CSE2008: Operating Systems

L19 L20 L21 & L22 : Deadlocks

VIT-AP UNIVERSITY

Dr. Subrata Tikadar

SCOPE, VIT-AP University

Recap

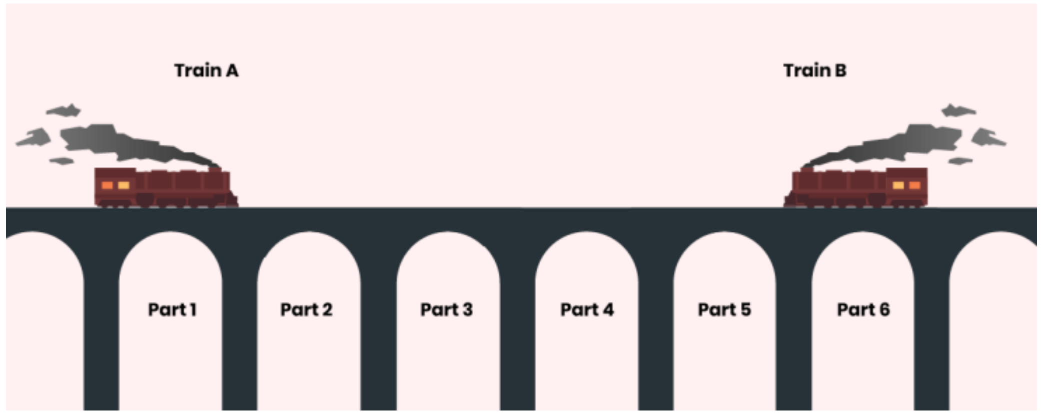
- Introductory Concepts
- Process Fundamentals
- IPC
- CPU Scheduling Algorithms
- Multithreading Concepts
- Synchronization

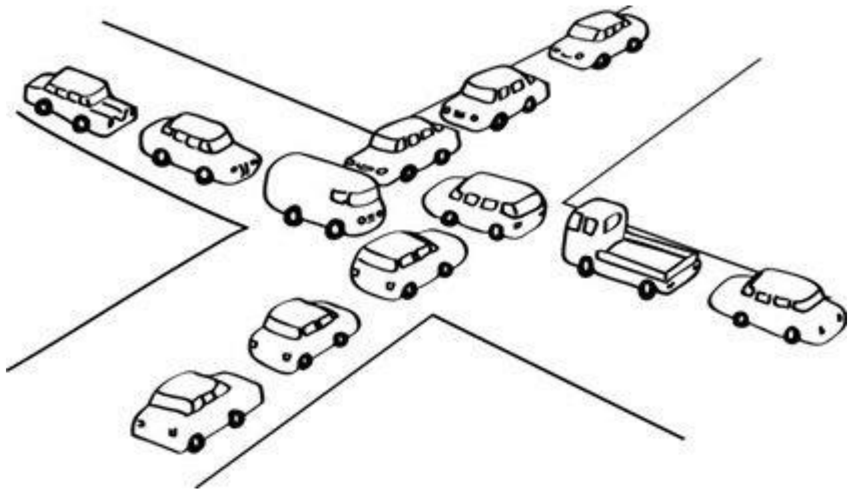
Outline

- Deadlock Concept
- System Model
- Deadlock Characterization
- Dealing with the Deadlock Problem
    - Ignore the Problem Altogether
    - Methods for Handling Deadlocks
        - Deadlock Prevention
        - Deadlock Avoidance
        - Deadlock Detection and Recovery from Deadlock

Deadlock Concept

- *A situation in which every process in a set of processes is waiting for an event that can be caused only by another process in the set*





System Model

- A system consists of a finite number of resources to be distributed among a number of competing processes (/threads)
  - The resources may be classified into several types, each consisting of some number of identical instances

    Example:
    - CPU cycles, files, and I/O devices are different resource types
    - If a system has four CPUs, then the resource type CPU has four instances
  - Resource utilization sequence
    1. **Request.** The thread requests the resource. If the request cannot be granted immediately (for example, if a mutex lock is currently held 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 mutex lock, the thread can access its critical section).
    3. **Release.** The thread releases the resource.

