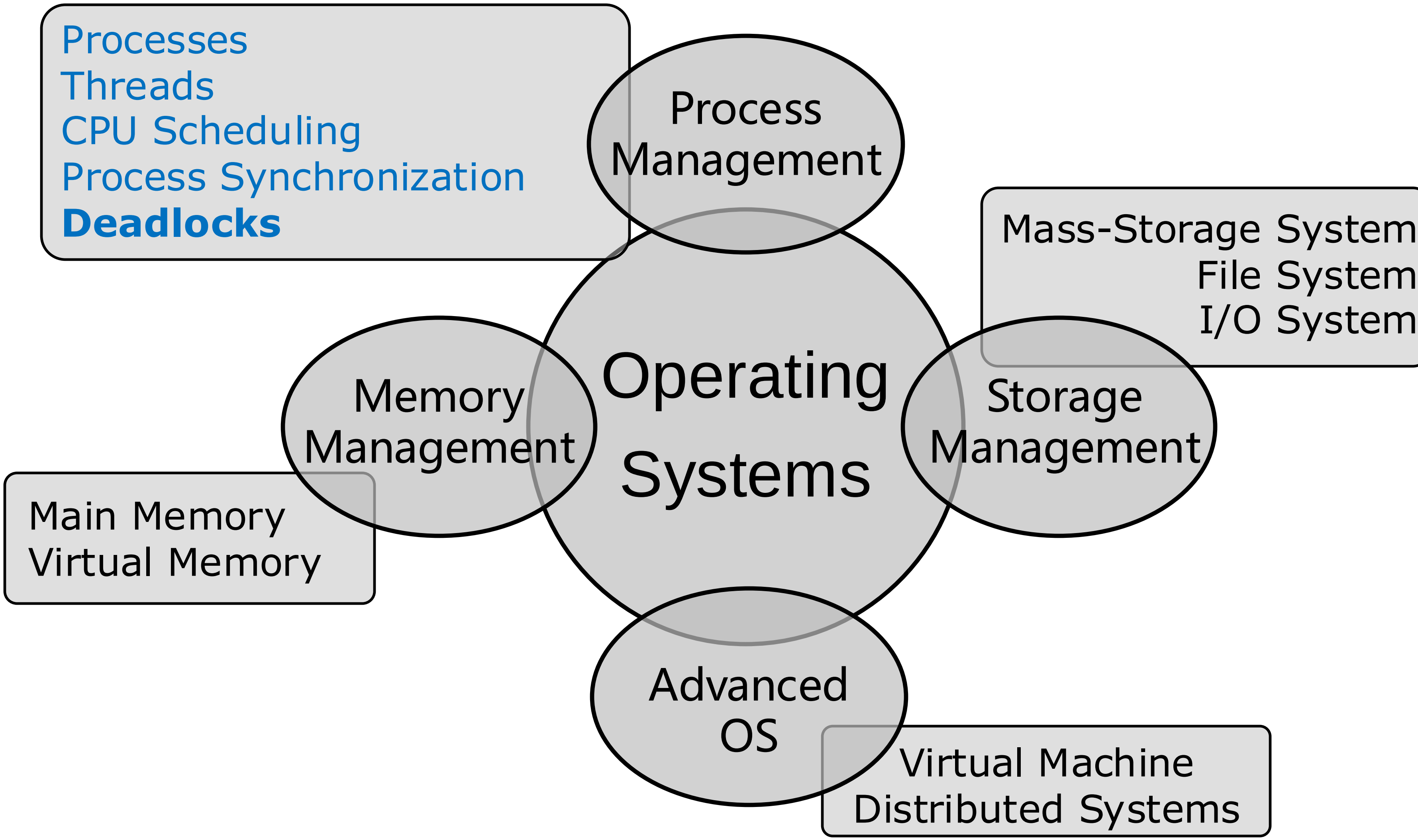
Deadlocks

Shengzhong Liu

Department of Computer Science and Engineering

Shanghai Jiao Tong University

Operating System Topics

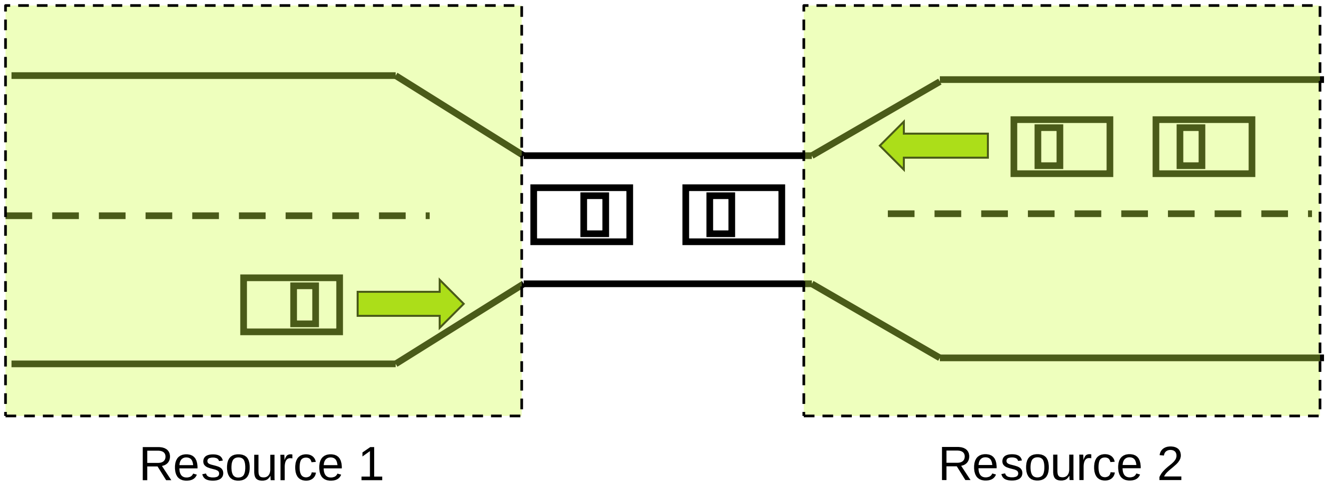


Outline

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - Never enters a deadlock:
    - Deadlock Prevention
    - Deadlock Avoidance
  - Enters and recovers from the deadlock:
    - Deadlock Detection
    - Recovery from Deadlock

System Model

Analogy: Bridge Crossing Example



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- A **deadlock** occurs when two cars get on the bridge from different directions at the same time

Deadlock with Semaphores

- Data:
  - A semaphore $\mathbf{s_1}$ initialized to 1
  - A semaphore $\mathbf{s_2}$ initialized to 1
- Assume we have two threads T_1 and T_2

- T_1:

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

- T_2:

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

