Deadlock

Outline

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

The concepts explained for threads in this lecture may also be applicable to processes

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

- Computer System consists of resources:
  - Physical – CPU cycles, RAM, HDD, NIC,I/O devices ...
  - Logical – Files, Processes, Threads, Virtual Memory….

- Resource types $R_1, R_2, . . ., R_m$
  - CPU cycles, RAM, Files, Processes, Virtual Memory….

- Each resource type $\boldsymbol{R}_i$ has $\boldsymbol{W}_i$ instances.

System Model

- Each process utilizes a resource as follows:
  - ❑ **Request**
    - – request the resource
    - – if the request is not granted (example:- mutex held by another process), wait until acquiring the resource.
  - ❑ **Use**
    - – operate on the resource (ex- acquire mutex lock, enter critical section).
  - ❑ **Release**
    - – operate on resource (ex- release mutex lock, exit critical section).

Deadlock with Semaphores

- **Data:**
  - A semaphore S_1 initialized to **1**
  - A semaphore S_2 initialized to **1**
- Two **threads** T_1 and T_2 are created and both have access to S_1 and S_2

T_1 runs, followed by T_2. Is there a possible Deadlock?

T_1:

```
wait(s1) {while(s1<=0); s1--;}
wait(s2) {while(s2<=0); s2--;}
```

T_2:

```
wait(s1) {while(s1<=0); s1--;}
wait(s2) {while(s2<=0); s2--;}
```

Deadlock with Semaphores

- **Data:**
  - A semaphore S_1 initialized to **1**
  - A semaphore S_2 initialized to **1**
- Two **threads** T_1 and T_2 are created and both have access to S_1 and S_2

T_1 runs, followed by T_2. Is there a possible Deadlock?

```
T1:
      wait(s1) {while(s1<=0); s1--;}
      wait(s2) {while(s2<=0); s2--;}
T2:
      wait(s2) {while(s2<=0); s2--;}
      wait(s1) {while(s1<=0); s1--;}
```

Deadlock with Semaphores

- **Data:**
  - A semaphore S_1 initialized to **1**
  - A semaphore S_2 initialized to **1**
- Two **threads** T_1 and T_2 are created and both have access to S_1 and S_2

T_1 and T_2 run concurrently. Is there a possible Deadlock?

```
T1:
    wait(s1) {while(s1<=0); s1--;}
    wait(s2) {while(s2<=0); s2--;}
T2:
    wait(s1) {while(s1<=0); s1--;}
    wait(s2) {while(s2<=0); s2--;}
```

Deadlock with Semaphores

- **Data:**
  - A semaphore S_1 initialized to **1**
  - A semaphore S_2 initialized to **1**
- Two **threads** T_1 and T_2 are created and both have access to S_1 and S_2

T_1 and T_2 run concurrently. Is there a possible Deadlock?

```
T1:
     wait(s1) {while(s1<=0); s1--;)
     wait(s2) {while(s2<=0); s2--;)
T2:
     wait(s2) {while(s2<=0); s2--;)
     wait(s1) {while(s1<=0); s1--;)
```

Deadlock with Mutex

- **Two mutex locks are Created and Initialized:**

```
pthread_mutex_t M1;
pthread_mutex_t M2;
pthread_mutex_init(&M1 , NULL);
pthread_mutex_init(&M2, NULL);
```

- Two **threads** T_1 and T_2 are created, and both of these have access to both the mutex locks.

Deadlock with Mutex

T_1 **runs, followed by** T_2 **. Is there a possible Deadlock?**

T_1 :

```
void *do_work_one(void *param){
        pthread mutex lock(& M1);
        pthread mutex lock(& M2);
        /* Do some work */
        pthread mutex unlock(& M2);
        pthread mutex unlock(& M1);
        pthread exit(0);
 }
```

T_2 :

```
void *do_work_two(void *param){
        pthread mutex lock(& M1);
        pthread mutex lock(& M2);
        /* Do some work */
        pthread mutex unlock(& M2);
        pthread mutex unlock(& M1);
        pthread exit(0);
}
```

Deadlock with Mutex

T_1 runs, followed by T_2 . Is there a possible Deadlock?

