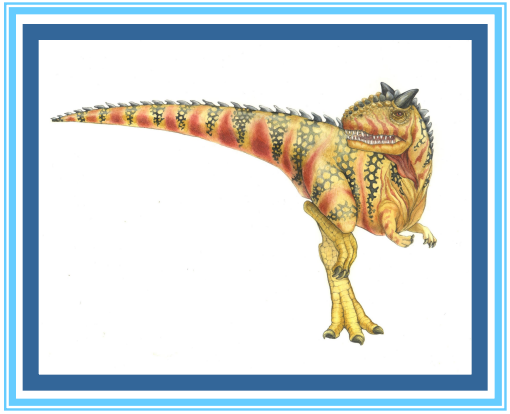
Chapter 7: Deadlocks



Chapter 8: Deadlocks

- 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

- In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.
- System consists of finite number of resources to be distributed among a number of competing processes. Eg. *CPU cycles, memory space, I/O devices* (such as printers and DVD drives).
- The resources may be partitioned into several types (or classes), each consisting of some number of identical instances . **Resource types $R_1, R_2, \ldots, R_m$. Each resource type R_i has W_i instances.**
- Under the normal mode of operation, a process may utilize a resource in only the following sequence:
  - **Request**. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
  - **Use**. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
  - **Release**. The process releases the resource.

System Calls

- The request and release of resources may be through system calls
- For instance
  - Physical resources like the **request() and release() device** and **allocate() and free() memory** system calls.
  - Logical resources like **open() and close() file**, the request and release of semaphores can be accomplished through the **wait() and signal() operations on semaphores** and **acquire() and release() of a mutex lock**.
  - For each use of a kernel-managed resource by a process or thread, the operating system checks to make sure that the process has requested and has been allocated the resource.
- A system table records whether each resource is free or allocated. For each resource that is allocated, the table also records the process to which it is allocated.
- If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.

Deadlock in Multithreaded Application

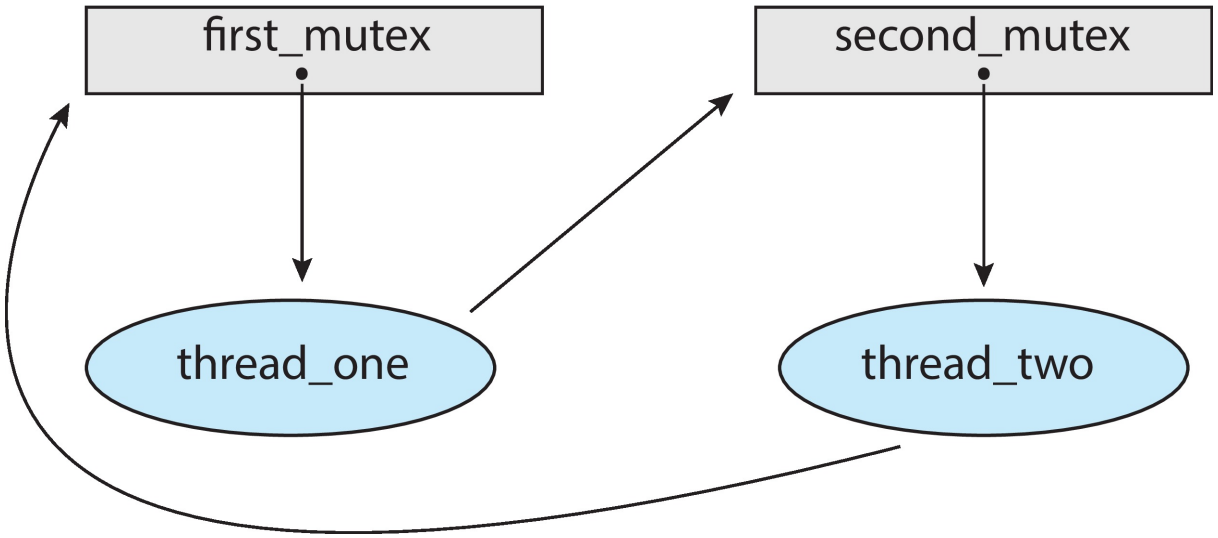
- Two mutex locks are created and initialized:

