Chapter 08
Deadlocks

A process waiting an event caused by another process

Outline

- System Model
- Deadlock in Multithreaded Applications
- 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
  - E.g. *CPU cycles, memory space, I/O devices*
  - Each resource type R_i has W_i instances.
- Mutex locks and semaphores are also system resources.
  - May lead to deadlock.
  - Not a problem. Each lock has its own resource class.
- We focus on kernel resources.

System Model

- Each thread utilizes a resource as follows:
  - **Request**
  - **Use**
  - **Release**
- The request and release of resources may be system calls.
  - **Wait and signal**
  - **Acquire and release**
- 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.

6

Deadlock Characterization

- Deadlock can arise if four conditions hold simultaneously.
  - Mutual exclusion: only one process at a time can use a resource
  - Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
  - No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task
  - Circular wait: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

7

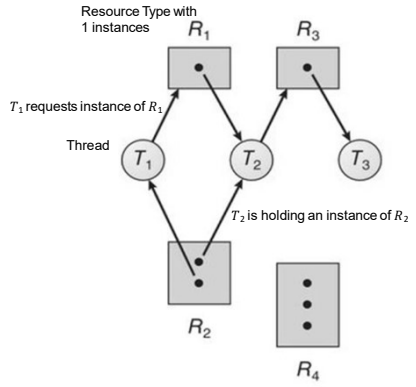
Deadlock Characterization

- Resource-Allocation Graph (RAG): A set of vertices V and a set of edges E.
  - V is partitioned into two types:
    - ★ $T = \{T_1, T_2, .., T_n\}$, the set consisting of all the threads in the system
    - ★ $R = \{R_1, R_2, .., R_m\}$, the set consisting of all resource types in the system
  - Request edge – directed edge $T_i \rightarrow R_j$
    - ★ Thread i requests for resource j
  - Assignment edge – directed edge $R_j \rightarrow T_i$
    - ★ Resource j is assigned to thread i

8

Deadlock Characterization

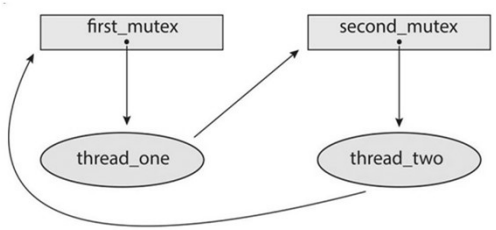
- RAG example



9

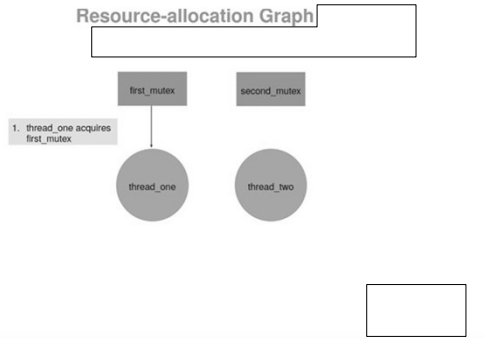
Deadlock Characterization

- RAG with two mutex locks and two threads
  - A deadlock occurs.