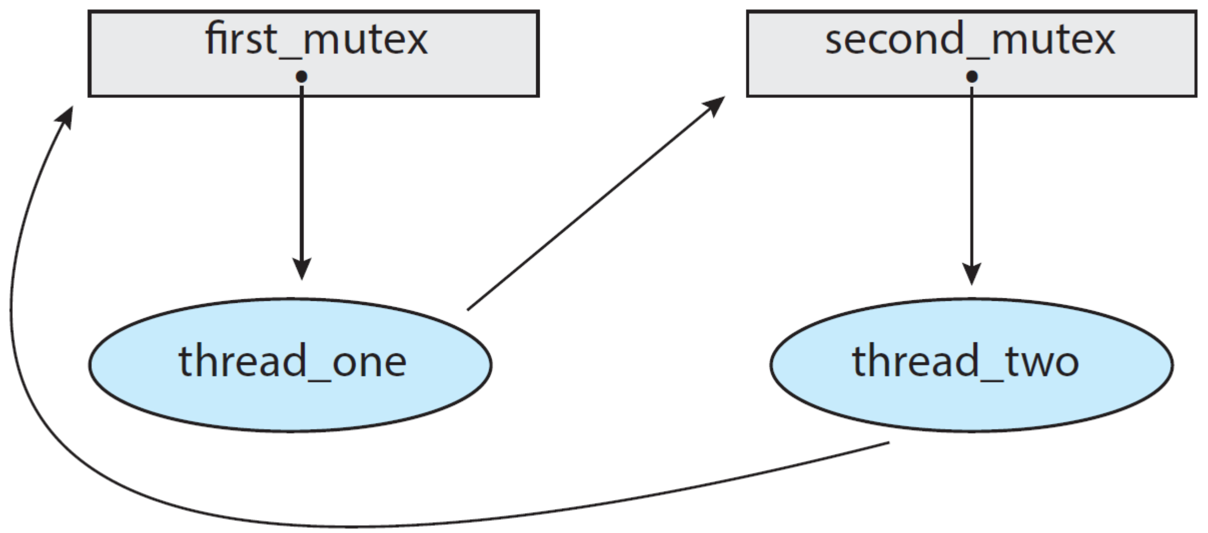
Deadlock Characterization

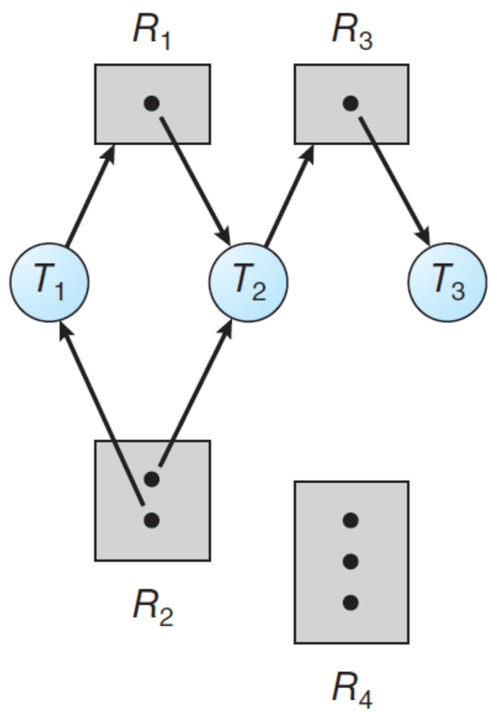
- Necessary Conditions
  1. **Mutual exclusion.** At least one resource must be held in a non-sharable mode; that is, only one thread at a time can use the resource. If another thread requests that resource, the requesting thread must be delayed until the resource has been released.
  2. **Hold and wait.** A thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other threads.
  3. **No preemption.** Resources cannot be preempted; that is, a resource can be released only voluntarily by the thread holding it, after that thread has completed its task.
  4. **Circular wait.** A set $\{T_0, T_1, ..., T_n\}$ of waiting threads must exist 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, and T_n is waiting for a resource held by T_0.

Note: *All these four conditions must hold for a deadlock to occur!*

Deadlock Characterization

- System Resource-Allocation Graph
  - A precise way to describe deadlocks

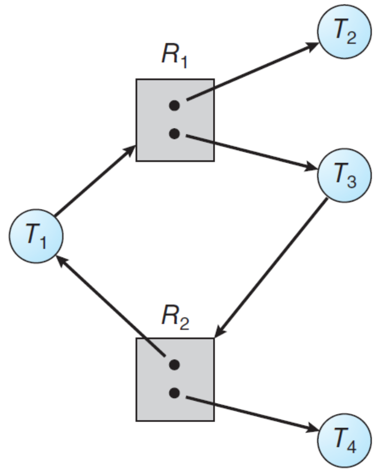


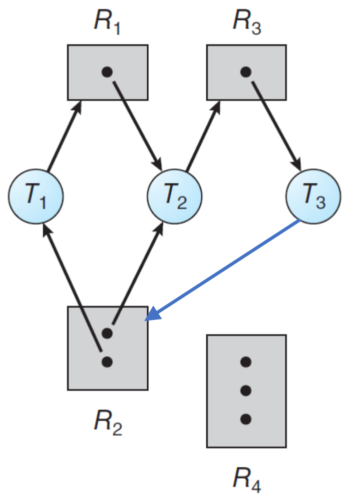


If the graph contains no cycles, then no thread in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