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

```
/* 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);
}
```

Deadlock in Multithreaded Application

- Deadlock is possible if thread 1 acquires **`first_mutex`** and thread 2 acquires **`second_mutex`**. Thread 1 then waits for **`second_mutex`** and thread 2 waits for **`first_mutex`**.
- Can be illustrated with a **resource allocation graph**:



Deadlock Characterization

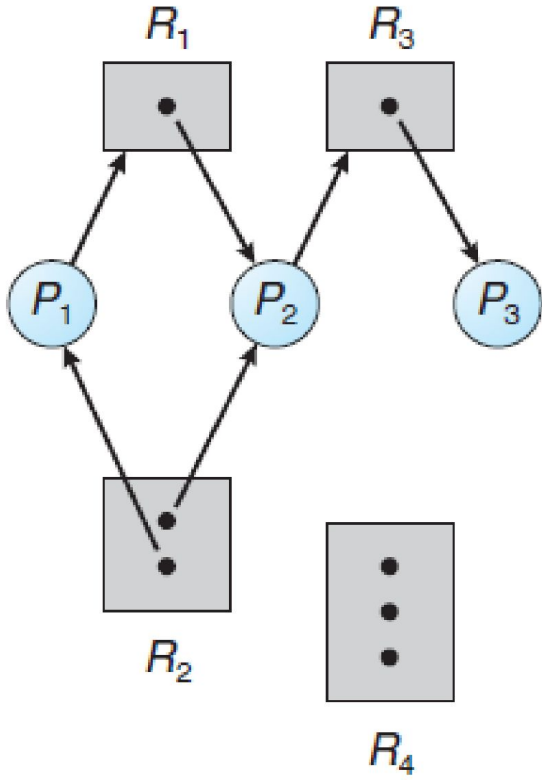
- In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.
- Deadlock can arise if four conditions hold **simultaneously**.
  - **Mutual exclusion**. At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
  - **Hold and wait.** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
  - **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its tasks.
  - **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.

Resource-Allocation Graph

- Deadlocks is described in terms of a directed graph called a **system resource-allocation graph**
- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the active processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$ - It signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource.
- **assignment edge** – directed edge $R_j \rightarrow P_i$ - It signifies that an instance of resource type R_j has been allocated to process Pi .
- Pi as a circle and each resource type Rj as a rectangle. resource type Rj may have more than one instance, each such instance is represented as a dot within the rectangle

