CSC 4320/6320: Operating Systems

Chapter 08: Deadlocks-II

Spring 2025

Instructor: Dr. Md Mahfuzur Rahman
Department of Computer Science, GSU

Disclaimer

The slides to be used in this course have been created, modified, and adapted from multiple sources:

- *The slides are copyright Silberschatz, Galvin and Gagne, 2018. The slides are authorized for personal use, and for use in conjunction with a course for which Operating System Concepts is the prescribed text. Instructors are free to modify the slides to their taste, as long as the modified slides acknowledge the source and the fact that they have been modified.*

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

Announcement

- Exam 2:
  - Date: March 27, 2025

Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system.
  - appropriate for systems where deadlocks are very rare and the overhead of addressing them is considered **too high**
  - **Example:** Single-user personal computers, embedded systems, etc.

Questions

1. To handle deadlocks, operating systems most often ____.
A) pretend that deadlocks never occur ✓
B) use protocols to prevent or avoid deadlocks
C) detect and recover from deadlocks
D) None of the above

2. Both deadlock prevention and deadlock avoidance techniques ensure that the system will never enter a deadlocked state.
True ✓
False

3. Most operating systems choose to ignore deadlocks, because
A) handling deadlocks is expensive in terms of performance and resources.
B) deadlocks occur infrequently.
D) methods used to recover from livelock may be put to use to recover from deadlock.
D) All of the above. ✓

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
- **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**:
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the 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:

```
/* 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);
}
```

It is up to application developers to write programs that follow the ordering.

Proof of no circular wait

- Assuming that a circular wait exists (proof by contradiction).
- Let the set of threads involved in the circular wait be $\{T_0, T_1, \ldots, T_n\}$, where T_i is waiting for a resource R_i, which is held by thread T_{i+1}. T_n is waiting for a resource R_n held by T_0.
- Since thread T_{i+1} is holding resource Ri while requesting resource R_{i+1}, we must have $F(R_i) < F(R_{i+1})$ for all i. But this condition means that $F(R_0) < F(R_1) < \ldots < F(R_n) < F(R_0)$.
- By transitivity, $F(R_0) < F(R_0)$, which is impossible. Therefore, there can be no circular wait.

Questions

1. Assume there are three resources, R1, R2, and R3, that are each assigned unique integer values 15, 10, and 25, respectively. What is a resource ordering which prevents a circular wait?

A) R1, R2, R3
B) R3, R2, R1
C) R3, R1, R2
D) R2, R1, R3 ✓

2. Deadlock prevention by denying the mutual-exclusion condition is the simplest way to prevent deadlocks.

True
False ✓

3. In deadlock prevention by denying hold-and-wait condition,

A) resource utilization may be low.
B) starvation is possible.
C) whenever a thread requests a resource, it does not hold any other resources.
D) All of the above. ✓