Deadlock Characterization

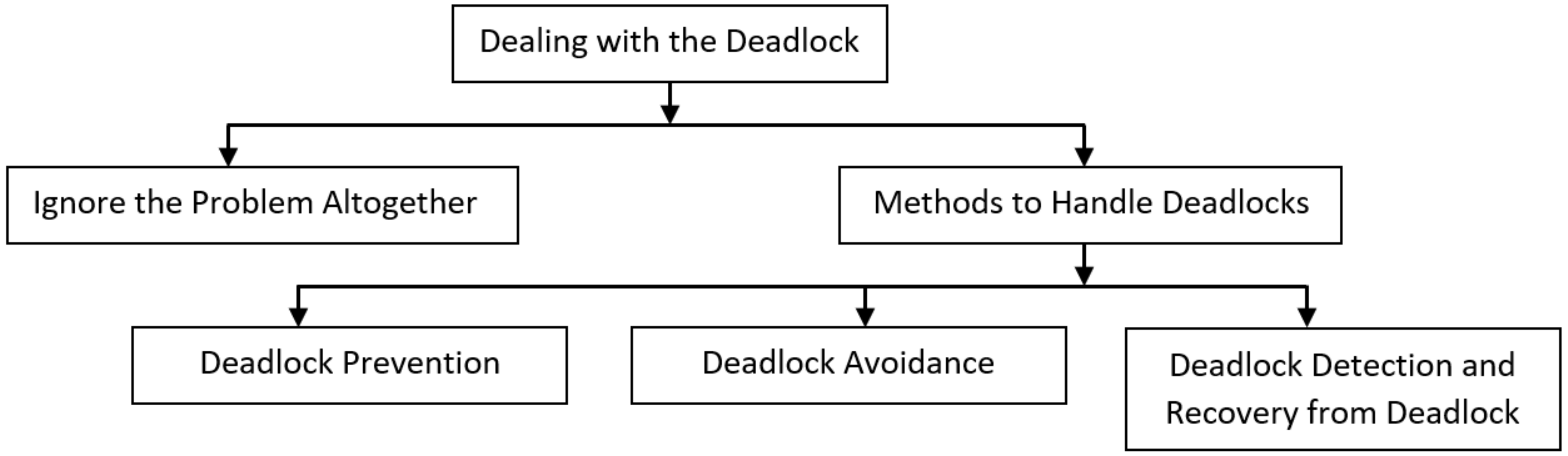
- System Resource-Allocation Graph
  - A precise way to describe deadlocks
    - If the graph contains no cycles, then no thread in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist
      - If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred
      - If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred - a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock





Dealing with the Deadlock Problem

- We can ignore the problem altogether and pretend that deadlocks never occur in the system.
- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.


Dealing with the Deadlock
Ignore the Problem Altogether
Methods to Handle Deadlocks
Deadlock Prevention
Deadlock Avoidance
Deadlock Detection and Recovery from Deadlock

Deadlock Prevention

- Prevention of
    - Mutual Exclusion
    - Hold and Wait
    - No Preemption
    - Circular Wait

Deadlock Prevention

- Prevention of
  - Mutual Exclusion

    Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock (e.g., a read only file)

    Problem: *Some resources are intrinsically nonsharable (a mutex lock cannot be cannot be simultaneously shared by several threads)*

Deadlock Prevention

- Prevention of
  - Hold and Wait
    - We must guarantee that, whenever a thread requests a resource, it does not hold any other resources (***impractical**** *for most applications due to the dynamic nature of requesting resources)*
    - Allow a thread to request resources only when it has none

Problems:

1. *Resource utilization may be low, since resources may be allocated but unused for a long period*
2. *Starvation is possible*

Deadlock Prevention

- Prevention of
  - No Preemption
    - We can use one of the following protocols
      - If a thread is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the thread must wait), then all resources the thread is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the thread is waiting. The thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
      - If a thread requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other thread that is waiting for additional resources. If so, we preempt the desired resources from the waiting thread and allocate them to the requesting thread. If the resources are neither available nor held by a waiting thread, the requesting thread must wait. While it is waiting, some of its resources may be preempted, but only if another thread requests them. A thread can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