- **Deadlock**: A set of blocked threads each holding some resources and waiting to acquire the resources held by another thread in the set

System Model

- Notations:
  - Threads **T_1, T_2, …, T_n**
  - Resource types **R_1, R_2, ..., R_m**

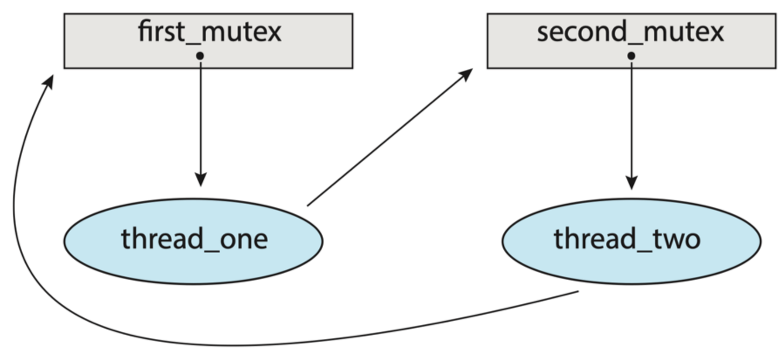
    e.g., CPU, memory space, I/O devices, mutex and semaphores
  - **Each resource type R_i has W_i instances.**
- Each process utilizes a resource in the following sequence:
  - **Request**: Wait if the request cannot be granted immediately.
  - **Use**: The process operates on the resource.
  - **Release**: The process releases the resource.
- **Deadlock State:**
  - 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.
  - Events mainly include resource acquisition and release.

Deadlock Characterization

Deadlock Characterization

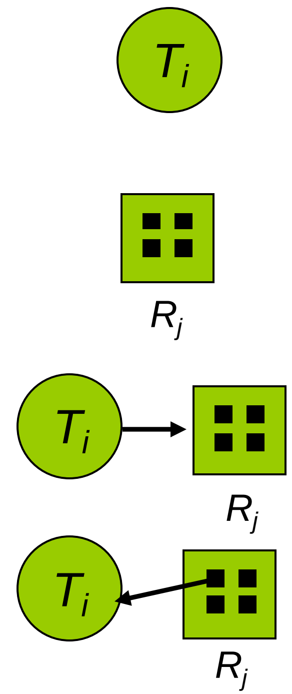
Deadlock can arise if four conditions hold simultaneously.