T_1:

```
void *do_work_one(void *param){
        pthread mutex lock(& M1);
        pthread mutex lock(& M2);
        /* Do some work */
        pthread mutex unlock(& M2);
        pthread mutex unlock(& M1);
        pthread exit(0);
 }
```

T_2:

```
void *do_work_two(void *param){
        pthread mutex lock(& M2);
        pthread mutex lock(& M1);
        /* Do some work */
        pthread mutex unlock(& M1);
        pthread mutex unlock(& M2);
        pthread exit(0);
}
```

Deadlock with Mutex

T_1 and T_2 run concurrently. Is there a possible Deadlock?

T_1:

```
void *do_work_one(void *param){
        pthread mutex lock(& M1);
        pthread mutex lock(& M2);
        /* Do some work */
        pthread mutex unlock(& M2);
        pthread mutex unlock(& M1);
        pthread exit(0);
 }
```

T_2:

```
void *do_work_two(void *param){
        pthread mutex lock(& M1);
        pthread mutex lock(& M2);
        /* Do some work */
        pthread mutex unlock(& M2);
        pthread mutex unlock(& M1);
        pthread exit(0);
}
```

Deadlock with Mutex

T_1 and T_2 run concurrently. Is there a possible Deadlock?

T_1:

```
void *do_work_one(void *param){
        pthread mutex lock(& M1);
        pthread mutex lock(& M2);
        /* Do some work */
        pthread mutex unlock(& M2);
        pthread mutex unlock(& M1);
        pthread exit(0);
 }
```

T_2:

```
void *do_work_two(void *param){
        pthread mutex lock(& M2);
        pthread mutex lock(& M1);
        /* Do some work */
        pthread mutex unlock(& M1);
        pthread mutex unlock(& M2);
        pthread exit(0);
}
```

Necessary Conditions for Deadlock

- Deadlock can arise if four conditions **hold simultaneously**.
  - **Mutual exclusion:** resource is nonsharable and only one thread at a time can use a resource.
  - **Hold and wait:** a thread holding at least one resource is waiting to acquire additional resources held by other threads.
  - **No preemption:** a resource can be released only voluntarily by the thread holding it, after that thread has completed its task.
  - **Circular wait:** there exists a set $\{T_0, T_1, \ldots, T_n\}$ of waiting threads such that T_0 is waiting for a resource that is held by T_1, T_1 is waiting for a resource that is held by T_2, …, T_{n-1} is waiting for a resource that is held by T_n, and T_n is waiting for a resource that is held by T_0.

Resource-Allocation Graph (RAG)

A set of vertices **V** and a set of edges **E**.

- **V** is partitioned into two types:
  - $\boldsymbol{T} = \{T_1, T_2, \ldots, T_n\}$, the set consisting of all the active threads in the system.
  - $\boldsymbol{R} = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $\boldsymbol{T_i} \rightarrow \boldsymbol{R_j}$
- **assignment edge** – directed edge $\boldsymbol{R_j} \rightarrow \boldsymbol{T_i}$

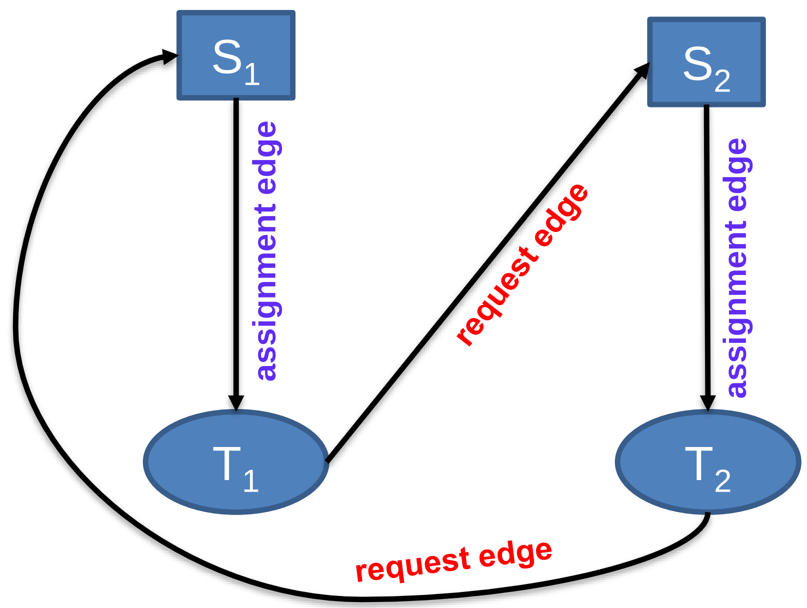
Resource-Allocation Graph (RAG)

