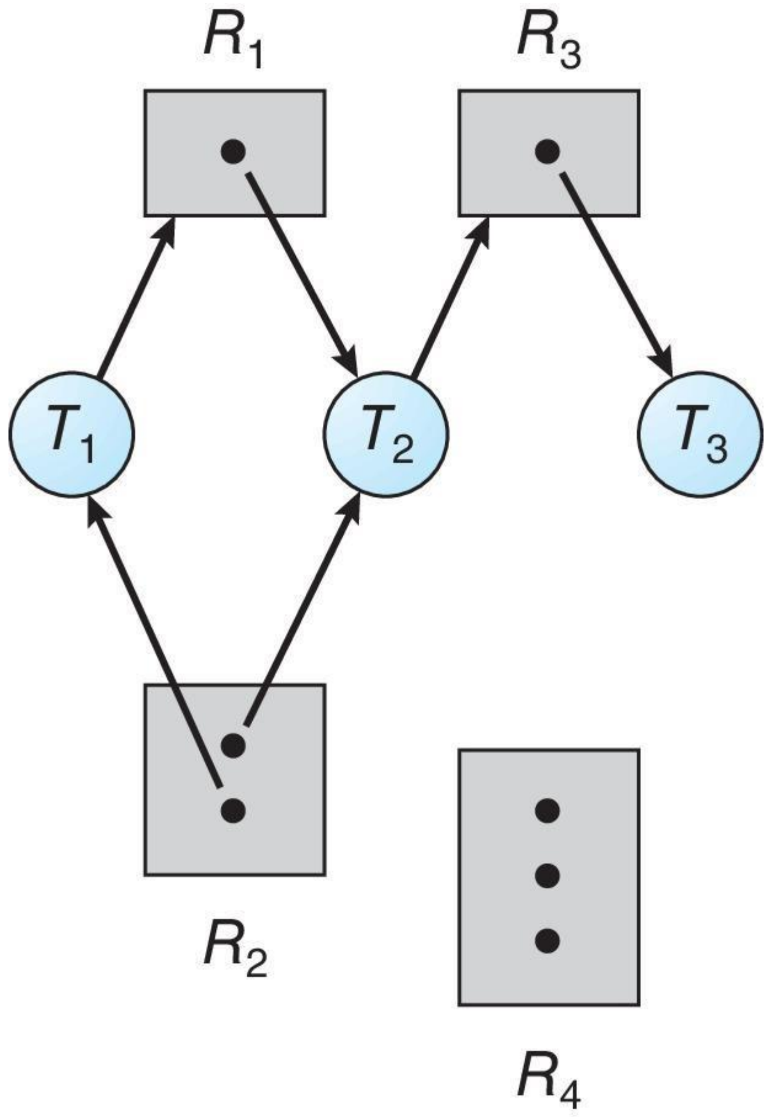
Request edge

T_i holds an instance of R_j

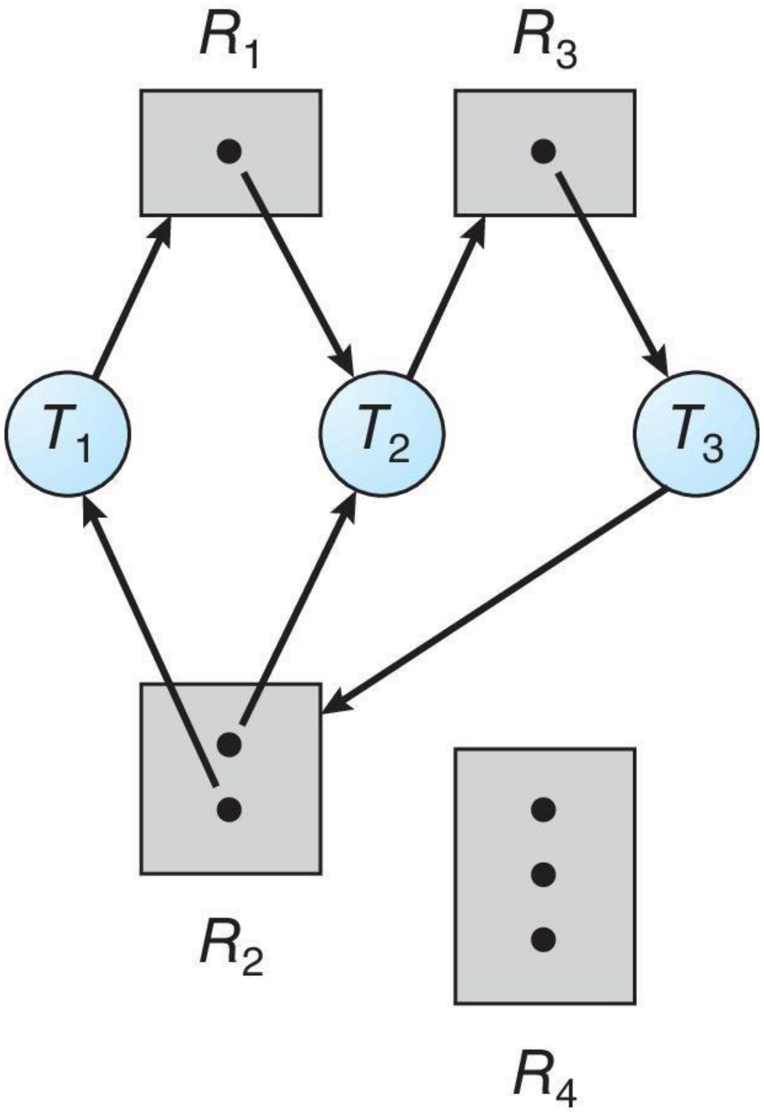


Assignment edge

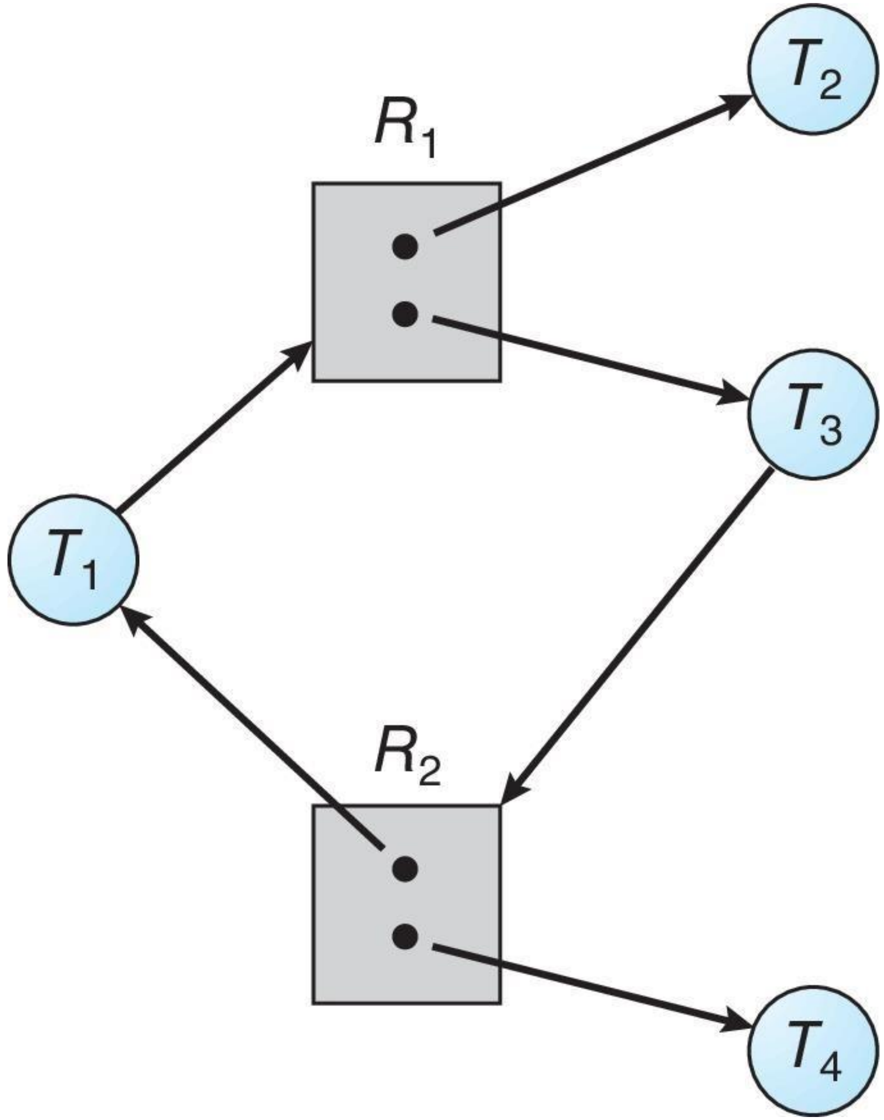
Resource allocation graph



Resource allocation graph



Resource allocation graph



Resource allocation graph

Summary

If a graph does NOT contain cycles → No deadlock

If a graph contains cycles → possibly cause deadlock

- But ONLY if there is one instance per. resource type
- If there are several instances per. Resource type, there is a risk

Strategies for managing deadlocks

- Ignore the problem and pretend that it never occurred…
  e.g.: Unix, Linux, Windows and more… It is then up to kernel and application developers to write programs that handle deadlocks (typically using approaches in the second solution)

- Use a protocol to make sure deadlock **never** occurs…
  i.e., **deadlock prevention and avoidance**

- Allow deadlock to occur and then detect it and recover afterwards… i.e., **deadlock detection and recovery**
  e.g.: databases

Deadlock prevention

For a deadlock to occur, the four conditions must be met.

If we can ensure that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

We elaborate on this approach by examining each of the four necessary conditions separately.