***Problem:** Cannot generally be applied to many resources that are mainly responsible for deadlock to occur*

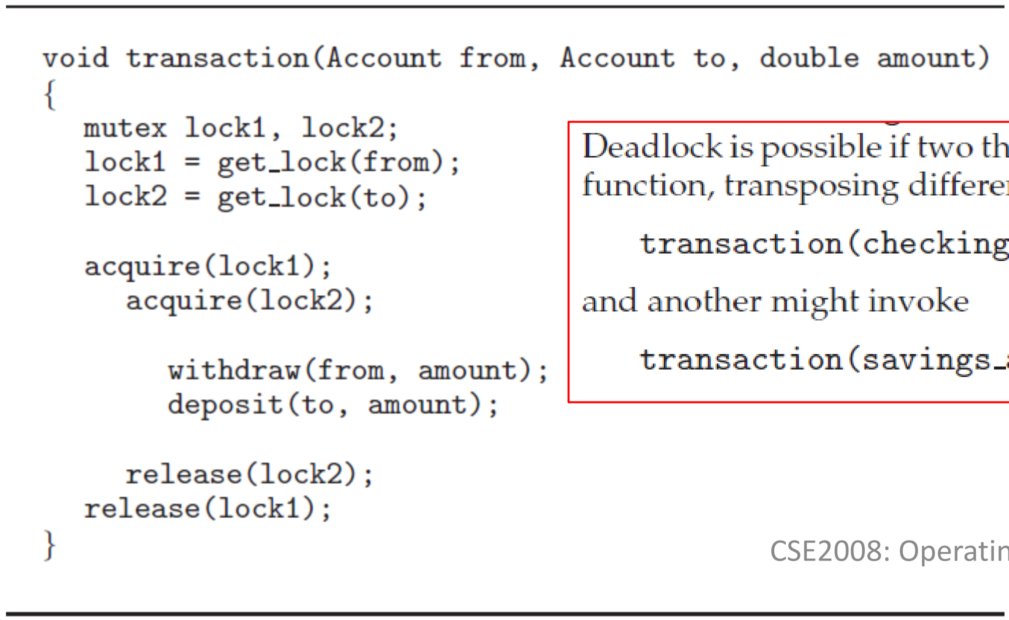
Deadlock Prevention

- Prevention of
  - Circular Wait
    - Impose a total ordering of all resource types and to require that each thread requests resources in an increasing order of enumeration
    - And then we can follow either of these two protocols:
      - Each thread can request resources only in an increasing order of enumeration. That is, a thread can initially request an instance of a resource—say, R_i. After that, the thread can request an instance of resource R_j if and only if $F(R_j) > F(R_i)$.
      - A a thread requesting an instance of resource R_j must have released any resources R_i such that $F(R_i) \geq F(R_j)$.

***Home task:** Prove that if these two protocols are used, then the circular-wait condition cannot hold.*

Deadlock Prevention

- Prevention of
  - Circular Wait
    - Important points to note:
      - Developing an ordering, or hierarchy, does not in itself prevent deadlock. It is up to application developers to write programs that follow the ordering.
      - Imposing a lock ordering does not guarantee deadlock prevention if locks can be acquired dynamically

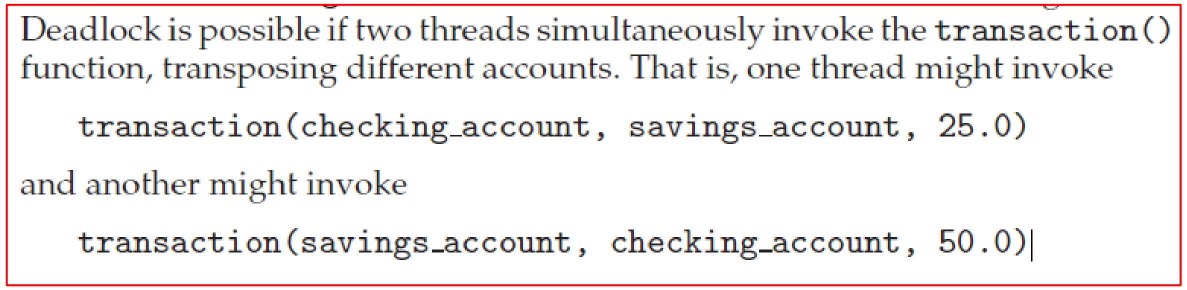


```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);

   acquire(lock1);
      acquire(lock2);

         withdraw(from, amount);
         deposit(to, amount);

      release(lock2);
   release(lock1);
}
```



Deadlock is possible if two threads simultaneously invoke the `transaction()` function, transposing different accounts. That is, one thread might invoke

`transaction(checking_account, savings_account, 25.0)`

and another might invoke

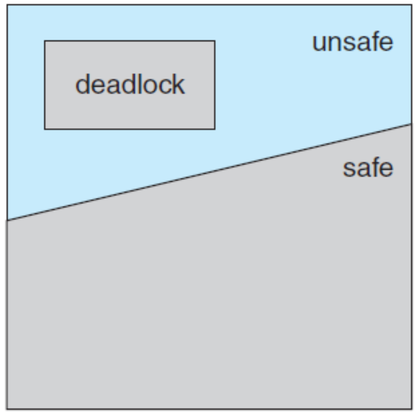
`transaction(savings_account, checking_account, 50.0)`

