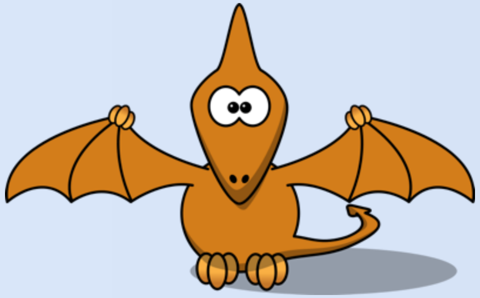
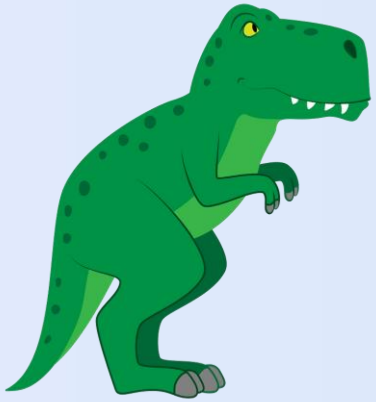


Chapter 8.

Deadlocks



Operating System Concepts (10th Ed.)

8.1 System Model

- A ***deadlock*** is
  - 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*.
  - a situation in which
    - a *waiting thread* (or *process*) can *never again change state*,
    - because the *resources* it has requested
    - are held *by other waiting threads* (or *processes*).

8.1 System Model

- Let us consider a system
  - consisting of *a finite number of resources*
    - to be distributed among a number of *competing threads*.
  - ***Resource types*** consist of
    - some number of ***identical instances***.
    - e.g., CPU cycles, files, and I/O devices(such as printers, drives, etc.)
  - If a thread requests an *instance* of a *resource type*,
    - the allocation of ***any instance*** should *satisfy* the request.
  - A thread may utilize a resource as follows:
    - ***Request*** – ***Use*** – ***Release***.

8.2 Deadlock in Multithreaded Applications

- How can a deadlock occur?

```
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);
}
```

Figure 8.1 *Deadlock example.*

8.3 Deadlock Characterization

- Four Necessary Conditions:
  1. **Mutual Exclusion**:
     - *At least* one resource *is held* in a *non-sharable* mode.
  2. **Hold and Wait**:
     - A thread *holds at least* one resource and *waiting to acquire* additional resources *held by* other threads.
  3. **No preemption**:
     - Resources *cannot* be *preempted*.
  4. **Circular Wait**:
     - *A set of waiting threads* exist such that the *dependency graph* of waiting is *circular*.

8.3 Deadlock Characterization

- **Resource-Allocation Graph**:
  - is a *directed graph* to describe deadlocks more precisely.
  - consists of a set of vertices V and a set of edges E.
  - Two different node types of V:
    - $T = \{T_1, T_2, \cdots, T_n\}$: the set of all the *active threads* in the system.
    - $R = \{R_1, R_2, \cdots, T_m\}$: the set of all the *resource types* in the system.
  - A directed edge: $T_i \rightarrow R_j$ (**request edge**)
    - signifies that a thread T_i has *requested* an instance of R_j.
  - A directed edge: $R_j \rightarrow T_i$ (**assignment edge**)
    - signifies that an instance of R_j has been *allocated to* a thread T_i.