- **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 only released voluntarily by the thread after its task has completed
- **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 held by T_1
  - T_1 is waiting for a resource held by T_2
  - …
  - T_{n-1} is waiting for a resource held by T_n
  - T_n is waiting for a resource held by T_0



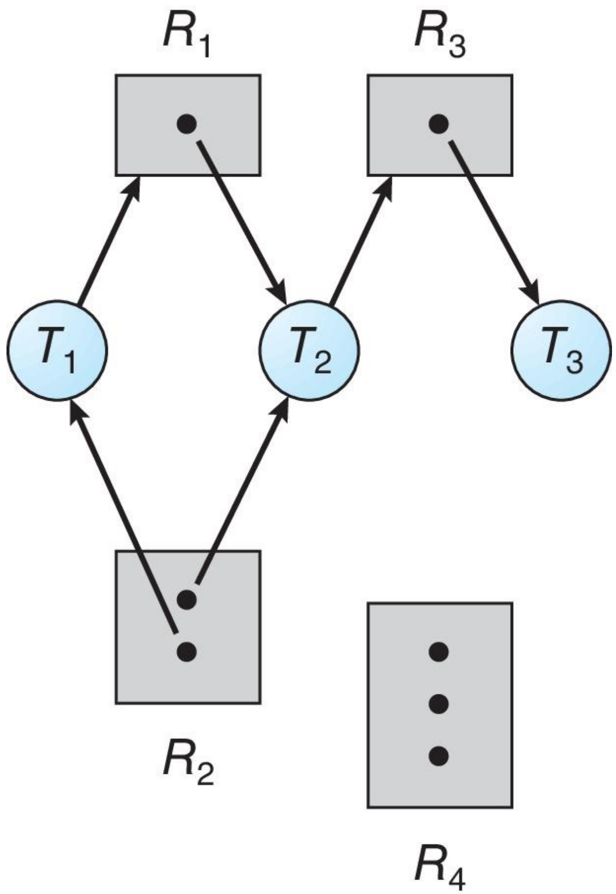
Resource-Allocation Graph

- Deadlocks can be identified with **system resource-allocation graph**.
  - A set of vertices **V** and a set of edges **E**.
- V is partitioned into two types:
  - $\mathbf{T = \{T_1, T_2, \ldots, T_n\}}$ is the set consisting of all the threads in the system.
  - $\mathbf{R = \{R_1, R_2, \ldots, R_m\}}$ is the set consisting of all resource types in the system
- There are two types of edges:
  - **Request edge** – directed edge $\mathbf{T_i \rightarrow R_j}$
  - **Assignment edge** – directed edge $\mathbf{R_j \rightarrow T_i}$



Example: Resource Allocation Graph

- Threads: $T = \{T_1, T_2, T_3\}$
- Resources: $R = \{R_1, R_2, R_3, R_4\}$
  - Resource instances:
    - R_1 x1, R_2 x2, R_3 x1, R_4 x3
- Edges: $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$
  - T_1:
    - holds one instance of R_2
    - is waiting for an instance of R_1
  - T_2
    - holds one instance of R_1, one instance of R_2
    - is waiting for an instance of R_3
  - T_3 :
    - holds one instance of R_3

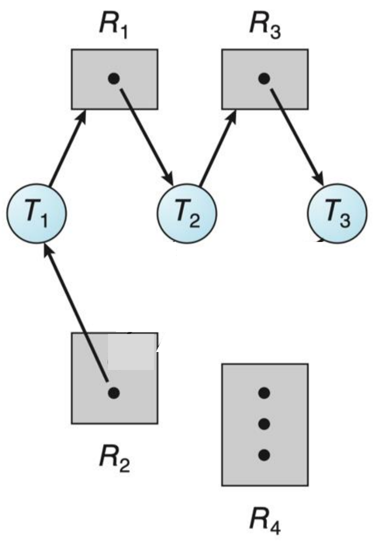


Basic Facts

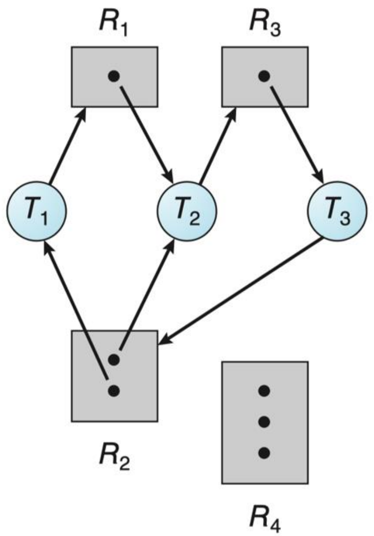
- If graph **contains no circle** $\Rightarrow$ No deadlock
- If graph **contains a circle** $\Rightarrow$
  - If only one instance per resource type, then deadlock
  - If several instances per resource type, possibility of deadlock
    - Cycle is a necessary but not sufficient condition for deadlocks.
- Question:
  - Can you find a way to determine whether there is a deadlock, given a resource allocation graph with several instances per resource type?