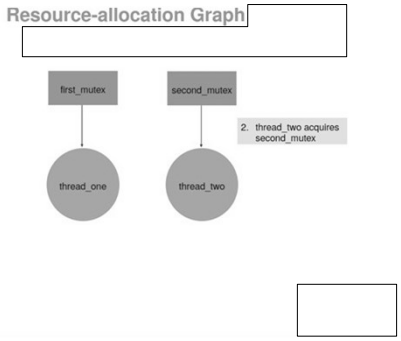
10

Deadlock Characterization



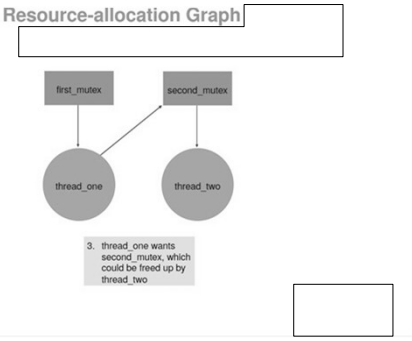
11

Deadlock Characterization



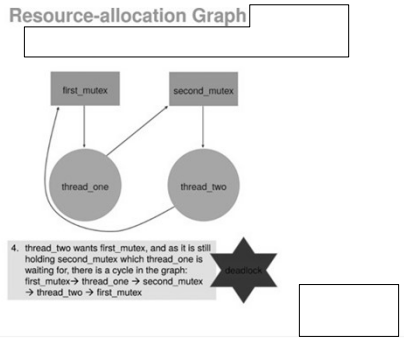
12

Deadlock Characterization

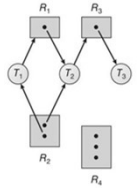


13

Deadlock Characterization



Deadlock Characterization



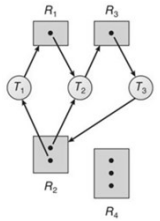
- Example of a Resource Allocation Graph
  - The sets T, R, and E:
    - $T = \{T_1, T_2, T_3\}$
    - $R = \{R_1, R_2, R_3, R_4\}$
    - $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$
  - Resource instances:
    - One instance of resource type R_1
    - Two instances of resource type R_2
    - One instance of resource type R_3
    - Three instances of resource type R_4
  - Thread states:
    - Thread T_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
    - Thread T_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.
    - Thread T_3 is holding an instance of R_3.

Deadlock Characterization

- When will deadlock occur according to RAG?
  - 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

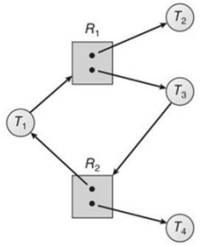
Deadlock Characterization

- Another example of RAG: Add a request edge $T_3 \rightarrow R_2$
  - Two minimal cycles:
    - $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
    - $T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$
  - Threads T_1, T_2, and T_3 are deadlocked



Deadlock Characterization

- Another example of RAG
  - Cycle: $T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
  - No deadlock exists.



18

Methods for Handling Deadlocks

- Ignore the problem and pretend that deadlocks never occur in the system
  - Used by most operating systems, including Linux and Windows
- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention- ensure one of the four conditions cannot hold
  - Deadlock avoidance- deny requests according to the information of resources
- Allow the system to enter a deadlock state and then recover
  - Deadlock detection
  - Deadlock recovery

19

Methods for Handling Deadlocks

- Some systems do not handle deadlock
  - Expense is an essential concern.
  - Deadlock occurs infrequently.
- Methods used to recover from other liveness conditions can be used for deadlock recovery.

20

Deadlock Prevention

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

21

Deadlock Prevention

- **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 process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Deadlock Prevention

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
  - Let $R = \{R_1, R_2, .., Rm\}$ be the set of resource types with unique IDs given by a one-to-one function $F: R \rightarrow N$
  - A thread owns R_i can request R_j iff $F(R_i) < F(R_j)$
  - Prove:
    - Let $\{T_1, .., Tn\}$ be a set of circular-wait threads
    - T_i waits for R_i held by T_{i+1} (T_n waits for R_n held by T_1)
    - $F(R_i) < F(R_{i+1})$
    - $F(R_1) < F(R_2) < \cdots < F(R_n) < F(R_1) \rightarrow\leftarrow$

Deadlock Avoidance

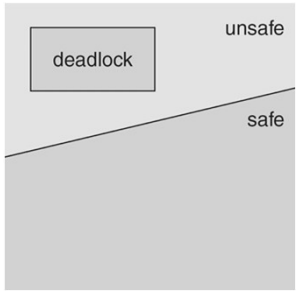
- Requires that the system has some additional ***a priori*** information available
  - Simplest and most useful model requires that each thread declare the ***maximum number*** of resources of each type that it may need.
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation 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.

Deadlock Avoidance

- Safe State
  - When a process 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 $<T_1, T_2, .., Tn>$ 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 threads 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.