- **Data:**
  - A semaphore S_1 initialized to 1
  - A semaphore S_2 initialized to 1
- Two **threads** T_1 and T_2 are created and both have access to S_1 and S_2

T_1 and T_2 run concurrently. Is there a possible Deadlock?

```
T1:
    wait(s1) {while(s1<=0); s1--;}
    wait(s2) {while(s2<=0); s2--;}
T2:
    wait(s2) {while(s2<=0); s2--;}
    wait(s1) {while(s1<=0); s1--;}
```

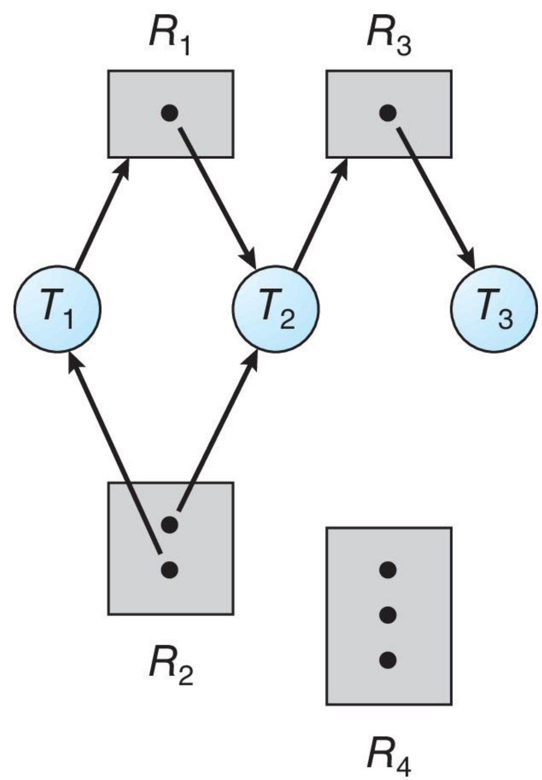
Deadlock !



RAG Example

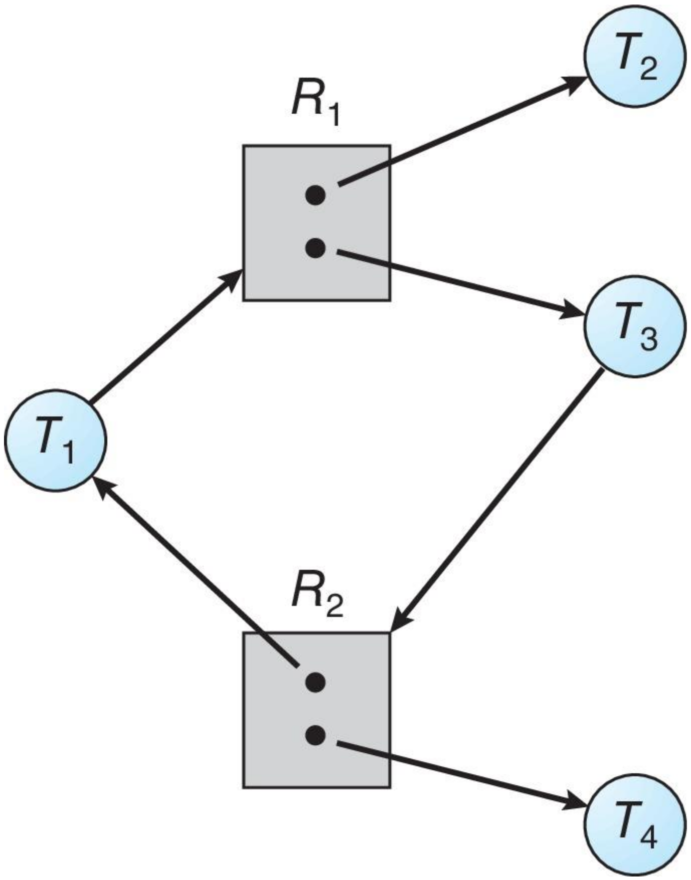
- One instance of $\mathbf{R_1}$
- Two instances of $\mathbf{R_2}$
- One instance of $\mathbf{R_3}$
- Three instances of $\mathbf{R_4}$