Example: Resource Allocation Graph w/ Deadlock(s)

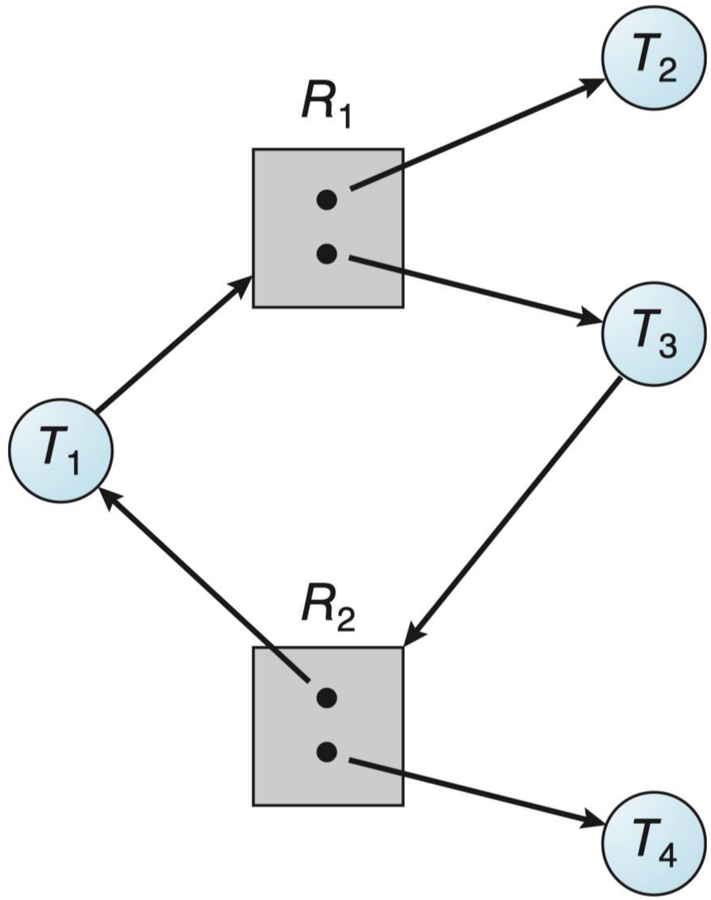
- One circle
  - $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$



- Two circles
  - $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
  - $T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$



Example: Graph with A Cycle w/o Deadlock



Methods for Handling Deadlocks

Methods for Handling Deadlocks

- **Option 1**: Ensure that the system will **never** enter a deadlock state
  - **Deadlock prevention**
    - 【Pessimistic policy, no prior information】
    - Ensure that at least one necessary condition cannot hold.
    - Conservative resource allocation, but may lead to underutilization
  - **Deadlock avoidance**
    - 【Dynamic estimation, use prior information】
    - Employ strategies to ensure the system does not enter deadlock at each step of resource allocation

- **Option 2**: **Allow** the system to enter a deadlock state and then recover
  - **Deadlock detection**
  - **Deadlock recovery**

Methods for Handling Deadlocks: Deadlock Prevention

Deadlock Prevention

Examine the **necessary conditions for deadlocks** to happen:

- **Mutual Exclusion** – removed for sharable resources (e.g., read-only files); must hold for non-sharable resources
  - We can not prevent deadlocks by denying the mutual-exclusion condition.
- **Hold and Wait** – when a thread requests a resource, it can not hold any other resources
  - <u>Option 1</u>: Require a thread to request and be allocated all its resources before it begins execution
    - Impractical for most applications.
  - <u>Option 2</u>: Allow a thread to request resources only when the thread has none allocated to it.
    - Before requesting additional resources, a thread must release all allocated resources.
  - <u>Limitations</u>: Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No Preemption**:
  - Support resource preemption:
    - If a thread is holding some resources and requesting another resource that can not be allocated, then release all its held resources
    - Preempted resources are added to the list of waiting resources
    - The thread is restarted only when it can regain its old resources and new resources
  - Often applied to resources whose state can be easily saved and restored later, such as CPU registers and database transactions.
    - Can not be applied to mutex locks and semaphores.
- **Circular Wait:**
  - Impose a total ordering of all resource types, and require that each thread requests resources in an increasing order of enumeration
  - The only practical option.

Break Circular Wait with Lock Order

- Policy:
  - 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.

```
/* 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);
}
```

- If:

```
first_mutex = 1
second_mutex = 5
```

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

```
/* 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);
}
```

Methods for Handling Deadlocks：Deadlock Avoidance

Deadlock Avoidance

Requires the system to have some additional **prior information**

- Simplest model: Each thread declare the **maximum #resources of each needed type**

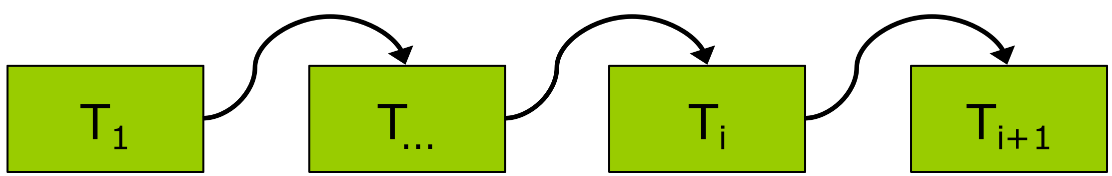
- **Avoidance**: dynamically check the resource-allocation state, ensure there can never be a circular-wait condition

- **Resource-allocation state:**
  - Number of available resources
  - Number of allocated resources
  - Maximum demands of the threads

Additional knowledge required!