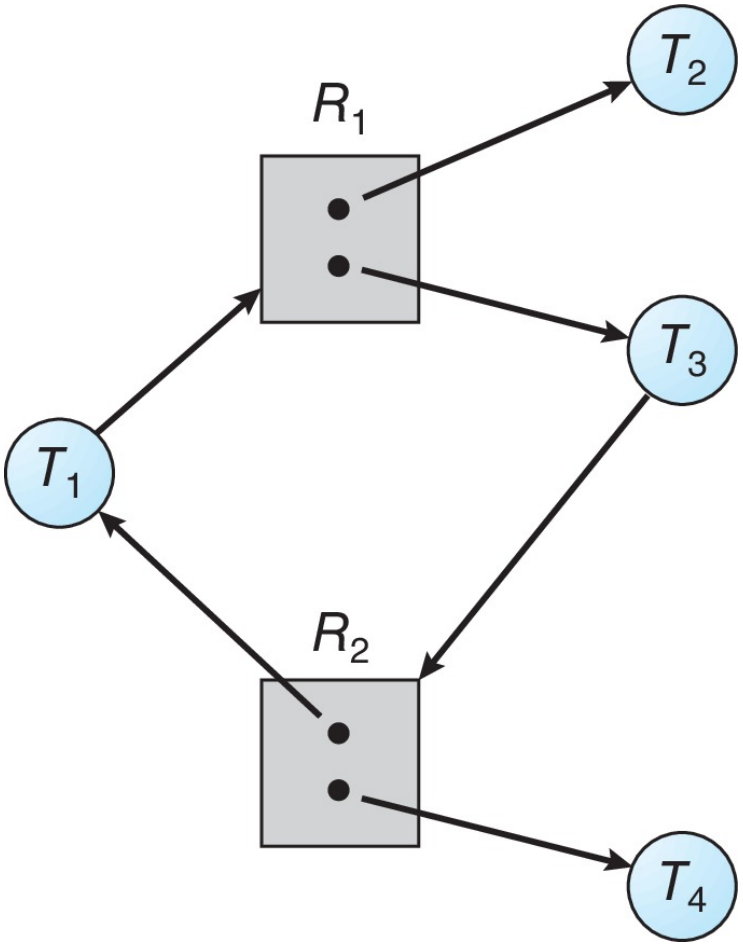
Resource Allocation Graph Example

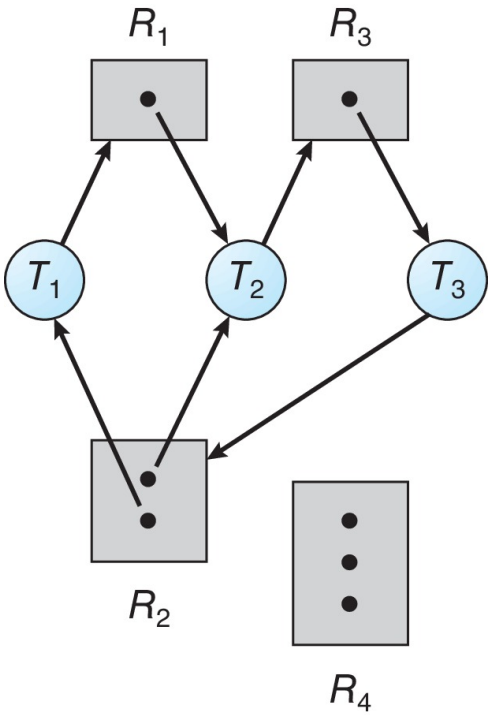
- The sets P, R, and E:
  - $P = \{P1, P2, P3\}$
  - $R = \{R1, R2, R3, R4\}$
  - $E = \{P1 \rightarrow R1, P2 \rightarrow R3, R1 \rightarrow P2, R2 \rightarrow P2, R2 \rightarrow P1, R3 \rightarrow P3\}$
- Resource Type Instances
  - One instance of R1
  - Two instances of R2
  - One instance of R3
  - Three instance of R4
- Process states:
  - T1 holds one instance of R2 and is waiting for an instance of R1
  - T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
  - T3 is holds one instance of R3



Resource Allocation Graph With A Deadlock

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.





Suppose that process P_3 requests an instance of resource R2

Graph With A Cycle But No Deadlock

If graph contains no cycles ⇒ no deadlock

If graph contains a cycle ⇒

if only one instance per resource type, then deadlock - In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

if several instances per resource type, *possibility of deadlock* - In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

*In summary, if a resource-allocation graph does not have a cycle, then the system is **not** in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.*

Methods for Handling Deadlocks

- Deadlock problem can be dealt with one of three Ways
  - We can use a protocol to prevent or avoid deadlocks, ensuring that the system will ***never*** enter a deadlocked state.
    - **Deadlock prevention** - provides a set of methods to ensure that at least one of the necessary conditions cannot hold.
    - **Deadlock avoidance** - requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime. With this additional knowledge, the operating system can decide for each request whether or not the process should wait.
  - We can allow the system to enter a deadlocked state, detect it, and recover.
  - Ignore the problem altogether and pretend that deadlocks never occur in the system.
- Before proceeding, we should mention that some researchers have argued that none of the basic approaches alone is appropriate for the entire spectrum of resource-allocation problems in operating systems. The basic approaches can be combined, however, allowing us to select an optimal approach for each class of resources in a system.