- $\mathbf{T_1}$ holds one instance of $\mathbf{R_2}$ and is waiting for an instance of $\mathbf{R_1}$
- $\mathbf{T_2}$ holds one instance of $\mathbf{R_1}$, one instance of $\mathbf{R_2}$, and is waiting for an instance of $\mathbf{R_3}$
- $\mathbf{T_3}$ holds one instance of $\mathbf{R_3}$



No Cycle, No Deadlock !

RAG with a Cycle but No Deadlock

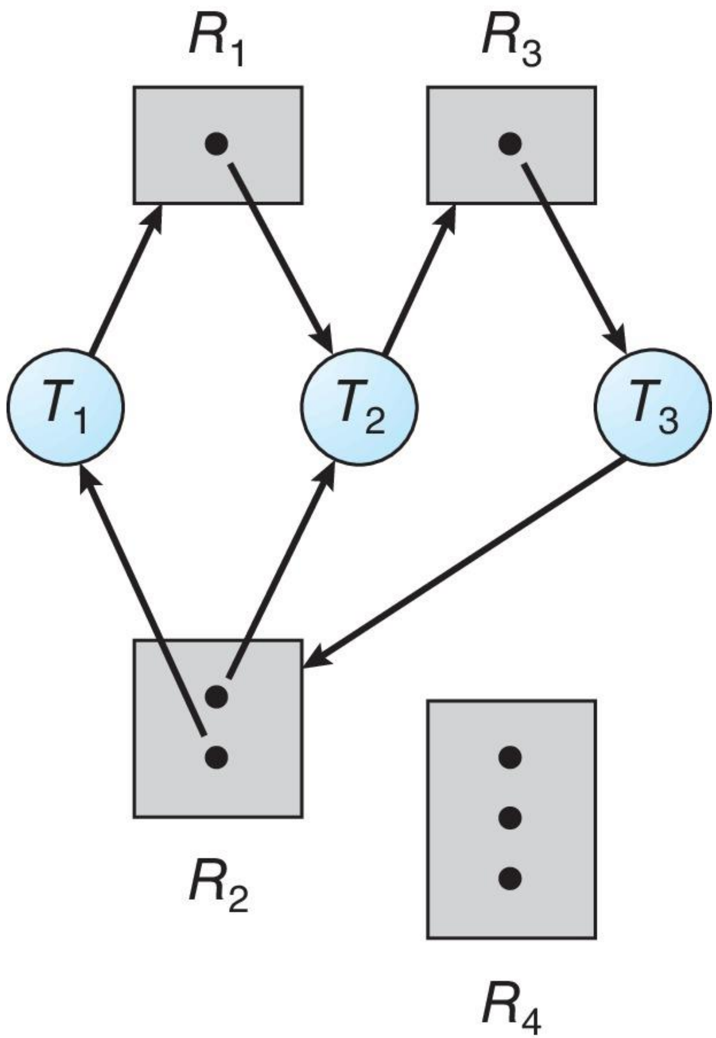


One cycle exists:

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

If Cycle, Deadlock may or may not exist

RAG with a Cycle and a Deadlock



Two cycles exist:

1) $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
2) $T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$

If Cycle, Deadlock may or may not exist

RAC Basic Facts

- If the graph contains no cycles $\Rightarrow$ **no deadlock**
- If the graph contains a cycle $\Rightarrow$
  - if *only one instance* per resource type, **there is a deadlock.**
  - if *several instances* per resource type, **there is a possibility of deadlock.**

Methods for Handling Deadlocks

1. **Ensure** that the system will **never** enter a deadlock state:
   - **Deadlock prevention** – prevent deadlocks by limiting how resource requests can be made.
   - **Deadlock avoidance** – avoid deadlocks by acquiring additional information about how resources are to be requested.