8.3 Deadlock Characterization

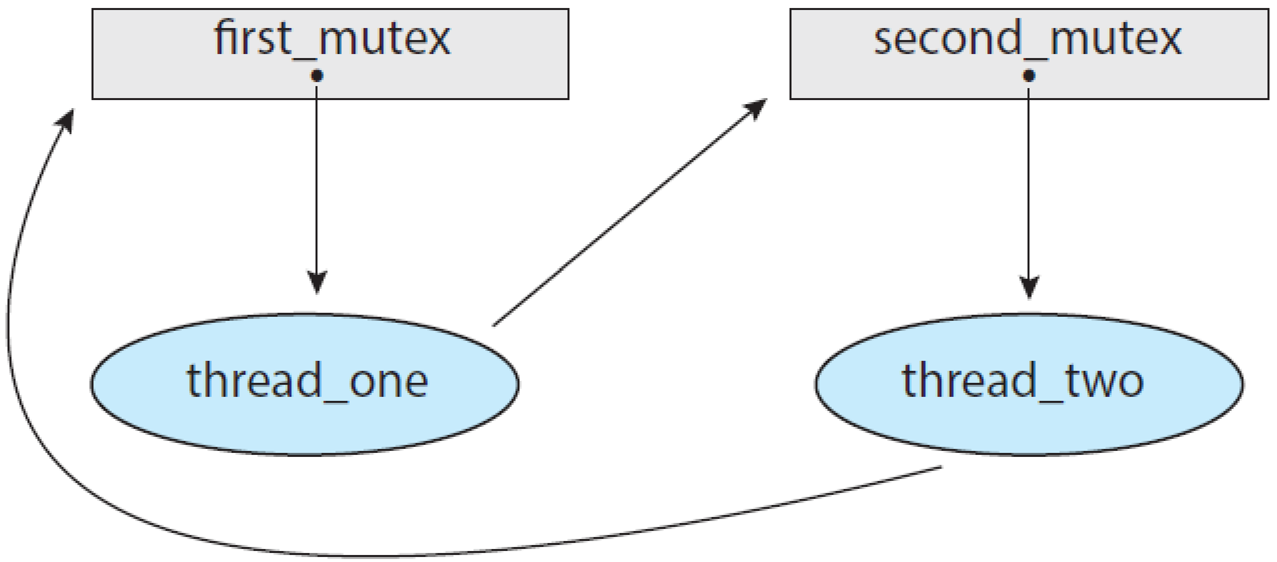
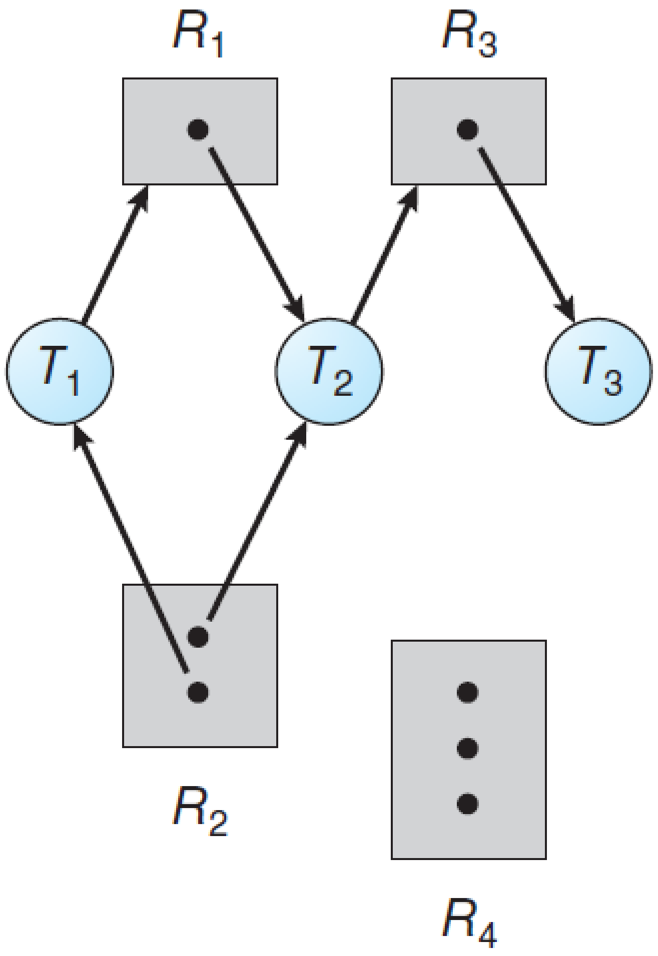