Deadlock Avoidance

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock



26

Deadlock Avoidance

- Safe State Example
  - Given a system with twelve resources and three threads: T_0, T_1, and T_2

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

  - Safe state for sequence $<T_1, T_0, T_2>$
  - Become unsafe if T_2 requests one more resource at time t_1
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

27

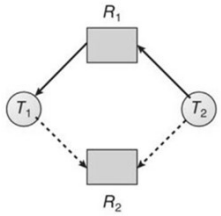
Deadlock Avoidance

- Avoidance Algorithms
  - Single instance of a resource type- Use a resource-allocation graph
  - Multiple instances of a resource type- Use the banker's algorithm

28

Deadlock Avoidance

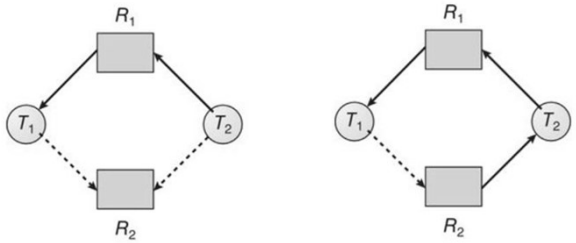
- Resource-Allocation Graph Scheme
  - Claim edge $T_i \rightarrow R_j$ indicated that thread T_j may request resource R_j; represented by a **dashed line**
  - Claim edge converts to request edge when a thread requests a resource
  - Request edge converted to an assignment edge when the resource is 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.



29

Deadlock Avoidance

- Suppose that process P_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



Deadlock Avoidance

- Banker's Algorithm
  - Multiple instances
  - 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.

Deadlock Avoidance

- Data Structures for the Banker's Algorithm
  - Let n = number of threads, 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 thread T_i may request at most k instances of resource type R_j
  - **Allocation**: $n \times m$ matrix. If $Allocation[i, j] = k$ then T_i is currently allocated k instances of R_j
  - **Need**: $n \times m$ matrix. If $Need[i, j] = k$, then T_i may need k more instances of R_j to complete its task
    $Need[i, j] = Max[i, j] - Allocation[i, j]$
- Further notations
  - Given two vectors X and Y, define $X < Y \leftrightarrow X[i] < Y[i] \forall i$
  - $Allocation_i$: resources allocated to thread T_i [ ]
  - $Need_i$: additional resources thread T_i needs.

Deadlock Avoidance

- Safety Algorithm
  1. Let $\boldsymbol{Work}$ and $\boldsymbol{Finish}$ be vectors of length m and n, respectively.
     Initialize:
     $\boldsymbol{Work} = \boldsymbol{Available}$
     $\boldsymbol{Finish}[i] = \boldsymbol{false}$ **for** $i = 0, 1, .., n - 1$
  2. Find an $\boldsymbol{i}$ such that both:
     (a) $\boldsymbol{Finish}[\boldsymbol{i}] == \boldsymbol{false}$
     (b) $\boldsymbol{Need}_i \leq \boldsymbol{Work}$
     If no such $\boldsymbol{i}$ exists, go to step4
  3. $\boldsymbol{Work} = \boldsymbol{Work} + \boldsymbol{Allocation}_i$
     $\boldsymbol{Finish}[\boldsymbol{i}] = \boldsymbol{true}$
     go to step 2
  4. If $\boldsymbol{Finish}[\boldsymbol{i}] == \boldsymbol{true}$ for all $\boldsymbol{i}$, then the system is in a safe state.
- Require $O(m \times n^2)$ operations

Deadlock Avoidance

- Resource-Request Algorithm for thread T_i

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

  1. If $\boldsymbol{Request_i} \le \boldsymbol{Need_i}$ go to step 2. Otherwise, raise error condition, since thread has exceeded its maximum claim
  2. If $\boldsymbol{Request_i} \le \boldsymbol{Available}$, 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 ⇒ the resources are allocated to $\boldsymbol{T_i}$
     - ⋆ If unsafe ⇒ $\boldsymbol{T_i}$ must wait, and the old resource-allocation state is restored

Deadlock Avoidance

- Example of Banker's Algorithm
  - 5 thread 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	

Deadlock Avoidance

- 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

Deadlock Avoidance

- Can request for (1,0,2) by $\boldsymbol{T_1}$ be granted?
  - Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

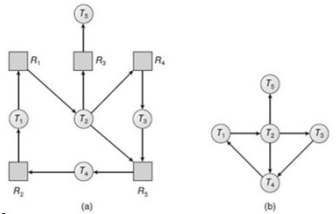
	Allocation	Need	Available
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

  - Executing safety algorithm shows that sequence $< T_1, T_3, T_4, T_2, T_0 >$ satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{T_4}$ be granted? No.
- Can request for (0,2,0) by $\boldsymbol{T_0}$ be granted? No.