2. **Allow** the system to enter a deadlock state, **detect** it, and **recover** from it.

3. **Ignore** the problem and **pretend** that deadlocks **never occur** in the system.

Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources.

- **Hold and Wait** – must guarantee that whenever a thread requests a resource, it does not hold any other resources.
  - Require threads to request and be allocated all their resources before they begin execution **OR** allow a thread to request resources only when the thread has none allocated to it.
  - Starvation possible.

Deadlock **Prevention** (Cont.)

- **No Preemption:**
  - ❑ If a thread is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held by this thread are released.
  - ❑ A thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

OR

  - ❑ If a thread is requesting resources and the resources are not available, check if any other thread is holding these unavailable resources in a waiting state.
  - ❑ If yes, preempt the resources from the waiting thread and allocate them to the requesting thread.
  - ❑ If no (neither available; nor held), the requesting thread must wait.

Deadlock Prevention (Cont.)

- **Circular Wait (a practical solution):**
  - **Impose a total ordering** of all resource types, and require that each thread requests resources in an increasing order of enumeration.
  - Define a **one-to-one function** $F: R \rightarrow N$, *where*
    - *N is a set of natural numbers*
    - *$R = \{R_1, R_2, \ldots, R_m\}$ is the set of resource types*

Circular Wait

Protocol 1:

- Each thread can request resources only in an increasing order of enumeration.
  - Request R_i followed by request to R_j, if and only if $F(R_j) > F(R_i)$.
  - Example - a thread that wants to use two mutexes $\mathbf{M_1}$ and $\mathbf{M_2}$ at the same time must first request $\mathbf{M_1}$ and then $\mathbf{M_2}$

Protocol 2:

- A thread requesting an instance of resource R_i must have released any resources R_j such that $F(R_j) > F(R_i)$.

Note - if several instances of the same resource type are needed, a ***single*** request for all of them must be issued.

Circular Wait

- If:

  first_mutex = 1
  second_mutex = 5

  code for **thread_two** could not be written as follows:

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}
```

```
/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

Deadlock **Prevention**

Deadlock prevention algorithms may lead to **low device utilization** and **reduced system throughput**.

Deadlock Avoidance

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

- The simplest and most useful deadlock avoidance 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

- If a system can allocate resources to each thread *up to its maximum* in some order and still avoid a deadlock, we say the system is in a **safe state.**

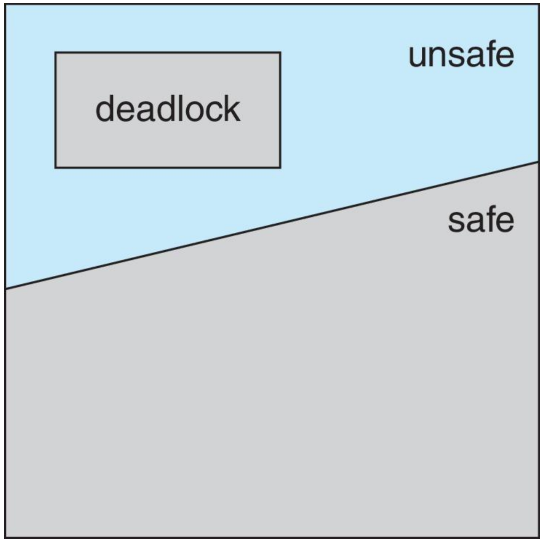
- The system is in a **safe state** if there exists a safe sequence **$<T_1, T_2, \ldots, T_n>$** of ALL the threads in the system such that for each T_i, the resources that T_i can still request can be satisfied by currently available resources **+** resources held by all the T_j, with **$j < i$**.

  That is:

  - If T_i resource needs are not immediately available, then T_i can wait until all T_j have finished.
  - When T_j is finished, T_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When T_i terminates, T_{i+1} can obtain its needed resources, and so on...

Safe State - Basic Facts

- If a system is in a safe state ⇒ no deadlocks.
- If a system is in an unsafe state ⇒ possibility of deadlock.
- Deadlock avoidance ⇒ ensures that a system will never enter an unsafe state.