Figure 8.3 *Resource-allocation graph for program in* **Figure 8.1.**

8.3 Deadlock Characterization



- $T = \{T_1, T_2, T_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$

Figure 8.4 *Resource-allocation graph.*

8.3 Deadlock Characterization

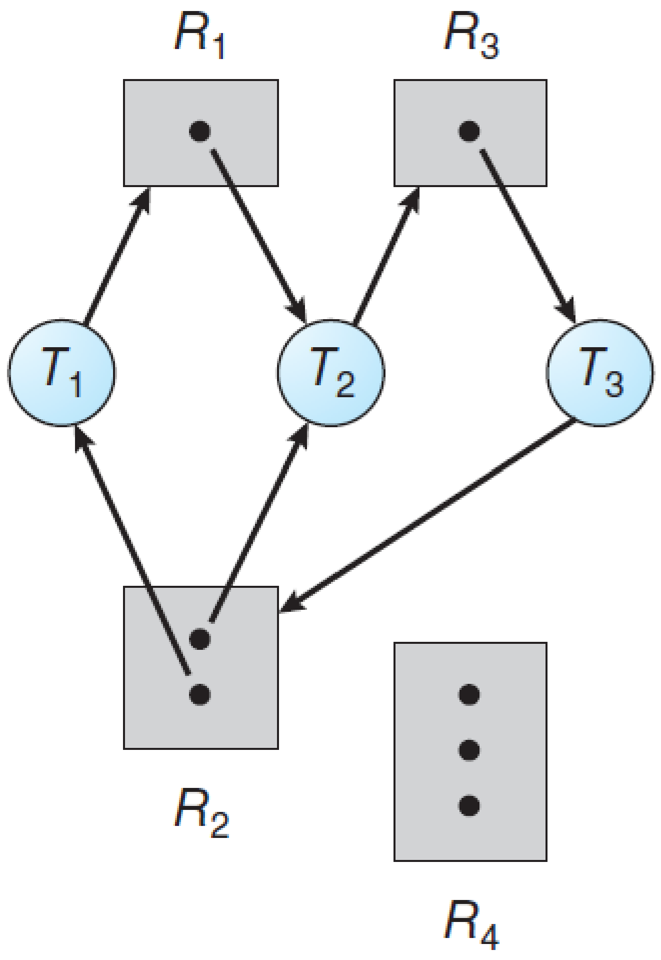
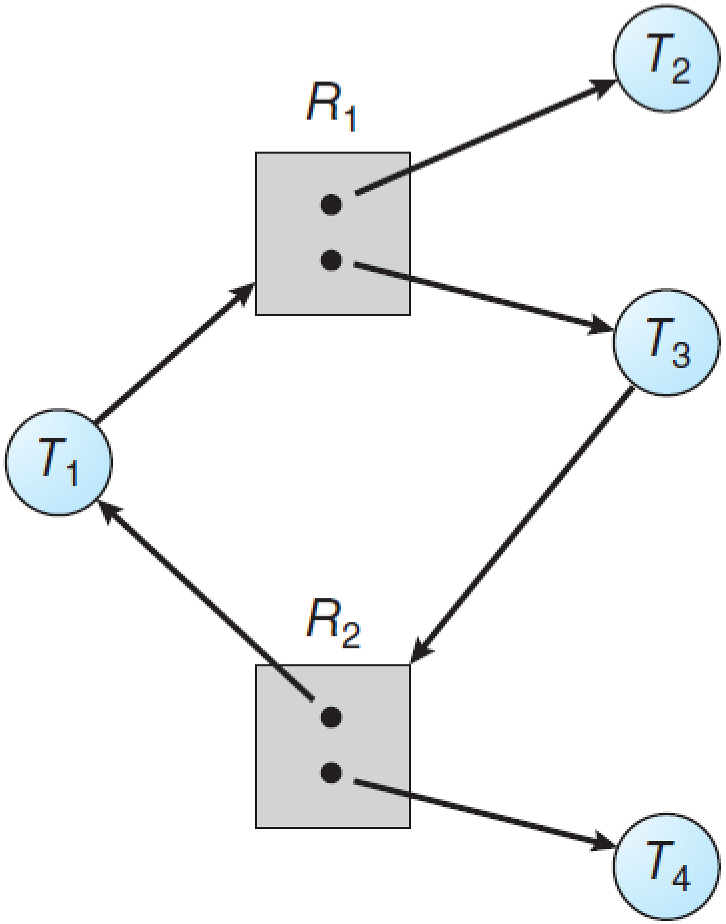


Figure 8.5 *Resource-allocation graph with a deadlock.*

- Two cycles exist in this graph.

$$T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$$
$$T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$$

8.3 Deadlock Characterization



- One cycle exists in this graph.

$$T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$$

Figure 8.6 *Resource-allocation graph with a cycle but no deadlock.*

8.3 Deadlock Characterization

- An important observation:
  - If a *resource-allocation graph* does *not have a cycle*,
    - then the system is ***not*** in a deadlocked state.
  - If a *resource-allocation graph* *has a cycle*,
    - then the system ***may*** or ***may not*** be in a deadlocked state.