Deadlock prevention

Mutual Exclusion

If we want to break this condition. Then all resources must be shareable …

It is unrealistic to think that such a system exists…

E.g.: mutex, semaphore, CPU time, etc.

Deadlock prevention

Hold and Wait

A thread can only start program execution when all the resources it needs to perform its work can be allocated.

E.g. retrieve data from DVD → save it in a file → read-only file → print

Protocol 1:

allocate DVD, file, printer → perform work→ release resources

protocol 2:

allocate DVD, file → copy → release resources
allocate file → sort → release resources
allocate file, printer → print → release resources

… Poor utilization of resources and possibility of starvation

Deadlock prevention

No Preemption

A thread that holds resources and requests another resource that cannot be immediately allocated to it, must implicitly release (devote) all its allocated resources.

Alternative

A thread that holds resources and requests new resources.

- if they are available, they are taken.
- if they are allocated to a waiting thread, they are taken.
- if they are not available and allocated to a running thread, the thread has to wait, and all the allocated resources are preempted

A thread can only continue once it has received all the resources it has been deprived of (preempted), as well as those it lacked.

… Is often used in connection with resources whose state can be easily saved.

Deadlock prevention

Circular Wait

Each resource type in the system is assigned a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.

the rule is that all threads only request resources in ascending number order.

e.g.

R1→R3→R27→R33

if:

R1→R3→R27→R33→ next is R5. Release R27 and R33 and start over…

R1→R3→R5→R27→R33

… There is no guarantee. It requires the threads to comply with the rule

Deadlock Avoidance

This strategy requires that the system receive some additional information about the threads before they are executed.

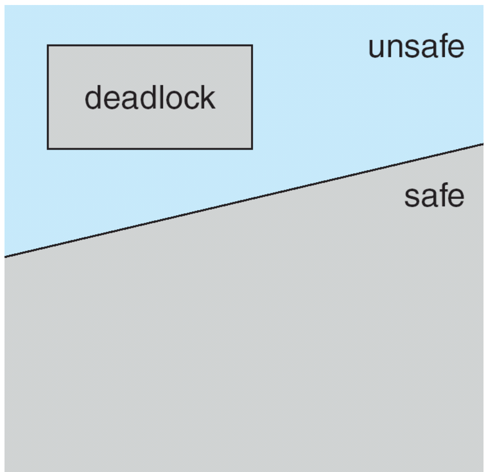
- The simplest and most useful method is that the thread tells how many resources (maximum) of each type it may need to use in its lifetime. Given this *a priori* information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state.
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.
- The 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 <T1, T2, ..., 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.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. However, not all unsafe states are deadlocks. An unsafe state may lead to a deadlock.