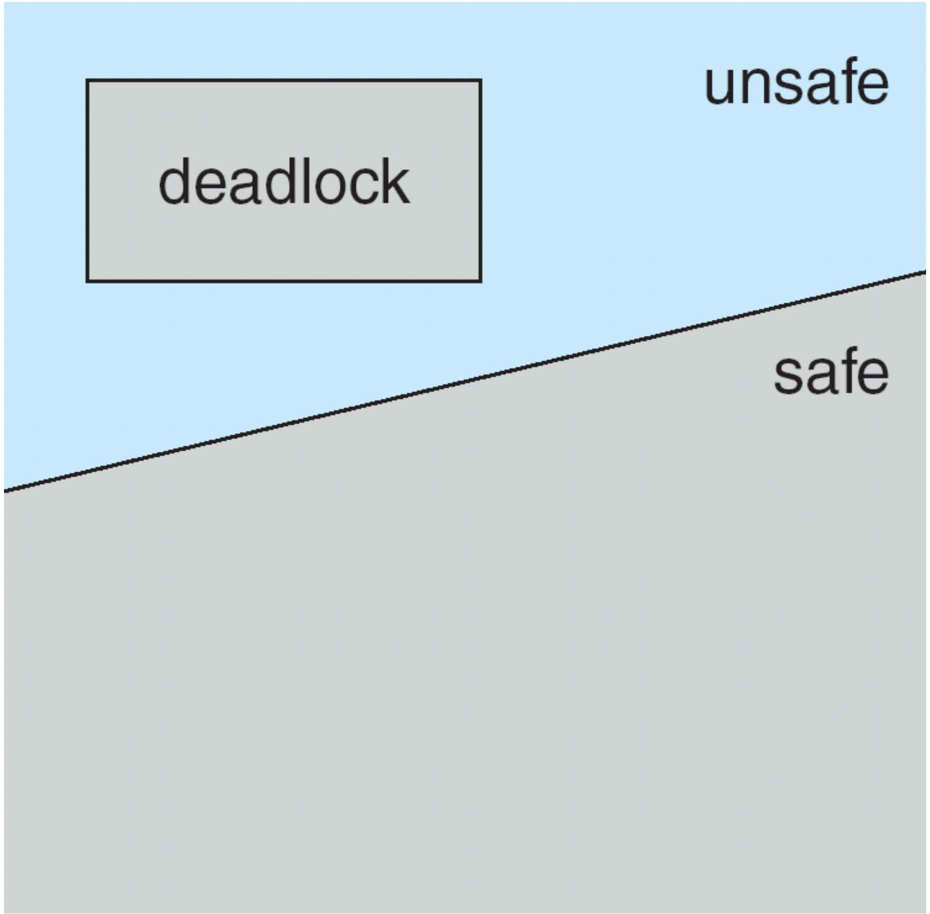
Safe State

- When a thread requests an available resource, must decide if the 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_i, the resources that it will request can be satisfied by currently available resources + resources held by all the T_j, with $j < i$
- That is:
  - If T_i resource needs are not available, T_i waits until all T_j have finished
  - When T_j is finished, T_i can obtain its resources, execute, return all resources, and terminate
  - When T_i terminates, T_{i+1} can obtain its needed resources, and so on



Safe, Unsafe, Deadlock State

- If a system is in safe state
  ⇒ no deadlocks
- If a system is in unsafe state
  ⇒ possibility of deadlock
- Avoidance
  ⇒ ensure that a system will never enter an unsafe state.



Safe & Unsafe States

	Maximum Needs	Holds	Needs
T_0	10	5	5
T_1	4	2	2
T_2	9	2	7

Available
3

Safe sequence: ?

Safe & Unsafe States

	Maximum Needs	Holds	Needs
T_0	10	5	5
T_1	4	4	0
T_2	9	2	7

Available
1

Safe sequence: T_1

Safe & Unsafe States

	Maximum Needs	Holds	Needs
T_0	10	5	5
T_1	4	--	--
T_2	9	2	7

Available
5

Safe sequence: T_1

Safe & Unsafe States

	Maximum Needs	Holds	Needs
T_0	10	10	0
T_1	4	--	--
T_2	9	2	7

Available
0

Safe sequence: $T_1 \rightarrow T_0$

Safe & Unsafe States

	Maximum Needs	Holds	Needs
T_0	10	--	--
T_1	4	--	--
T_2	9	2	7

Available
10

Safe sequence: $T_1 \rightarrow T_0$

Safe & Unsafe States

	Maximum Needs	Holds	Needs
T_0	10	--	--
T_1	4	--	--
T_2	9	9	0

Available
3

Safe sequence: $T_1 \rightarrow T_0 \rightarrow T_2$

Safe & Unsafe States

	Maximum Needs	Holds	Needs
T_0	10	--	--
T_1	4	--	--
T_2	9	--	--

Available
12

Safe sequence: $T_1 \rightarrow T_0 \rightarrow T_2$

Safe & Unsafe States

Back from beginning.