8.3 Methods for Handling Deadlocks

- Three ways of dealing with the Deadlock Problem:
  - *Ignore* the problem altogether
    - and pretend that deadlocks never occur in the system.
  - Use a protocol to ***prevent*** or ***avoid*** deadlocks,
    - ensuring that the system will *never enter* a deadlocked state.
    - Deadlock Prevention ($8.5)
    - *Deadlock Avoidance ($8.6):* ***Banker's Algorithm***
  - Allow the system to enter a deadlocked state,
    - then ***detect*** it, and ***recover*** it.
    - Deadlock Detection ($8.7)
    - Recovery from Deadlock ($8.8)

8.5 Deadlock Prevention

- **Deadlock Prevention:**
  - For a deadlock to occur,
    - each of the *four necessary conditions* must hold.
  - Hence, we can ***prevent*** the occurrence of a deadlock,
    - by ensuring that *at least one* of these conditions cannot hold.

1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

8.5 Deadlock Prevention

- ***Mutual Exclusion***
  - At least one resource must be non-sharable.
  - *In general*, it *cannot be applied to* most applications.
    - some resources are *intrinsically* non-sharable.
    - e.g., a mutex lock cannot be shared by several threads.

- ***Hold and Wait***
  - We can guarantee that, whenever a thread requests a resource,
    - it does not hold any other resources.
  - It is *impractical* for most applications.

8.6 Deadlock Avoidance

- ***No preemption***
  - We can use a protocol to ensure that there should be *preemption*.
  - If a thread is holding some resources and requests another resources
    - that cannot be immediately allocated to it.
    - then, all resources the thread is currently holding are *preempted*.
  - The preempted resources are added to the list of resources
    - for which the threads are waiting.
  - The thread will be restarted
    - only when it can regain its *old resources* as well as *new ones*.
  - *cannot generally be applied* to most applications.

8.6 Deadlock Avoidance

- ***Circular Wait***: sometimes *practical*.
  - Impose a *total ordering* of all resource types
    - and to *require* that each thread requests
    - resources in an *increasing order* of enumeration.
  - It is *provable* that these two protocols are used,
    - then the circular-wait condition *cannot hold*.
  - Note that, however,
    - imposing a lock ordering does not guarantee deadlock prevention,
    - if locks can be acquired dynamically.

8.6 Deadlock Avoidance

```
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);
}
```

```
transaction(checking_account, savings_account, 25.0)

transaction(savings_account, checking_account, 50.0)
```

Figure 8.7 *Deadlock example with lock ordering.*

8.6 Deadlock Avoidance

8.17 In Section 8.5.4, we described a situation in which we prevent deadlock by ensuring that all locks are acquired in a certain order. However, we also point out that deadlock is possible in this situation if two threads simultaneously invoke the `transaction()` function. Fix the `transaction()` function to prevent deadlocks.

8.6 Deadlock Avoidance

- The Demerits of the Deadlock Prevention:
  - It prevents deadlocks by limiting how requests can made,
    - ensuring that *at least one of the necessary conditions* cannot occur.
  - However, possible side effects of preventing deadlocks are
    - *low device utilization* and *reduced system throughput*.

- **Deadlock Avoidance**:
  - Let the system to decide for each request whether or not
    - the thread should *wait* in order to *avoid* a possible *future deadlock*.
  - It requires *additional information* about
    - *how resources are to be requested*.
  - For example, in a system with resources R_1 and R_2,
    - A thread P will request first R_1 and then R_2 before releasing them.
    - A thread Q will request R_2 then R_1.

8.6 Deadlock Avoidance

- Given a *priori* information,
  - it is possible to *construct an algorithm* that
    - ensures the system will *never enter* a deadlocked state.
  - Let the ***maximum number*** of resources of each type that it may need.
  - Let the ***state*** of resource allocation be
    - the number of ***available*** and ***allocated*** resources
    - and the ***maximum demands*** of the threads.