Safe state

Displays three threads using a particular resource with 12 available instances

	Maximum Needs	Current Needs
T_0	10	5
T_1	4	2
T_2	9	2

If there is an order in which the threads can be allocated resources up to their maximum needs, and release them after use.
Then the system is in **safe state** because there is a **safe sequence**.

Here, the safe sequence is: **<T_1,T_0,T_2>**

If T_2 was allocated another instance of the resource (i.e., **3**), then a **safe sequence** would no longer exist. T_1 would be able to get its 2 instances and then release **4** available instances, but neither T_0 nor T_2 can be allocated up to their maximum needs of 5 + **5** and 3 + **6** instances, respectively!

Resource-Allocation-Graph Algorithm

In addition to the request and assignment edges in a standard resource-allocation graph, we introduce a new type of edge, called a **claim edge**. A claim edge Ti → Rj indicates that thread Ti may request resource Rj **at some time in the future**.

This edge resembles a request edge in direction but is represented in the graph by a **dashed line**.

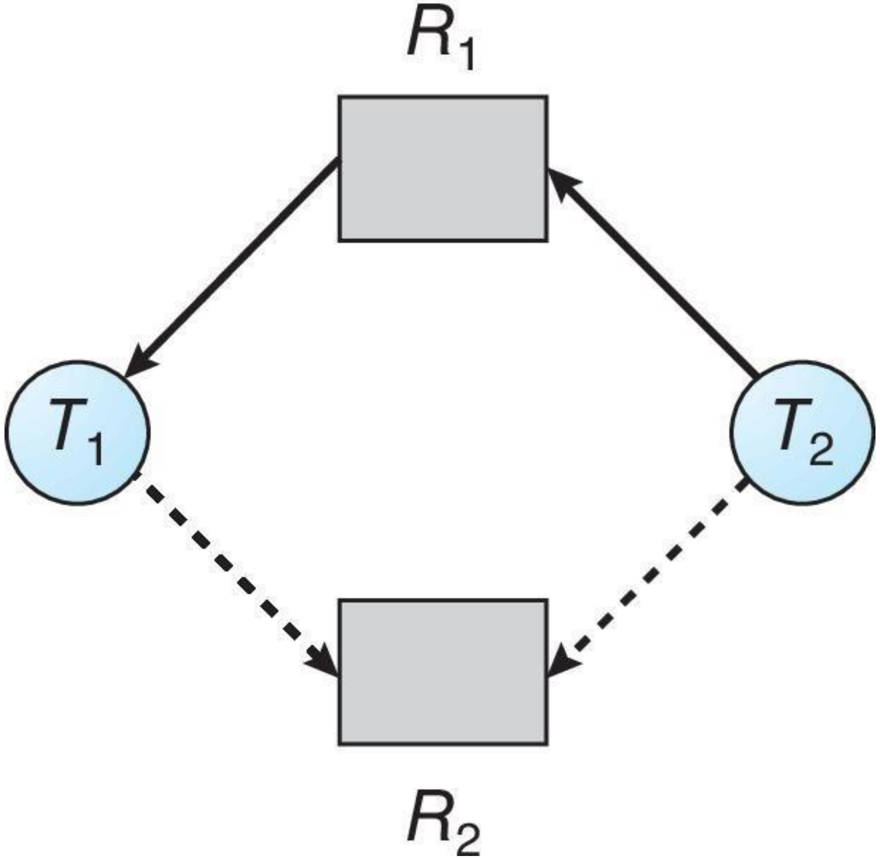
When thread Ti requests resource Rj, the claim edge Ti → Rj is converted to a request edge.

Similarly, when a resource Rj is released by Ti, the assignment edge Rj → Ti is reconverted to a claim edge Ti → Rj.

Note that the resources must be claimed a priori in the system. That is, before thread Ti starts executing, all its claim edges must already appear in the resource-allocation graph.