Deadlock Prevention

- By ensuring that at least one of four conditions cannot hold, we can ***prevent*** the occurrence of a deadlock.

- **Mutual Exclusion** – The mutual exclusion condition must hold. That is, at least one resource must be non-sharable. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock.
  - Read-only files is an example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. **A process never needs to wait for a sharable resource**.

In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non sharable. For example, a mutex lock cannot be simultaneously shared by several processes not required for sharable resources (e.g., read-only files);

Deadlock Prevention (Cont.)

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.
  - An alternative protocol allows a process to request resources only when it has none. - A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.
  - Disadvantages –
    - resource utilization may be low, since resources may be allocated but unused for a long period.
    - starvation - A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

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 process is waiting
- A process 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.
- This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as mutex locks and semaphores.

Deadlock Prevention (Cont.)

Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

- To illustrate, we let $R = \{R1, R2, ..., Rm\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function $F: R \rightarrow N$, where N is the set of natural numbers. (Eg.: F(tape drive) = 1)
- protocol to prevent deadlocks:
  - Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type - say, Ri . After that, the process can request instances of resource type Rj if and only if $F(Rj) > F(Ri)$.
  - Alternatively, we can require that a process requesting an instance of resource type R_j must have released any resources R_i such that $F(R_i) \geq F(R_j)$
  - Note - if several instances of the same resource type are needed, a ***single*** request for all of them must be issued.
- Also note that the function F should be defined according to the normal order of usage of the resources in a system Eg.: the tape drive is usually needed before the printer, it would be reasonable to define F(tape drive)<F(printer).