8.6 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 *if only if* there exists a **safe sequence**.
  - A sequence of threads $\langle T_1, T_2, \cdots, T_n \rangle$ is a *safe sequence*,
    - if, for each thread T_i, the resources that T_i can still request
    - can be satisfied by the currently available resources + resources held by all T_j, with $j < i$.

8.6 Deadlock Avoidance

- Basic facts:
  - A *safe state* is *not* a *deadlocked state*.
  - Conversely, a deadlocked state is an unsafe state.
  - However, not all unsafe states are deadlocks,
    - an unsafe state ***may*** lead to a deadlock.

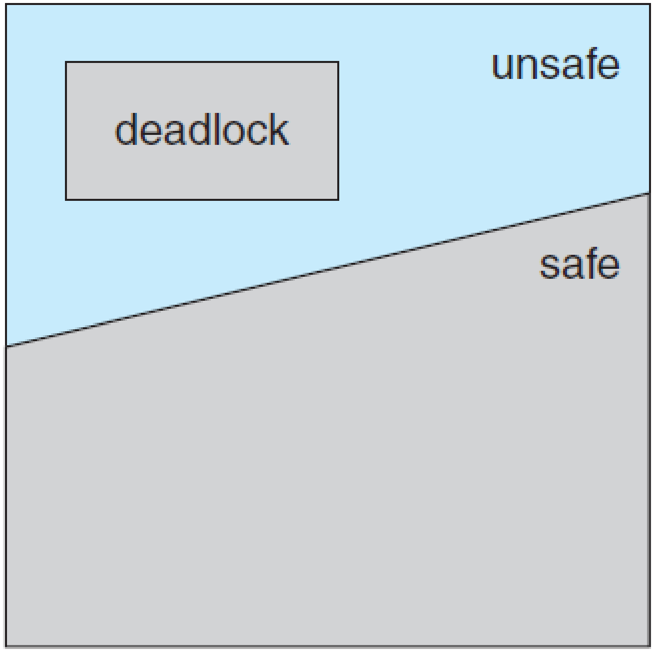


Figure 8.8 *Safe, unsafe, and deadlocked state spaces.*

8.6 Deadlock Avoidance

- Given the concept of a **safe state**:
  - we can define an avoidance algorithm that
    - ensure that the system will *never enter* a deadlocked state.
  - The idea is simply to ensure that
    - the system will ***always remain in a safe state***.
  - *Initially*, the system is *in a safe state*.
  - Whenever a thread *requests* a *resource* that is currently *available*,
    - the system *decides* whether the resource can be *allocated or not*.
  - The request is *granted*
    - *if and only if* the allocation *leaves* the system *in a safe state*.

8.6 Deadlock Avoidance

- Revisit the Resource-Allocation Graph:
  - Suppose that a system has *only one instance* of each resource type.
  - Then, introduce a new type of edge, called a ***claim edge***.
  - A *claim edge*: $T_i \rightarrow R_j$ indicates that
    - a thread *may request* a resource *at some time in the future*.
  - Then we can check for the safety
    - by a *cycle-detection* algorithm in a directed graph.
  - If ***no cycle*** exists, the request can be *granted* immediately,
    - since the resource allocation will *leave* the system in a *safe state*.
  - If a *cycle* is *detected*, then the request *cannot be granted*,
    - since the resource allocation will *put* the system in an *unsafe state*.