Resource-Allocation-Graph Algorithm

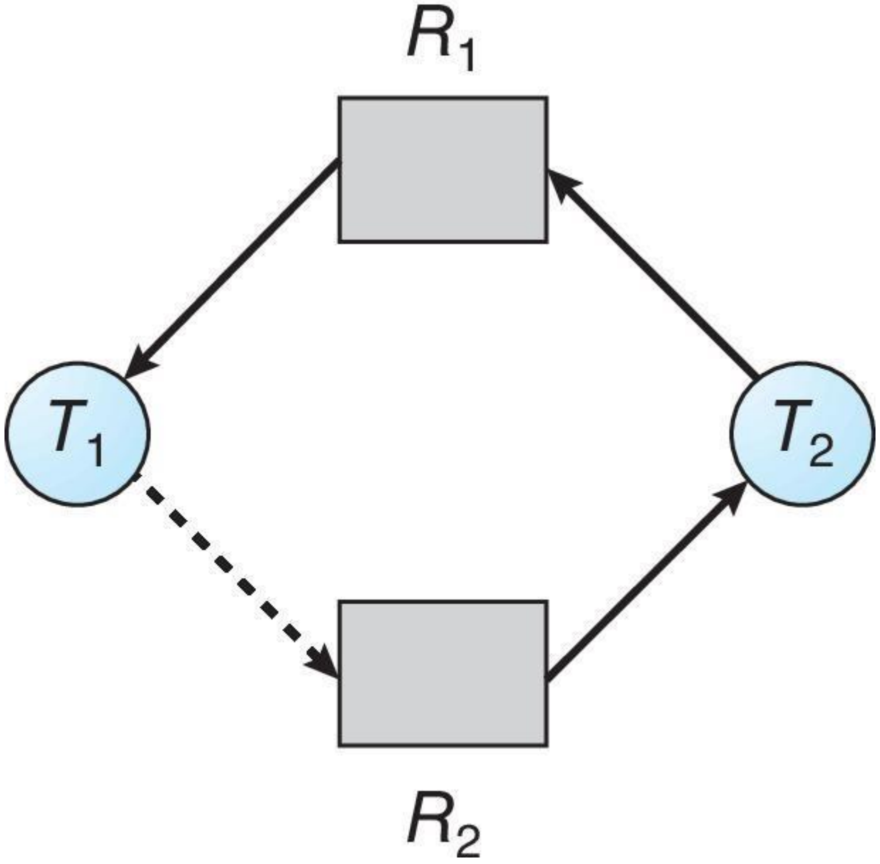
Example



Here, T_1 and T_2 will request to get resource R_2 in future. T_2 will not be able to request and get R_2 even if it is available!

Resource-Allocation-Graph Algorithm

Example



Because If T_2 requests R_2 and gets it allocated to it, this action will create a cycle in the graph. Then we end up in **<u>unsafe state</u>!**

Banker's algorithm

Similar to what we just looked at… there are just more resources with many instances for each research type.

We will not go into depth with Banker's algorithm, but only emphasize the following points:

- A new thread that enters the system must state its maximum needs for the different types of resources. These needs must not exceed the total number of different types of resources.
- When a thread requests a set of resources, then investigate the system about if this allocation will leave the system in a safe state.
  - if so, then assign the set of resources to the thread.
  - if no, then the thread has to wait…

Banker's algorithm

Example

5 threads T_0 through T_4;

3 resource types:

A (10 instances), *B* (5 instances), 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	

Banker's algorithm

Example

The contents of the matrix ***Need*** are defined as follows: ***Max – Allocation***

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

The system is in safe state, as the sequence $\mathbf{<T_1,T_3,T_4,T_2,T_0>}$ meets the safety criterion of safe sequence, which is a way out of any possible deadlock problems.