Safe, Unsafe, Deadlock State

Deadlock Avoidance Algorithms

- If a single instance of each resource type
  - Use a **resource-allocation graph varient**
- If multiple instances of a resource type
  - Use the **Banker's Algorithm**

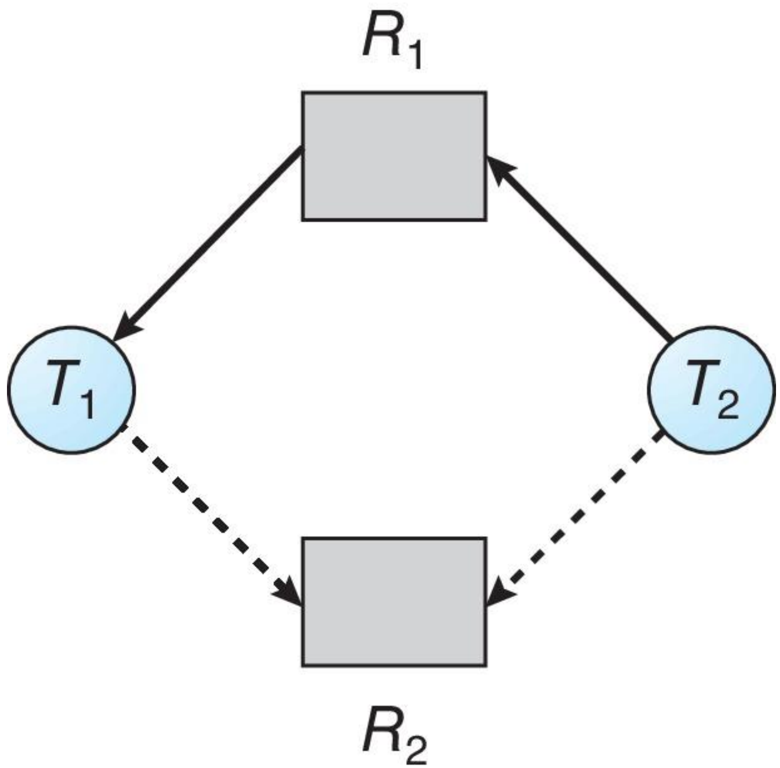
Deadlock Avoidance – RAG Variant

- **Claim edge** $T_i \rightarrow R_j$ indicated that thread T_j may request resource R_j. Represented by a **dashed line.**

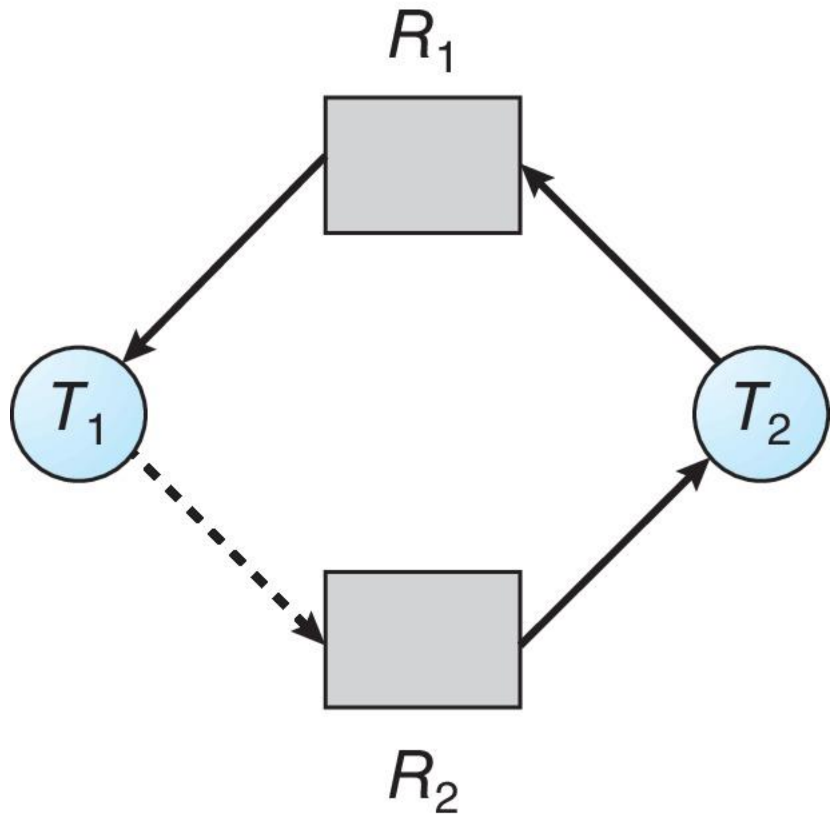
1. Claim edge converts to a request edge when a thread requests a resource.
2. The request edge is converted to an assignment edge when the resource is allocated to the thread.
3. When a resource is released by a thread, the assignment edge reconverts to a claim edge.