8.6 Deadlock Avoidance

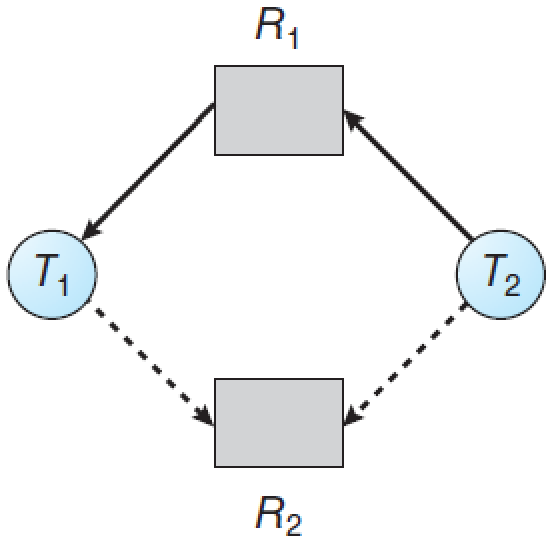


Figure 8.9 *Resource-allocation graph for deadlock avoidance.*

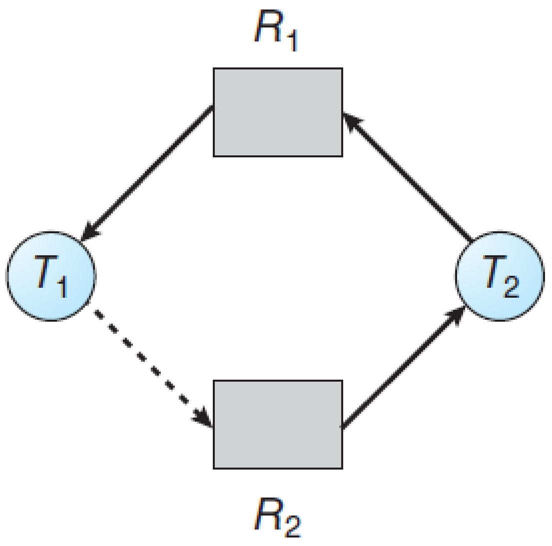


Figure 8.10 *An unsafe state in a resource-allocation graph.*

8.6 Deadlock Avoidance

- Banker's Algorithm:
  - RAG is *not applicable* to a resource allocation system
    - with *multiple instances* of each resource type.
  - Banker's algorithm is *applicable* to such a system
    - but *less efficient* and *more complicated* than the RAG.
  - Why Banker's?
    - the bank never allocates its available cash in such a way that
    - it could no longer satisfy the needs of all its customers.

8.6 Deadlock Avoidance

- Data structures:
  - Let n be the number of *threads* in the system
    - and let m be the number of *resource types*.
  - **Available**: A *vector* indicates the number of *available resource types*.
  - **Max**: A *matrix* defines the *maximum demand* of each thread.
  - **Allocation**: A *matrix* defines the number of resources of each type *currently allocated* to each thread.
  - **Need**: A *matrix* indicates the *remaining resource need* of each thread.

8.6 Deadlock Avoidance

- Data Structures:
  - $\boldsymbol{Available[m]}$:
    - if $Available[j] == k$, then k instances of R_j are available.
  - $\boldsymbol{Max[n \times m]}$:
    - if $Max[i][j] == k$, then T_i may request at most k instances of R_j .
  - $\boldsymbol{Allocation[n \times m]}$:
    - if $Allocation[i][j] == k$, then T_i is currently allocated k instances of R_j.
  - $\boldsymbol{Need[n \times m]}$:
    - if $Need[i][j] == k$, then T_i may need k more instances of R_j.

8.6 Deadlock Avoidance

Safety Algorithm:

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

8.6 Deadlock Avoidance

- **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:

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

8.6 Deadlock Avoidance

- An illustrative example:
  - a set of five threads: $T = \{T_0, T_1, T_2, T_3, T_4\}$
  - a set of three resource types: $R = \{A, B, C\}$
  - the number of instances of each resource types: $A = 10, B = 5, C = 7$
  - the snapshot representing the current state of the system:

	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	

8.6 Deadlock Avoidance

- Note that $Need[i][j] = Max[i][j] - Allocation[i][j]$.

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

8.6 Deadlock Avoidance

- Now we claim that the system is currently in a safe state.
  - In deed, the sequence $\langle T_1, T_3, T_4, T_2, T_0 \rangle$ satisfies the safety criteria.