Banker's algorithm

Example

T_1 request a resource set (1,0,2)

The system examines whether **Need** $\leq$ **Available**

That is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true

New condition

	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	

There exists a safe sequence < T_1, T_3, T_4, T_0, T_2> and T1's request is granted

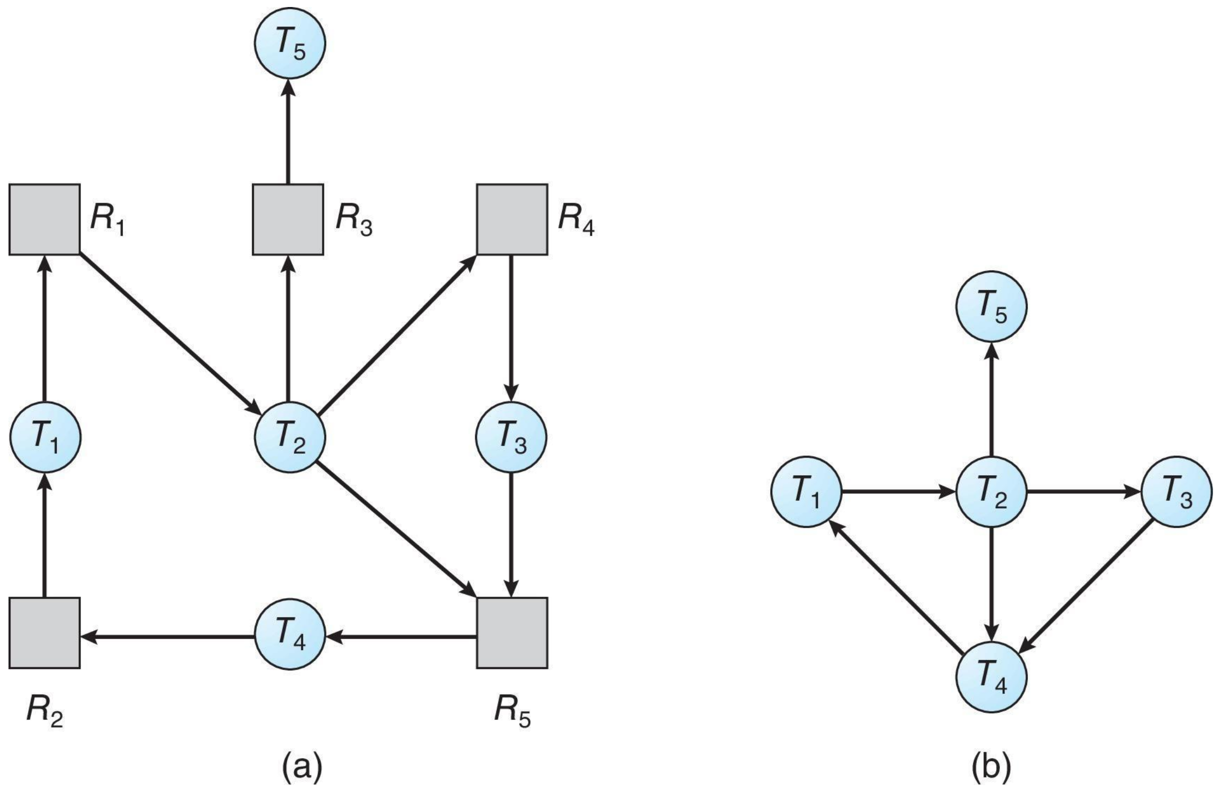
A request (3,3,0) from T_4 is *rejected* because $(3,3,0) \leq (2,3,0) \Rightarrow$ false

A request (0,2,0) from T_0 is *rejected* even though the resource is available, as a safe sequence would no longer exist.

Deadlock detection

Here we allow the system to enter a deadlock

On a periodic basis, the system checks for the existence of cycles



(a) Resource allocation graph

(b) Wait-for graph

…best suited for systems with one instance of each resource type

Deadlock detection

If we have multiple instances of each resource, then matrices of Banker's algorithm are used…

5 threads $\boldsymbol{T_0}$ through $\boldsymbol{T_4}$;

3 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	

A safe sequence exists< $\boldsymbol{T_0, T_2, T_3, T_1, T_4}$> i.e., no deadlock

Note that Request-matrix is what the threads request for at t_0

Deadlock detection

T_2 requests an instance of type ***C***

	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 **1**	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