Deadlock Avoidance

- Safe State
- Resource-Allocation-Graph Algorithm
- Banker’s Algorithm
    - Safety Algorithm
    - Resource-Request Algorithm

Deadlock Avoidance

- 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, ..., T_n> is a safe sequence for the current allocation state if, for each T_i, the resource requests that T_i can still make can be satisfied by the currently available resources plus the resources held by all T_j, with $j < i$.
    - In this situation, if the resources that T_i needs are not immediately available, then T_i can wait until all T_j have finished. When they have finished, T_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When T_i terminates, T_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.



Deadlock Avoidance

- Safe State
  - Example of Safe and Unsafe States
    - Assume that there is a system with **twelve resources** and **three threads**

Thread	Max Need	Current Allocation
T_0	10	5
T_1	4	2
T_2	9	2

Available** = {12 – (5+2+2)} = **3

The system is safe.
This is because we have the Safe Sequence <T_1, T_0, T_2>

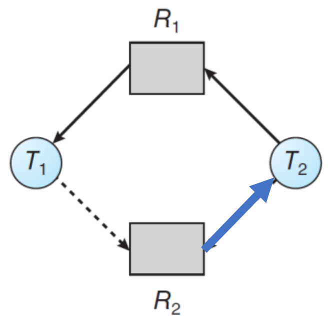
Thread	Max Need	Current Allocation
T_0	10	5
T_1	4	2
T_2	9	**3**

Available** = {12 – (5+2+2)} = **3

The system is not safe.
This is because we have no Safe Sequence

Deadlock Avoidance

- Resource-Allocation-Graph Algorithm
  - In addition to the request and assignment edges already described, we introduce a new type of edge, called a ***claim edge***
    - A claim edge $T_i \rightarrow R_j$ indicates that thread T_i may request resource R_j 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 T_i requests resource R_j, the claim edge $T_i \rightarrow R_j$ is converted to a request edge. Similarly, when a resource R_j is released by T_i, the assignment edge $R_j \rightarrow T_i$ is reconverted to a claim edge $T_i \rightarrow R_j$
    - If thread T_i requests resource R_j. The request can be granted only if converting the request edge $T_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow T_i$ does not result in the formation of a cycle in the resource-allocation graph.
      - We check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of threads in the system.



Deadlock Avoidance

- Banker's Algorithm
  - Safety Algorithm
  - Resource-Request Algorithm

Deadlock Avoidance

- Banker’s Algorithm
  - Data structures that are required to maintain to implement banker’s algo
    - **Available.** A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type R_j are available.
    - **Max.** An n × m matrix defines the maximum demand of each thread. If Max[i][j] equals k, then thread T_i may request at most k instances of resource type R_j.
    - **Allocation.** An n × m matrix defines the number of resources of each type currently allocated to each thread. If Allocation[i][j] equals k, then thread T_i is currently allocated k instances of resource type R_j.
    - **Need.** An n × m matrix indicates the remaining resource need of each thread. If Need[i][j] equals k, then thread T_i may need k more instances of resource type R_j to complete its task. Note that Need[i][j] equals Max[i][j] − Allocation[i][j].