	Maximum Needs	Holds	Needs
T_0	10	5	5
T_1	4	2	2
T_2	9	2	7

Available
3

Safe sequence: ?

Safe & Unsafe States

Give T2 one more?

	Maximum Needs	Holds	Needs
T_0	10	5	5
T_1	4	2	2
T_2	9	3	6

Available
2

Safe sequence: ?

Safe & Unsafe States

	Maximum Needs	Holds	Needs
T_0	10	5	5
T_1	4	--	--
T_2	9	3	6

Available
4

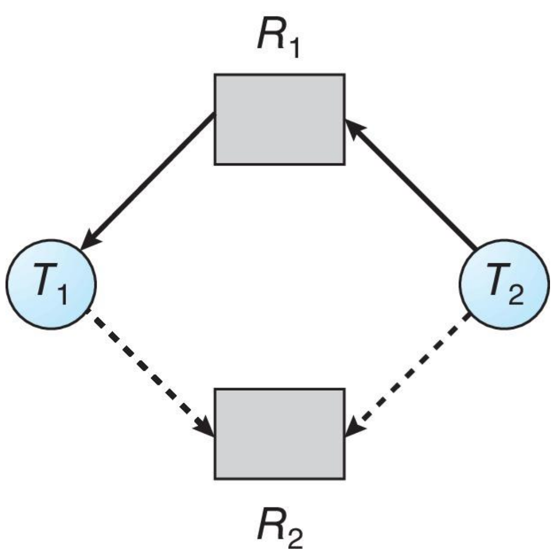
Safe sequence: $T_1 \rightarrow$?

Deadlock Avoidance Algorithms

- Avoidance algorithms ensure the system will never deadlock
  - When a process requests a resource, the request is granted only if **the allocation leaves the system in a safe state**.
- Two avoidance algorithms
  - If we have a single instance of a resource type
    - Use a **resource-allocation-graph algorithm**
  - If we have multiple instances of a resource type
    - Use the **banker's algorithm**

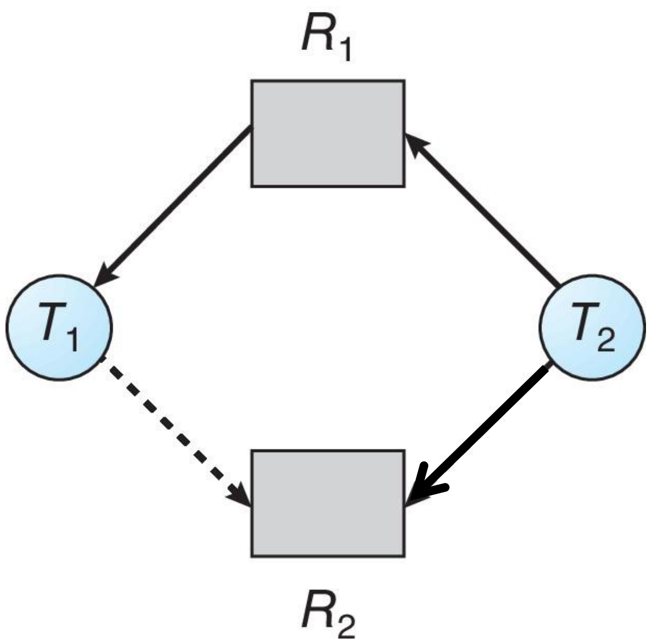
Resource-Allocation-Graph Algorithm

- **[New]**: **Claim edge** $T_i \rightarrow R_j$ indicates that thread T_i **may** request resource R_j; represented by a **directed dashed line**
  - **Claim edge** converts to **request edge** when a thread requests a resource
  - **Request edge** converts 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 (the edge is removed if the thread finishes)