- Resources must be claimed ***a priori*** in the system:
  - before thread *Ti* starts executing, all its claim edges must already appear in the resource-allocation graph.

Deadlock Avoidance – RAG Variant



Deadlock Avoidance – Unsafe State in RAG Variant



Resource-Allocation Graph Algorithm

- Suppose that thread T_i requests a resource R_j
  - The request can be **granted** only if converting the request edge to an assignment edge **does not** result in the formation of a **cycle** in the resource allocation graph.

Banker's Algorithm

1. Safety Algorithm
2. Resource-Request Algorithm

Data Structures for the Banker's Algorithm

Let n = number of threads, and m = number of resource 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]

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 thread T_i

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

1. If $\boldsymbol{Request_i \le Need_i}$ go to step 2. Otherwise, raise error condition, since thread has exceeded its maximum claim.

2. If $\boldsymbol{Request_i \le Available}$, 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:

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

- **If safe** $\Rightarrow$ the resources are allocated to $\boldsymbol{T_i}$
- **If unsafe** $\Rightarrow$ $\boldsymbol{T_i}$ must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- Considering a system with five processes T_0 through T_4 and three resources of type A, B, C.

- Resource type A has 10 instances, B has 5 instances and type C has 7 instances.

- Suppose at time t_0 following snapshot of the system has been taken

Example of Banker's Algorithm

- **5** threads T_0 through T_4;

  3 resource types:

  A (10 instances), B (5 instances), and C (7 instances)

- **Snapshot at time T_0:**

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

Example of Banker's Algorithm

- **Snapshot at time T_0:**
- The content of the **matrix** ***Need*** is defined to be ***Max – Allocation***

	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	7 4 3
T_1	2 0 0	3 2 2		1 2 2
T_2	3 0 2	9 0 2		6 0 0
T_3	2 1 1	2 2 2		0 1 1
T_4	0 0 2	4 3 3		4 3 1

Safety Algorithm - 1

- m=3, n=5
- Work = Available = (3,3,2)
- Finish[5] = {false, false, false, false, false}

- For i=0, **$Need_0$** = (7,4,3)
- Finish[0] = **false && $Need_0$ (7,4,3) > Work (3,3,2)**
- So, **T_0 has to wait**

	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

Safety Algorithm - 2

- For i=1, **Need$_1$** = (1,2,2)
- Finish[1] = false && Need$_1$ (1,2,2) < Work (3,3,2)
- So **T_1** **is kept in safe sequence**

	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

Safety Algorithm - 2

- Work = Work + Allocation_1
- Work = (3,3,2) + (2,0,0)=(5,3,2)
- Finish[5] = {false, true, false, false, false}

	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	*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	

Safety Algorithm - 3

- For i=2, **Need$_2$** = (6,0,0)
- Finish[2] = false && Need$_2$ (6,0,0) > Work (5,3,2)
- So **T$_2$** **must wait**

	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

Safety Algorithm - 4

- For i=3, **Need$_3$** = (0,1,1)
- Finish[3] = false && Need$_3$ (0,1,1) < Work (5,3,2)
- So **T$_3$** **is kept in safe sequence**

	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

Safety Algorithm - 4

- Work = Work + Allocation$_3$
- Work = (5,3,2) + (2,1,1)=(7,4,3)
- Finish[5] = {false, true, false, true, false}

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	*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	

Safety Algorithm - 5

- For i=4, **$Need_3$** = (4,3,1)
- Finish[4] = false && $Need_4$ (4,3,1) < Work (7,4,3)
- So **T_4** **is kept in safe sequence**