Deadlock Avoidance

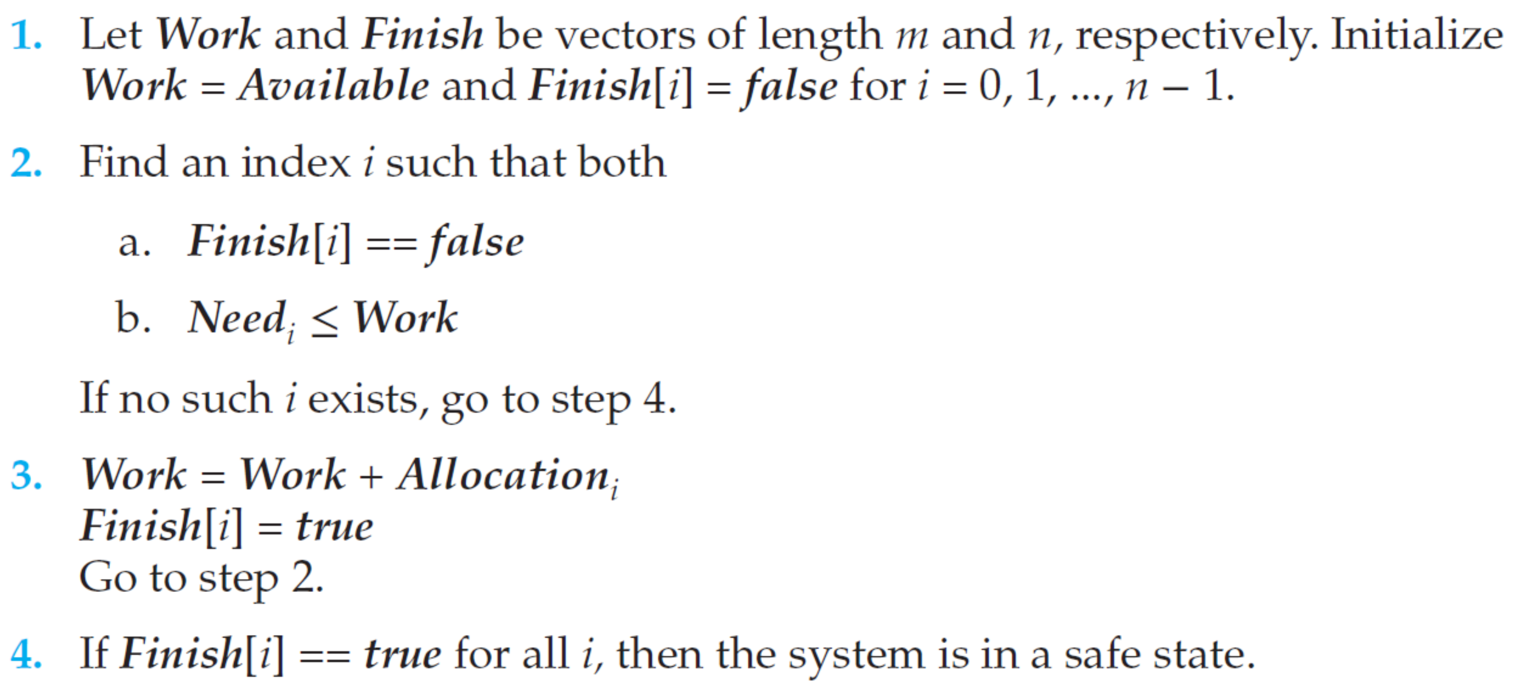
- Banker's Algorithm
  - Data structures that are required to maintain to implement banker's algo
    - **Available.** A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type R_j are available.
    - **Max.** An n × m matrix defines the maximum demand of each thread. If Max[i][j] equals k, then thread T_i may request at most k instances of resource type R_j.
    - **Allocation.** An n × m matrix defines the number of resources of each type currently allocated to each thread. If Allocation[i][j] equals k, then thread T_i is currently allocated k instances of resource type R_j.
    - **Need.** An n × m matrix indicates the remaining resource need of each thread. If Need[i][j] equals k, then thread T_i may need k more instances of resource type R_j to complete its task. Note that Need[i][j] equals Max[i][j] − Allocation[i][j].

	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	

	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

Deadlock Avoidance

- Banker's Algorithm
  - Safety Algorithm



1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work* = *Available* and *Finish*[i] = *false* for $i = 0, 1, ..., n - 1$.
2. Find an index i such that both
   a. *Finish*[i] == *false*
   b. $Need_i \leq Work$

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

Deadlock Avoidance

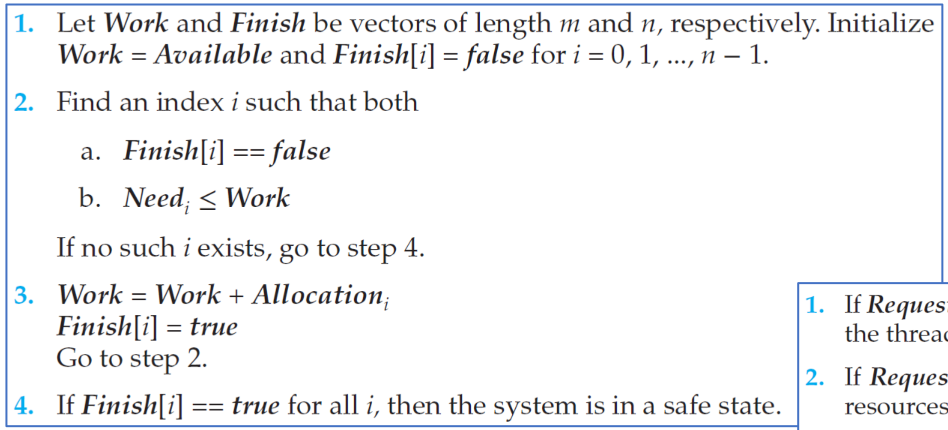
- Banker's Algorithm
  - Resource-Request Algorithm