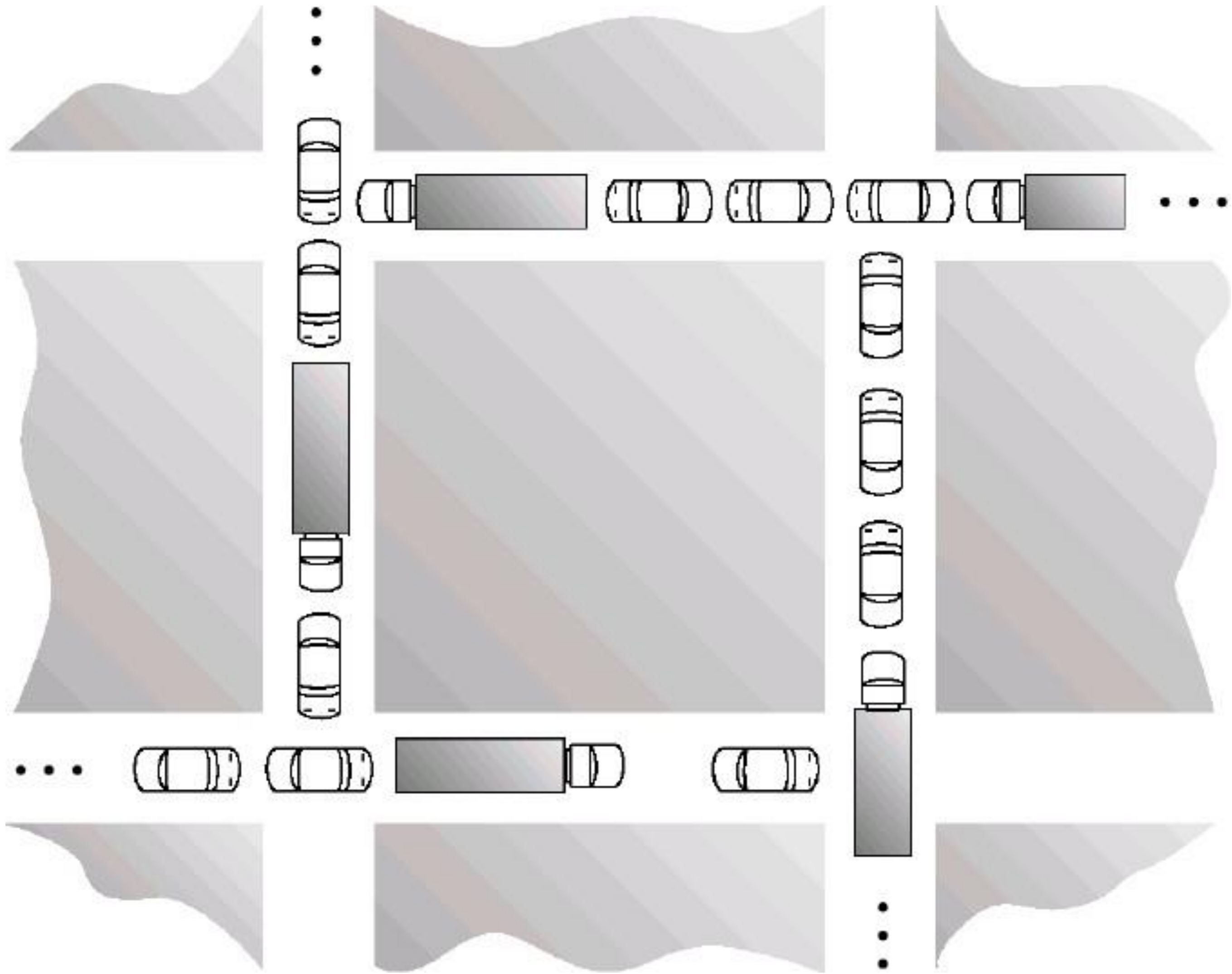
Although ***T_0***, can release all of its resources, it is not sufficient to meet the requests of the other processes! Therefore, a **deadlock**, exists where ***T_1***, ***T_2***, ***T_3***, and ***T_4*** are included

Recovery from deadlock

Deadlock in the real world…

Recovery from deadlock

Thread termination and resource preemption



How are threads chosen for termination? (e.g., What the priority of the process is? How many and what types of resources the process has used)

Who do we choose thread as a victim?

What about starvation?

System modes

The strategies:

Avoidance ↓ safe
Prevention ↓ safe + unsafe
Detection ↓ safe + unsafe + deadlock