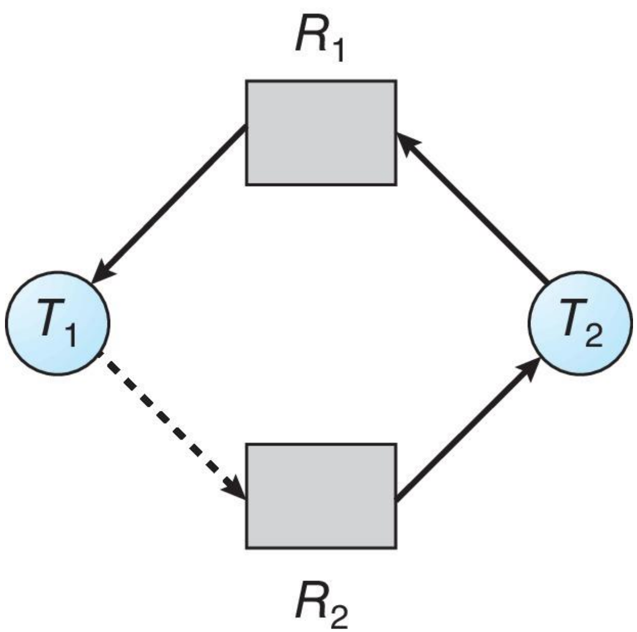


Resource-Allocation-Graph Algorithm

- 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 a circle in the resource allocation graph



Can we grant T_2's request for R_2?



Circle! Therefore, T_2's request cannot be granted, and T_2 needs to wait.

Banker's Algorithm

- We have multiple instances of each resource
- Each thread must claim its maximum use of each resource in advance
- When a thread 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 threads, and ***m*** = number of resource types.

- **`Available`**: **m vector**. The #available resources of each type.
  - If `available[j] = k` ➔ *k* instances of resource type R_j are available
- **`Max`**: **n x m matrix**. Maximum demand of each thread.
  - If `Max[i,j] = k` ➔ thread T_i may request at most *k* of R_j
- **`Allocation`**: **n x m matrix**. #allocated resource of each type.
  - If `Allocation[i,j] = k` ➔ T_i is currently allocated *k* of R_j
- **`Need`**: **n x m matrix**. The remaining resource need of each thread.
  - If `Need[i,j] = k` ➔ P_i may need *k* more R_j to complete
  - `Need[i,j] = Max[i,j] – Allocation[i,j]`

Banker's Algorithm

- **【Variable】Work**: The available resources can be allocated to processes at any given point during execution.
  - As the algorithm progresses, the "work" variable is used to simulate resource allocation to threads.

- **【Status】Finish**: A boolean array that indicates whether each thread can complete its execution without leading to deadlock.
  - As the algorithm proceeds, the "finish" array is updated to reflect which processes can potentially complete execution safely.

- **Key idea**:
  - Employs a series of **checks and simulations** to ensure resource allocation will not result in a deadlock.
  - If **<u>a safe sequence exists</u>**, resources can be allocated to threads without risking deadlock.
  - **<u>Otherwise</u>**, the system should wait until resources become available or deny the request to avoid potential deadlock.

Subroutine: Safety Check Algorithm

Find out whether or not the system is in a safe state.

1. Let **Work** and **Finish** be vectors of length *m* and *n*, respectively.
   - **Work = Available**
   - **Finish[i] = false**, for **i = 0, 1, …, n- 1**

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

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

   (b) **Need$_i$ ≤ Work**

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

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

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

Resource-Request Algorithm for Thread T_i

Determine **whether the requrests can be safely granted**.

$Request_i$ = request vector for thread T_i. If **$Request_i[j] = k$** then thread T_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 thread has exceeded its maximum claim
2. If **$Request_i \leq Available$**, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- Test the safety of the new system state:
  - 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*

Example of Banker's Algorithm

- **5 threads** T_0 through T_4;
  3 resource types:
  A (**10** instances), *B* (**5** instances), and *C* (**7** instances)

Initial system snapshot:

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

- **Is the system in safe state?**

Applying Safety Algorithm

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

Safe sequence: T_1

Applying Safety Algorithm

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

Safe sequence: $T_1 \rightarrow T_3$

Applying Safety Algorithm

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

Safe sequence: $T_1 \rightarrow T_3 \rightarrow T_0$

Applying Safety Algorithm

	Max	Allocation	Need	Available
	A B C	*A B C*	*A B C*	*A B C*
				10 5 5
T_4	4 3 3	0 0 2	4 3 1	

Safe sequence: $T_1 \rightarrow T_3 \rightarrow T_0 \rightarrow T_2$

Applying Safety Algorithm

	Max	Allocation	Need	Available
	A B C	*A B C*	*A B C*	*A B C*
				10 5 7

Safe sequence: $T_1 \rightarrow T_3 \rightarrow T_0 \rightarrow T_2 \rightarrow T_4$

Safe!