1. If $\boldsymbol{Request}_i \leq \boldsymbol{Need}_i$, go to step 2. Otherwise, raise an error condition, since the thread has exceeded its maximum claim.
2. If $\boldsymbol{Request}_i \leq \boldsymbol{Available}$, go to step 3. Otherwise, T_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to thread T_i by modifying the state as follows:

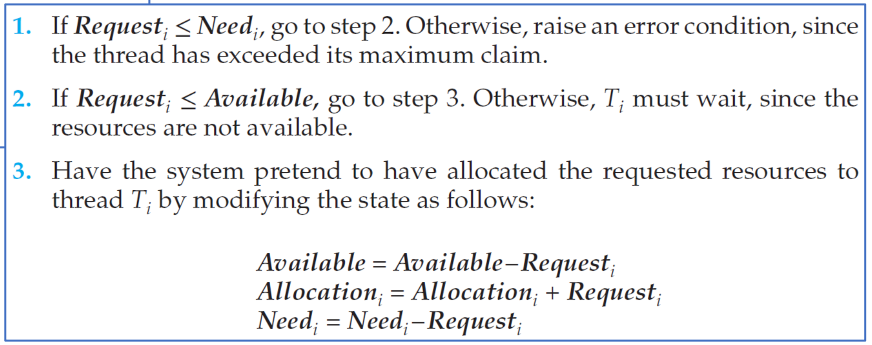
$$\begin{aligned}&\boldsymbol{Available} = \boldsymbol{Available} - \boldsymbol{Request}_i\\&\boldsymbol{Allocation}_i = \boldsymbol{Allocation}_i + \boldsymbol{Request}_i\\&\boldsymbol{Need}_i = \boldsymbol{Need}_i - \boldsymbol{Request}_i\end{aligned}$$

Deadlock Avoidance

- Banker's Algorithm
  - Example

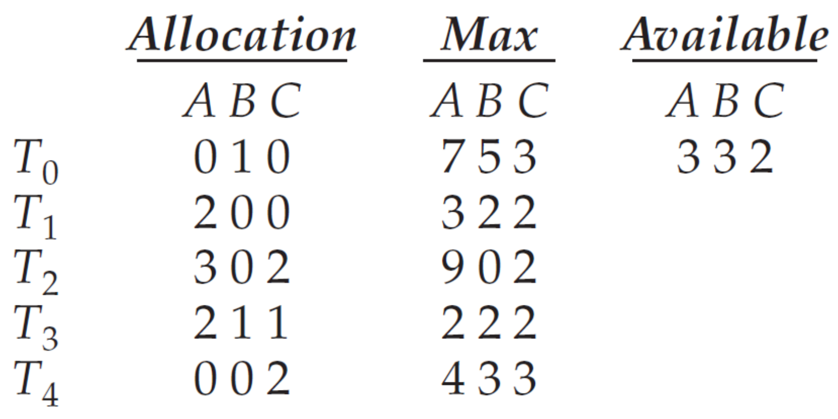


```
1. Let Work and Finish be vectors of length m and n, respectively. Initialize
   Work = Available and Finish[i] = false for i = 0, 1, ..., n − 1.
2. Find an index i such that both
      a. Finish[i] == false
      b. Need_i ≤ Work
   If no such i exists, go to step 4.
3. Work = Work + Allocation_i
   Finish[i] = true
   Go to step 2.
4. If Finish[i] == true for all i, then the system is in a safe state.
```

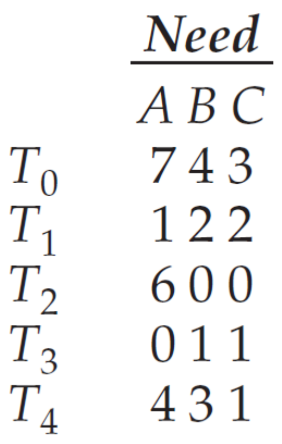


1. If $Request_i \le Need_i$, go to step 2. Otherwise, raise an error condition, since the thread has exceeded its maximum claim.
2. If $Request_i \le Available$, go to step 3. Otherwise, T_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to thread T_i by modifying the state as follows:

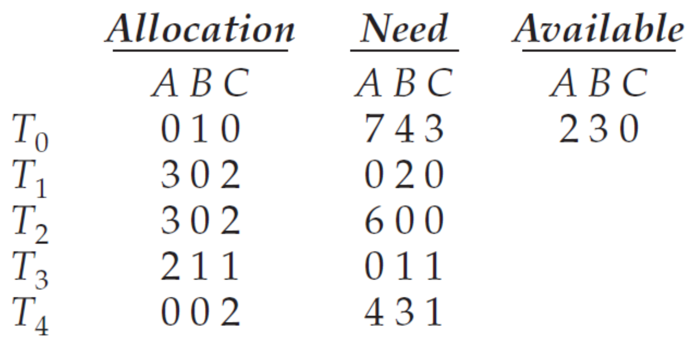
$$\begin{aligned} Available &= Available - Request_i \\ Allocation_i &= Allocation_i + Request_i \\ Need_i &= Need_i - Request_i \end{aligned}$$