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

8.6 Deadlock Avoidance

- When a new request is submitted:
  - Suppose that T_1 requests one instance of A and two instances of C.
    - $Request_1 = (1, 0, 2)$,
  - Decide whether this request should be *granted or not*.

$$Request_1 \leq Available$$
$$(1,0,2) \leq (3,3,2)$$

$$(3,3,2) - (1,0,2) = (2,3,0)$$
$$(1,2,2) - (1,0,2) = (0,2,0)$$

	Allocation	Need	Available
	A B C	A B C	A B C
T_0	0 1 0	7 4 3	$\boxed{2\ 3\ 0}$
T_1	3 0 2	$\boxed{0\ 2\ 0}$	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

8.6 Deadlock Avoidance

- Now, determine whether this new system state is safe.
  - Safety algorithm finds that $\langle T_1, T_3, T_4, T_0, T_2 \rangle$ satisfies the safety.

	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	

8.6 Deadlock Avoidance

- Now, determine with a request of $(3, 3, 0)$ by T_4.
  - $Request_4 = (3, 3, 0)$,

	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	

8.6 Deadlock Avoidance

- How about a request of $(0, 2, 0)$ by T_0?
  - $Request_0 = (0, 2, 0)$,

	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	

8.7 Deadlock Detection

- Deadlock Detection:
  - If a system does not *prevent* or *avoid* the deadlock,
    - then a deadlock situation may occur.
  - In this environment, the system may provide:
    - An algorithm that examines the state of the system to *determine whether* a deadlock has *occurred*.
    - An algorithm to *recover* from the deadlock.

8.6 Deadlock Avoidance

- Single Instance of Each Resource Type
  - Maintain a ***wait-for graph***,
    - a variant of the resource-allocation graph.
  - *Periodically*, *invoke an algorithm* that
    - searches for a cycle in the *wait-for* graph.