Example: T_1 Request (1,0,2)

- Check that Request ≤ Available (that is, **T_1 → (1,0,2)** ≤ (3,3,2) ⇒ true)

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

- Executing safety algorithm shows that sequence < T_1, T_3, T_0, T_2, T_4> satisfies safety requirement

- Grant the request by T_1

Example: T_0 Request (0,2,0)

- Check that Request ≤ Available (that is, **T_0 → (0,2,0)** ≤ (2,3,0) ⇒ true)

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

- Does there exist a safe sequence exist?
  - **No!**
- **The request should be held and wait.**

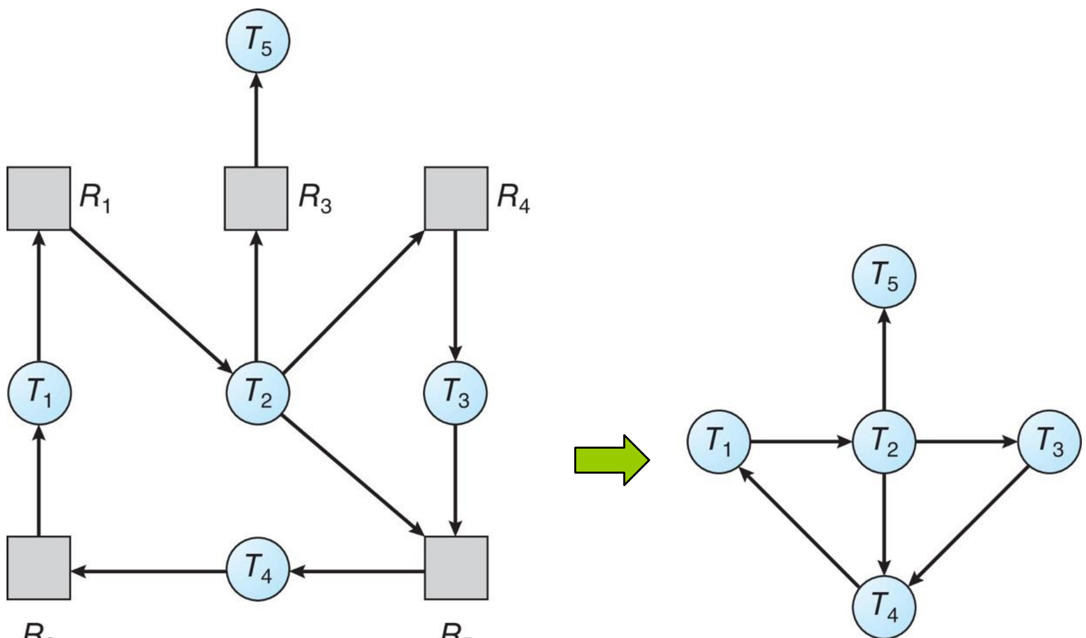
Methods for Handling Deadlocks: Deadlock Detection

Deadlock Detection

- Allow the system to enter a deadlock state
- Deadlock detection algorithm
- Deadlock 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 circle in the graph. If there is a circle, there exists a deadlock



(a) (b)

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, and initialize:
   (a) Work = Available
   (b) For i = 1,2, …, n, if $Allocation_i \neq 0$, then Finish[i] = false; otherwise, Finish[i] = true

2. Find an index i such that both:
   (a) Finish[i] == false
   (b) $Request_i \leq Work$
   If no such i exists, go to step 4

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

4. If Finish[i] == false, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, **if Finish[i] == false, then 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 threads T_0 through T_4; three resource types
  A (7 instances), B (2 instances), and C (6 instances)

- Initial snapshot:

	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

	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	

- State of the system?
  - Can reclaim resources held by thread T_0, but insufficient resources to fulfill other threads' requests
  - Deadlock exists, consisting of threads 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 threads will need to be rolled back?

- If the detection algorithm is invoked for every resource request, considerable overhead in computation time will be incurred.

- If the detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph
  - We would not be able to tell which of the many deadlocked threads “caused” the deadlock.

Methods for Handling Deadlocks: Recovery from Deadlocks

Recovery from Deadlock

- Process/Thread **Termination**
  - Abort one or more processes to break the circular wait

- Resource **Preemption**
  - Preempt some resources from one or more of the deadlocked threads

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?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to compete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Resource Preemption

- **Selecting a victim** – minimize cost
  - We must determine the order of preemption to minimize cost
- **Rollback** – return to some safe state, restart process from that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor
  - We must ensure that a process can be picked as a victim only a (small) finite number of times
  - We can include number of rollbacks in the cost factor

Summary

- Deadlock occurs in a set of threads when every thread in the set is waiting for an event that can only be caused by another thread in the set
- Four necessary conditions for deadlock: (1) mutual exclusion, (2) hold and wait, (3) no preemption, and (4) circular wait.
  - Deadlock is only possible when all four conditions are present
- Deadlocks can be modeled with **resource-allocation graphs**, where a cycle indicates deadlock.
- **Deadlock prevention**: Ensuring one of the four necessary conditions cannot occur
- **Deadlock avoidance**: Evaluate threads and resources to determine if the system is in a deadlocked state
- **Deadlock recovery**: Process termination or resource preemption.

Homework

- Reading
  - Chapter 8

- Check Canvas for HW2 release, due on **Mar. 21 at 23:59**!