Deadlock Detection

- A system allowing deadlock state may provide
  - Detection algorithm
  - Recovery scheme

Deadlock Detection



- Single Instance of Each Resource Type
  - Maintain wait-for graph
    - ★ Nodes are threads
    - ★ $\boldsymbol{T_i \rightarrow T_j}$ if $\boldsymbol{T_i}$ is waiting for $\boldsymbol{T_j}$
      That is, two edges $T_i \rightarrow R_q$ and $R_q \rightarrow T_j$ exist.
  - 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 $\boldsymbol{n^2}$ operations, where $\boldsymbol{n}$ is the number of vertices in the graph

Deadlock Detection

- Several Instances of a Resource Type
  - **Available**: A vector of length $\boldsymbol{m}$ indicates the number of available resources of each type
  - **Allocation**: An $\boldsymbol{n} \times \boldsymbol{m}$ matrix defines the number of resources of each type currently allocated to each thread
  - **Request**: An $\boldsymbol{n} \times \boldsymbol{m}$ matrix indicates the current request of each thread. If $\boldsymbol{Request\ [i][j] = k}$, then thread $\boldsymbol{T_i}$ is requesting $\boldsymbol{k}$ more instances of resource type $\boldsymbol{R_j}$.
- Further notations
  - Given two vectors X and Y, define $X \leq Y \leftrightarrow X[i] \leq Y[i]\ \forall i$
  - $Allocation_i$: resources allocated to thread T_i
  - $Request_i$: current request from thread T_i

Deadlock Detection

- Detection Algorithm
  1. Let $\boldsymbol{Work}$ and $\boldsymbol{Finish}$ be vectors of length $\boldsymbol{m}$ and $\boldsymbol{n}$, respectively Initialize:
     (a) $\boldsymbol{Work = Available}$
     (b) For $\boldsymbol{i = 0, 1, ..., n-1}$, if $\boldsymbol{Allocation_i \neq 0}$, then $\boldsymbol{Finish[i] = false}$; otherwise, $\boldsymbol{Finish[i] = true}$
  2. Find an index $\boldsymbol{i}$ such that both:
     (a) $\boldsymbol{Finish[i] == false}$
     (b) $\boldsymbol{Request_i \leq Work}$
     If no such $\boldsymbol{i}$ exists, go to step4
  3. $\boldsymbol{Work = Work + Allocation_i}$
     $\boldsymbol{Finish[i] = true}$
     go to step 2
  4. If $\boldsymbol{Finish[i] == false}$, for some $\boldsymbol{i}$, $0 \leq \boldsymbol{i} < \boldsymbol{n}$, then the system is in deadlock state. Moreover, if $\boldsymbol{Finish[i] == false}$, then $\boldsymbol{T_i}$ is deadlocked.
- Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state.

Deadlock Detection

- 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

Deadlock Detection

- Further, if 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 threads T_1, T_2, T_3, and T_4

Deadlock Detection

- 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?
  - Invoke every time a request for allocation cannot be granted immediately
    - ★ Incur considerable overhead
  - Invoke at defined intervals
    - ★ Periodically
    - ★ CPU utilization is low
  - 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

- Two possibilities to recovery from a deadlock
  - Process and Thread Termination
  - Resource Preemption

Recovery from Deadlock

- Process and Thread Termination
  - Abort all deadlocked processes
    - ★ Cost at recomputing the results
  - Abort one process at a time until the deadlock cycle is eliminated
    - ★ Cost at deadlock-detection
    - ★ In which order should we choose to abort?
      1. Priority of the process
      2. Processing time and completion time
      3. Resources the process has used
      4. Resources process needs to complete
      5. The number of processes will need to be terminated

Recovery from Deadlock

- Resource Preemption
  - Preempt resources to other processes until a deadlock is broken
  - Three issues:
    - ★ 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