8.6 Deadlock Avoidance

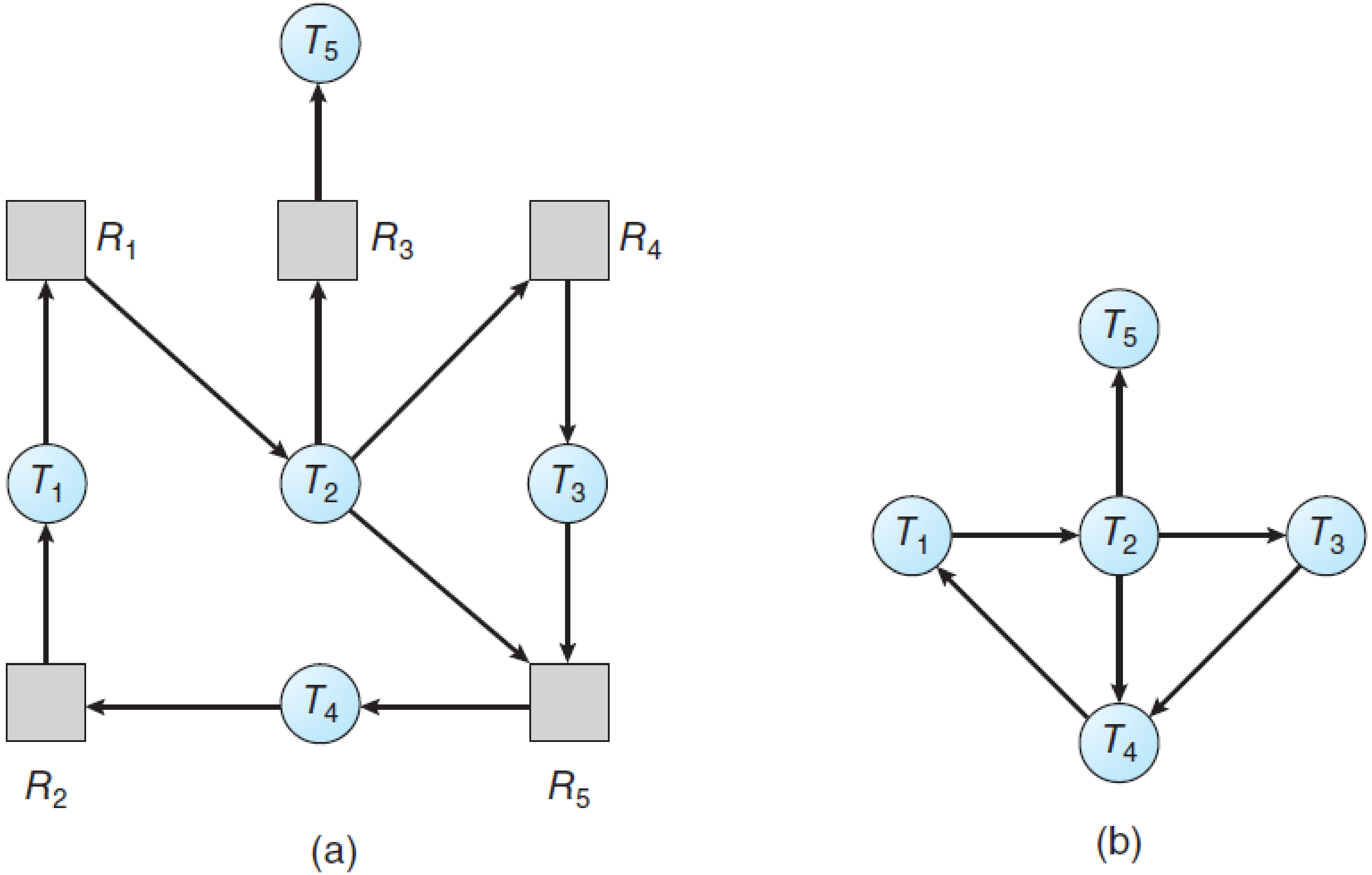


Figure 8.11 *(a) Resource-allocation graph. (b) Corresponding wait-for graph.*

8.7 Deadlock Detection

- Several Instances of a Resource Type:
  - The wait-for graph is not applicable to a system
    - with *multiple instances* of each resource type.
  - We can design a deadlock detection algorithm
    - that is *similar to* those used in *the banker's algorithm*.

8.7 Deadlock Detection

- Data Structures:
  - $\boldsymbol{Available}[\boldsymbol{m}]$:
  - $\boldsymbol{Allocation}[\boldsymbol{n} \times \boldsymbol{m}]$:
  - $\boldsymbol{Request}[\boldsymbol{n} \times \boldsymbol{m}]$: indicates the *current request* of each thread.
    - if $Request[i][j] == k$, then T_i is requesting k more instances of R_j.

8.7 Deadlock Detection

▪ Detection 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.

8.7 Deadlock Detection

- An illustrative example:
  - a set of five threads: $T = \{T_0, T_1, T_2, T_3, T_4\}$
  - a set of three resource types: $R = \{A, B, C\}$
  - the number of instances of each resource types: $A = 7, B = 2, C = 6$.
  - the snapshot representing the current state of the system:

	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	

8.7 Deadlock Detection

- Now we claim that the system is *not in a deadlocked state*.
  - the sequence $\langle T_0, T_2, T_3, T_1, T_4 \rangle$ results in $Finish[i] == true$ for all i.

	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	

8.7 Deadlock Detection

- Now we claim that the system is *now deadlocked*.
  - a deadlock exists, consisting of threads $T_1, T_2, T_3,$ and T_4.

	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	$\boxed{0\ 0\ 1}$	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

8.8 Recovery from Deadlock

- *When* should we invoke the detection algorithm?
  - How *often* is a deadlock likely to occur?
    - *more frequent* deadlocks, *more frequent* deadlock detections.
  - How *many* threads will be affected by deadlock when it happens?
    - the number of threads involved in the deadlock cycle *may grow*.

  - Invoking *for every request* .vs. invoking *at defined intervals*.
    - Note that there is a considerable overhead in computation time.
    - However, there may be many cycles in the resource graph,
    - if the detection algorithm is invoked at arbitrary points in time.

8.4 Recovery from Deadlock

- When a detection algorithm determines a deadlock exists,
  - *inform the operator* that a deadlock has occurred.
  - or let the system *recover from* the deadlock *automatically*.
    - Process and Thread Termination
    - Resource Preemption

8.4 Recovery from Deadlock

- Deadlock Recovery
  - Process and Thread Termination:
    - Abort *all* deadlocked processes.
    - Abort *one* process *at a time* until the deadlock cycle is eliminated.
  - Resource Preemption:
    - Selecting a victim: consider the order of preemption to *minimize cost*.
    - Rollback: *roll back* the process to some *safe state* and *restart* it.
    - Starvation: *picked as a victim* only a finite number of times.

Any Questions?