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

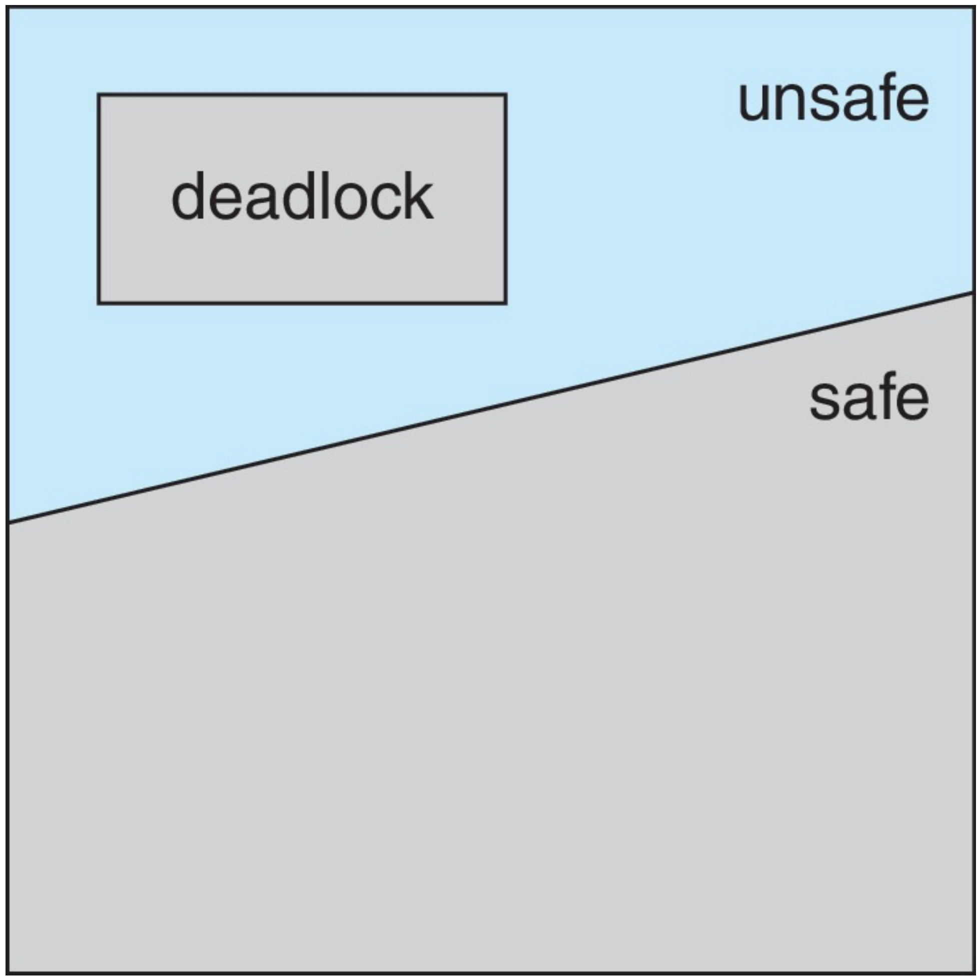
Safe State

- 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_i, the resources that T_i can still 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 immediately available, then T_i can wait until all T_j have finished
  - When T_j is finished, T_i can obtain needed resources, execute, return allocated resources, and terminate
  - When T_i terminates, T_{i+1} can obtain its needed resources, and so on

Basic Facts

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

Safe, Unsafe, Deadlock State



Illustration

- consider a system with 12 resources and 3 threads: T0, T1, and T2.
- The max need for resources and current allocation at time t0 is as shown in the figure.
- three free resources.
- Is there any safe sequence?
- What if, at time t1, thread T2 requests and is allocated one more resource?

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

The sequence <T1, T0, T2> satisfies the safety condition.