- Although ensuring that resources are acquired in the proper order is the responsibility of application developers, certain software can be used to verify that locks are acquired in the proper order and to give appropriate warnings when locks are acquired out of order and deadlock is possible.
- One lock-order verifier, which works on BSD versions of UNIX such as FreeBSD, is known as **witness**. Witness uses mutual-exclusion locks to protect critical sections. It works by dynamically maintaining the relationship pof lock orders in a system.
- **`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);
}
```

Deadlock Avoidance done through notes

Requires that the system has some additional ***a priori*** information available

- Simplest and most useful model requires that each process 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

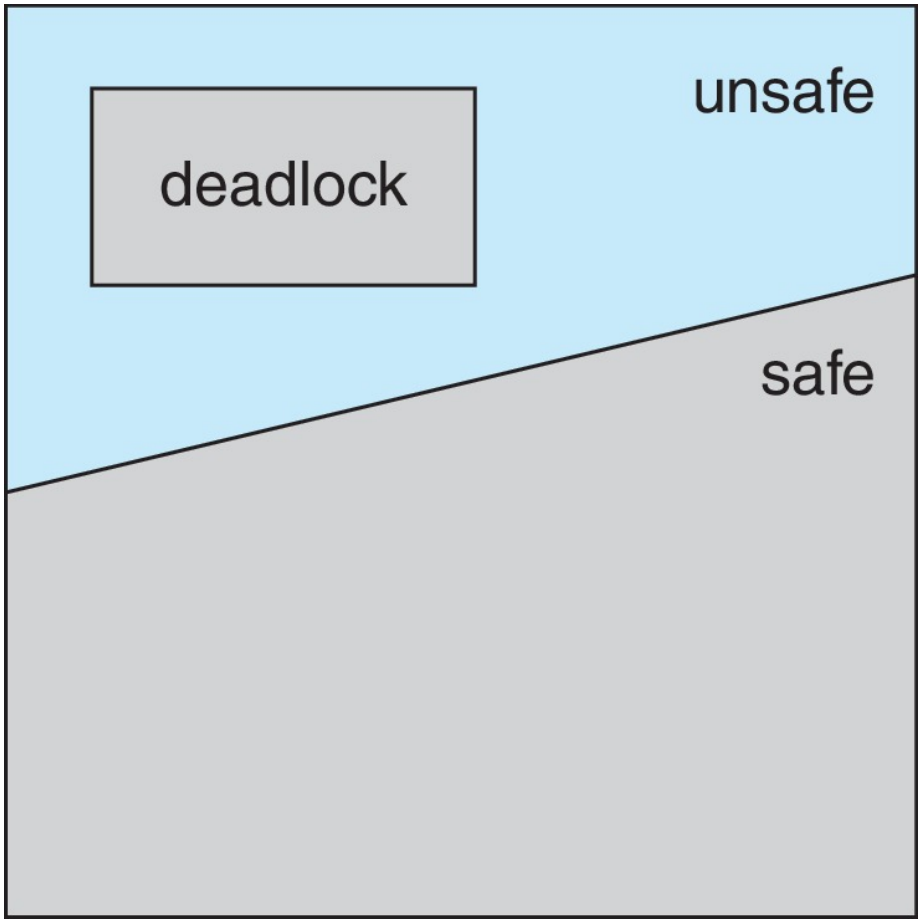
Safe State

- When a 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 $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- 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, Deadlock State



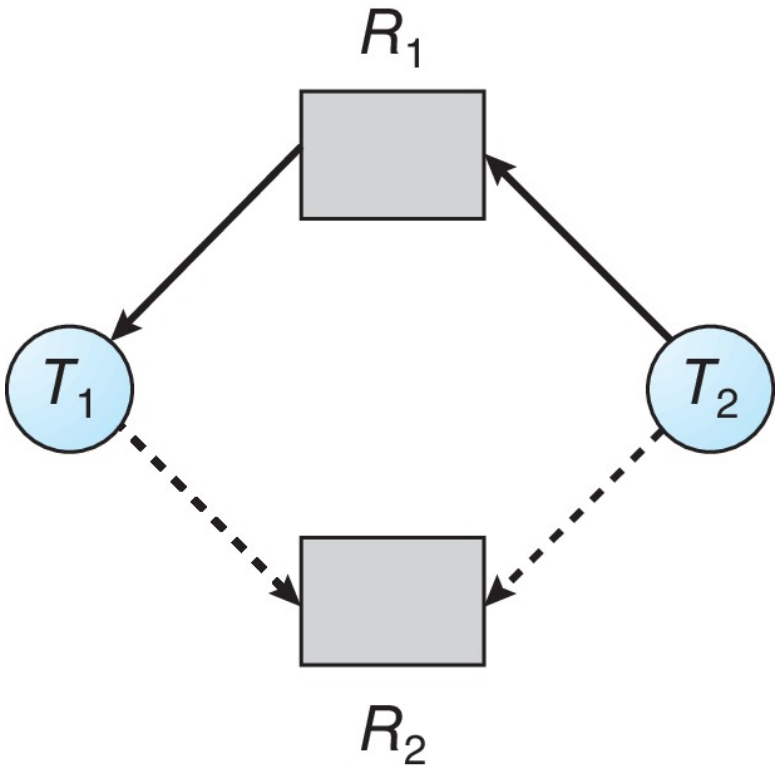
Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the Banker's Algorithm

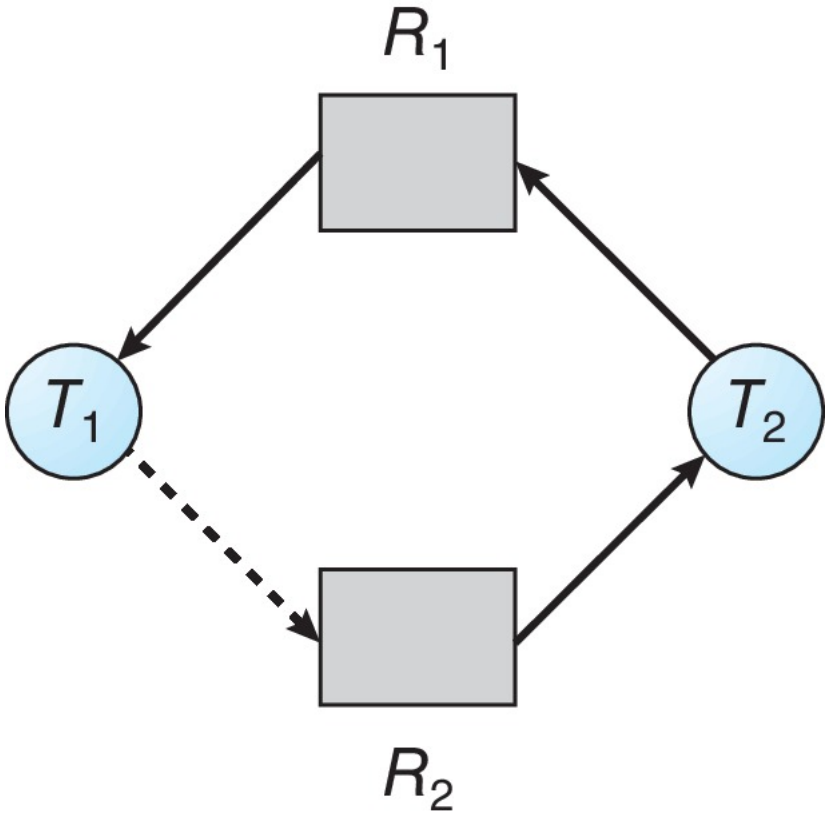
Resource-Allocation Graph Scheme

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

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- 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

Banker's Algorithm

- Multiple instances of resources
- Each process must a priori claim maximum use
- When a process 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 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**: $n \times m$ matrix. If $Max\ [i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: $n \times m$ matrix. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: $n \times m$ matrix. If $Need[i,j] = k$, then P_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) $Need_i \leq Work$

   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

Resource-Request Algorithm for Process P_i

$Request_i$ = request vector for process P_i. If $Request_i[j] = k$ then process P_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 process 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:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe ⇒ the resources are allocated to P_i
- If unsafe ⇒ P_i must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_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*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_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