	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	



	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



	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	

Deadlock Avoidance

- Banker's Algorithm

Exercise:

Consider the following snapshot of a system and answer the following questions using the banker's algorithm:

a. Illustrate that the system is in a safe state by demonstrating an order in which the threads may complete.

b. If a request from thread T_4 arrives for (2, 2, 2, 4), can the request be granted immediately?

c. If a request from thread T_2 arrives for (0, 1, 1, 0), can the request be granted immediately?

d. If a request from thread T_3 arrives for (2, 2, 1, 2), can the request be granted immediately?

Allocation	*Max*	*Available*
A B C D	*A B C D*	*A B C D*
3 1 4 1	6 4 7 3	2 2 2 4
2 1 0 2	4 2 3 2	
2 4 1 3	2 5 3 3	
4 1 1 0	6 3 3 2	
2 2 2 1	5 6 7 5	

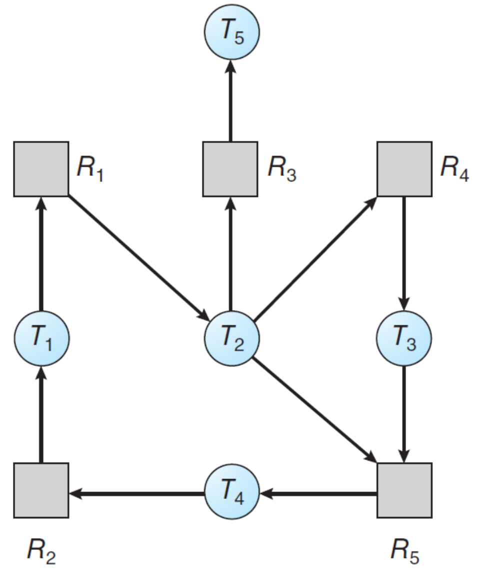
Deadlock Detection and Recovery from Deadlock

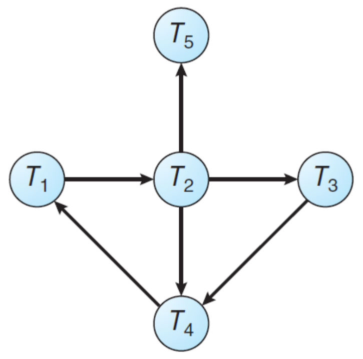
- Deadlock Detection
- Recovery from Deadlock

Deadlock Detection and Recovery from Deadlock

- Deadlock Detection
  - Single Instance of Each Resource Type
    - We can 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

Example:





To detect deadlocks, the system needs to ***maintain*** *the wait for graph and periodically* ***invoke an algorithm*** *that searches for a cycle in the graph.*

Deadlock Detection and Recovery from Deadlock

- Deadlock Detection
  - Several Instances of a Resource Type
    - The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm:
      - **Available.** A vector of length m indicates the number of available resources of each type.
      - **Allocation.** An n × m matrix defines the number of resources of each type currently allocated to each thread.
      - **Request.** An n × m matrix indicates the current request of each thread. If Request[i][j] equals k, then thread T_i is requesting k more instances of resource type R_j.

Deadlock Detection and Recovery from Deadlock

- Deadlock Detection
  - Several Instances of a Resource Type
    - The Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work* = *Available*. For $i = 0, 1, \ldots, n-1$, 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, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then thread T_i is deadlocked.

	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	

	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

Deadlock Detection and Recovery from Deadlock

- Recovery from Deadlock
  - Process and Thread Termination
  - Resource Preemption

Deadlock Detection and Recovery from Deadlock

- Recovery from Deadlock
  - Process and Thread Termination
    - **Abort all deadlocked processes.** This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
    - **Abort one process at a time until the deadlock cycle is eliminated.** This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Deadlock Detection and Recovery from Deadlock

- Recovery from Deadlock
  - Resource Preemption
    - Three issues need to be addressed to deal with deadlock by resource preemption
      1. **Selecting a victim.** Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
      2. **Rollback.** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.
      3. **Starvation.** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

Reference

- Abraham Silberschatz , Peter B. Galvin, Greg Gagne, “Operating System Concepts”, Addison Wesley, 10th edition, 2018
    - Chapter 8: Section 8.1 – 8.8

Next

- **Module 4:** Memory Management

Thank You