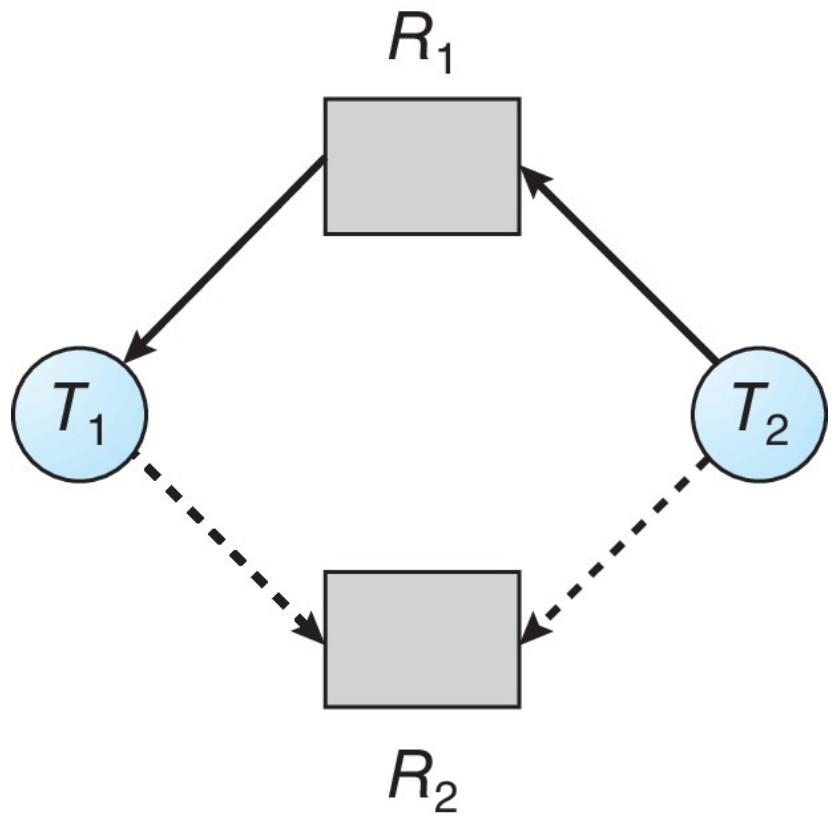
Avoidance Algorithms

- 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

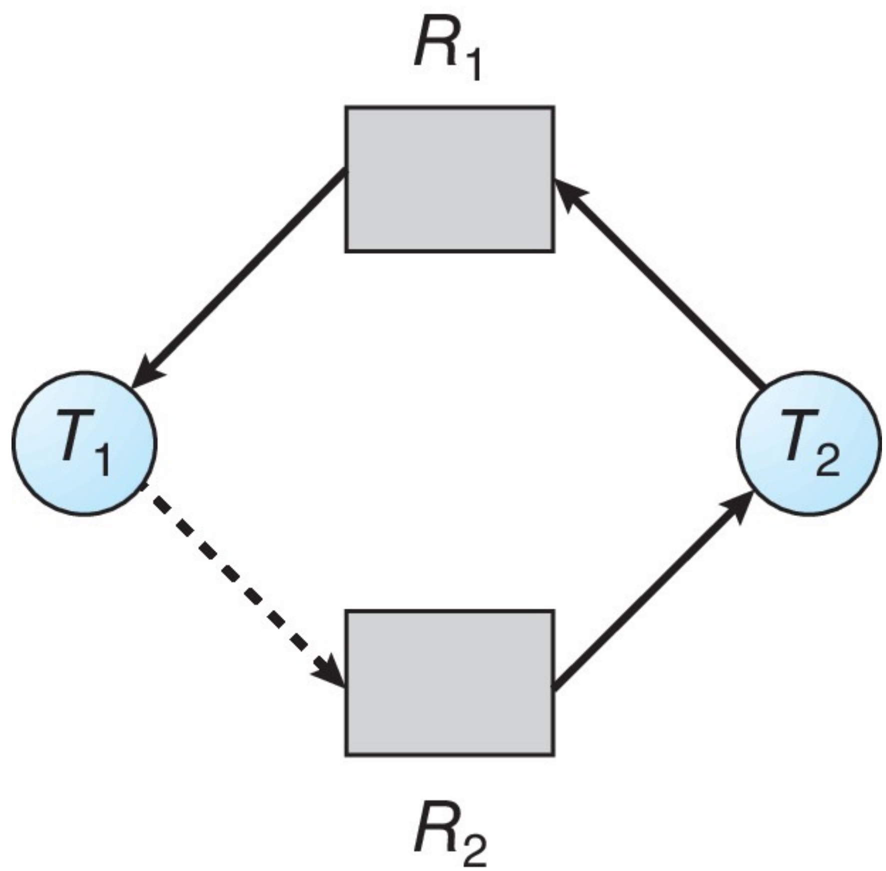
- **Claim edge** $T_i \rightarrow R_j$ indicated that process T_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a thread requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the thread
- When a resource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

Resource-Allocation Graph



Although R2 is currently free, we cannot allocate it to T2, since this action will create a cycle in the graph

Unsafe State In 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

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) $\boldsymbol{Need_i} \leq \boldsymbol{Work}$

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

3. ***Work* = *Work* + $\boldsymbol{Allocation_i}$**

   Finish*[*i*] = *true

   go to step 2

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

Resource-Request Algorithm for Process P_i

Request$_i$ = request vector for process T_i. If ***Request$_i$* [*j*] = *k*** then process T_i wants ***k*** instances of resource type R_j

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

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to T_i
- If unsafe $\Rightarrow$ 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)

- Suppose now that thread T_1 requests one additional instance of resource type A and two instances of resource type C
- 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*
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? (Are resources available?)
- Can request for (0,2,0) by T_0 be granted? (Does it keep the system safe?)