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria

Example: P_1 Request (1,0,2)

- 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*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < $\boldsymbol{P_1}$, $\boldsymbol{P_3}$, $\boldsymbol{P_4}$, $\boldsymbol{P_0}$, $\boldsymbol{P_2}$> satisfies safety requirement
- Can request for additional (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for additional (0,2,0) by $\boldsymbol{P_0}$ be granted?

Deadlock Detection

- If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.
- Allow system to enter deadlock state
- Detect the deadlock using an algorithm
- Recover from the deadlock
- A detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm but also the potential losses inherent in recovering from a deadlock.

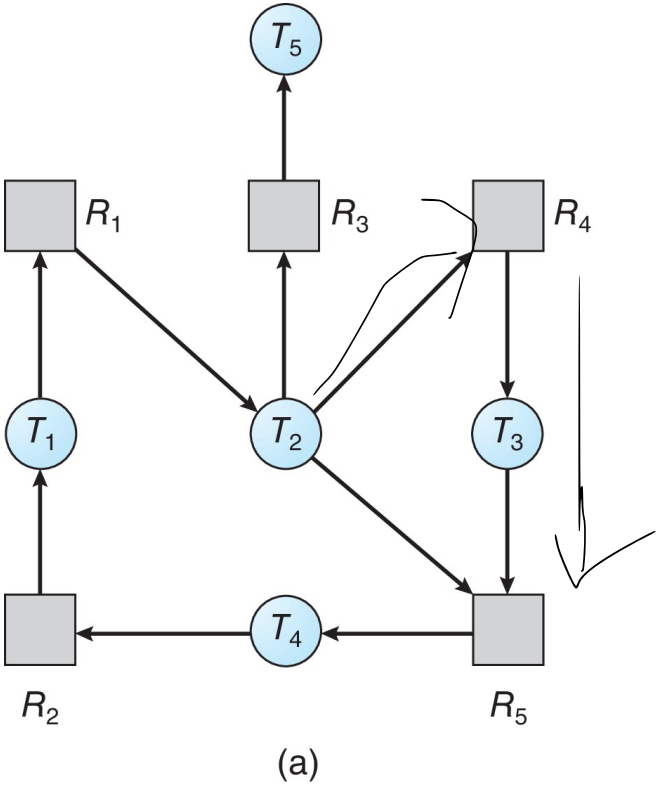
Single Instance of Each Resource Type

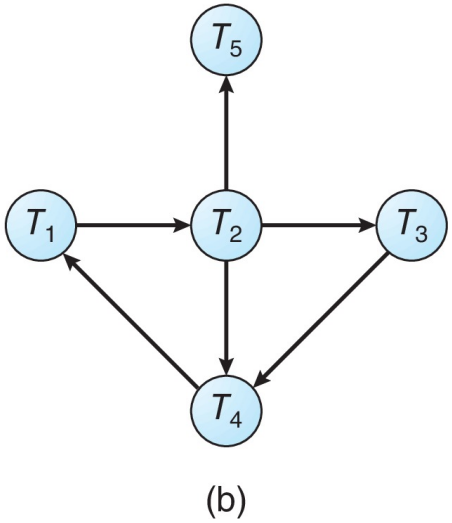
- Define an Deadlock Detection algorithm that uses **wait-for** graph - a variant of RAG
  - removing the resource nodes and collapsing the appropriate edges.
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j, to release a resource that *Pi* needs.
    - the corresponding RAG contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$

- ***Maintain*** the *wait-for* graph and 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 n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph





Resource-Allocation Graph

Corresponding wait-for graph

Several Instances of a Resource Type

- The algorithm employs several time-varying data structures
  - **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 process
  - **Request**: An ***n*** x ***m*** matrix indicates the current request of each process. If ***Request*** **[*i*][*j*] = *k***, then process P_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***, respectively Initialize:
   (a) ***Work = Available***
   (b) For ***i* = 1,2, …, *n***, if ***Allocation*$_i$ ≠ 0**, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***
2. Find an index ***i*** such that both:
   (a) ***Finish*[*i*] == *false***
   (b) ***Request*$_i$ ≤ *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] == false***, for some ***i***, $1 \le i \le n$, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then P_i is deadlocked

Algorithm requires an order of O($m \times n^2$) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes P_0 through P_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*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Example (Cont.)

- P_2 requests an additional instance of type C

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How ***many*** processes will be affected / need to be rolled back by deadlock when it happens? - one for each disjoint cycle
- If deadlocks occur frequently, then the detection algorithm should be invoked frequently.
  - Resources allocated to deadlocked processes will be idle
  - the number of processes involved in the deadlock cycle may grow.
- In the extreme, then, we can invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately. This request may be the final request that completes a chain of waiting processes.
- A less expensive alternative is simply to invoke the algorithm at defined intervals—for example, once per hour or whenever CPU utilization drops below 40 percent. (A deadlock eventually cripples system throughput and causes CPU utilization to drop.)
- 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 processes "caused" the deadlock.

Recovery from Deadlock: Process Termination

- To eliminate deadlocks by aborting a process, we use one of two methods –
  - **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.
- If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated that will incur the minimum cost

Process Termination contd.

- In which order should we choose to abort?
  - What the priority of the process is
  - How long the process has computed and how much longer the process will compute before completing its designated task
  - How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
  - How many more resources the process needs in order to complete
  - How many processes will need to be terminated
  - Whether the process is interactive or batch

Recovery from Deadlock: Resource Preemption

- To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. For this following three issues need to be addressed:
  - **Selecting a victim**. Which resources and which processes are to be preempted? As in process termination, determine the order of preemption to minimize cost - the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
  - **Rollback**. If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution - 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 – restart the process. **Although it is more effective to roll back the process only as far as necessary to break the deadlock -** requiring the system to keep more information about the state of all running processes.
  - **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?

End of Chapter 8