	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

Safety Algorithm - 5

- Work = Work + Allocation$_4$
- Work = (7,4,3) + (0,0,2)=(7,4,5)
- Finish[5] = {false, true, false, true, true}

	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	*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	

Safety Algorithm - 6

- For i=0, **Need$_0$** = (7,4,3)
- Finish[0] = false && Need$_0$ (7,4,3) < Work (7,4,5)
- So T_0 **is kept in safe sequence**

	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	*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	

Safety Algorithm - 6

- Work = Work + $\text{Allocation}_{\mathbf{0}}$
- Work = (7,4,5) + (0,1,0)=(7,5,5)
- Finish[5] = {true, true, false, true, true}

	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	*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	

Safety Algorithm - 7

- For i=2, **$Need_2$** = (6,0,0)
- Finish[2] = false && $Need_2$ (6,0,0) < Work (7,5,5)
- So **T_2** **is kept in safe sequence**

	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

Safety Algorithm - 7

- Work = Work + $Allocation_2$
- Work = (7,5,5) + (3,0,2)=(10,5,7)
- Finish[5] = {true, true, true, true, true}

	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	*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	

System in Safe Sequence

The system is in a safe state since the sequence
< T_1, T_3, T_4, T_2, T_0> satisfies safety criteria

T_1 Request (1,0,2)

Run the Resource-Request Algorithm

T_1 checks
Step1: $Request_1$ < $Need_1$ i.e.,(1,0,2) < (7,4,3) and
Step2: $Request_1$ < Avaiable i.e.,(1,0,2) < (3,3,2)

According to Step3 in Resource-Request Algorithm for Process T_i
Available = Available – $Request_i$;
$Allocation_i$ = $Allocation_i$ + $Request_i$;
$Need_i$ = $Need_i$ – $Request_i$;

We have
(3,3,2) – (1,0,2) = (2,3,0) Available
(0,1,0) + (1,0,2) = (1,1,2) $Allocation_1$
(1,2,2) – (1,0,2) = (0,2,0) $Need_1$

T_1 Request (1,0,2)

- Determine whether this new state is safe.
  - ❑ **Execute Safety Algorithm**

Snapshot at old time T_0:

	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	7 4 3
T_1	2 0 0	3 2 2		1 2 2
T_2	3 0 2	9 0 2		6 0 0
T_3	2 1 1	2 2 2		0 1 1
T_4	0 0 2	4 3 3		4 3 1

Snapshot at new time T_1:

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

Safety Algorithm - 1

- m=3, n=5
- Work = Available = (2,3,0)
- Finish[5] = {false, false, false, false, false}

- For i=0, **$Need_0$** = (7,4,3)
- Finish[0] = **false && $Need_0$ (7,4,3) > Work (2,3,0)**
- So **T_0 has to wait**

	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	

Safety Algorithm - 2

- For i=1, **Need$_1$** = (0,2,0)
- Finish[1] = false && Need$_1$ (0,2,0) < Work (2,3,0)
- So **T$_1$** **is kept in safe sequence**

	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	

Safety Algorithm - 2

- Work = Work + $\text{Allocation}_{\mathbf{1}}$
- Work = (2,3,0) + (3,0,2)=(5,3,2)
- Finish[5] = {false, true, false, false, false}

	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	

Safety Algorithm - 3

- For i=2, **Need$_2$** = (6,0,0)
- Finish[2] = false && Need$_2$ (6,0,0) > Work (5,3,2)
- So **T$_2$** **must wait**

	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	

Safety Algorithm - 4

- For i=3, **Need$_3$** = (0,1,1)
- Finish[3] = false && Need$_3$ (0,1,1) < Work (5,3,2)
- So **T$_3$** **is kept in safe sequence**

	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	

Safety Algorithm - 4

- Work = Work + $Allocation_3$
- Work = (5,3,2) + (2,1,1)=(7,4,3)
- Finish[5] = {false, true, false, true, false}

	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	

Safety Algorithm - 5

- For i=4, **Need$_3$** = (4,3,1)
- Finish[4] = false && Need$_4$ (4,3,1) < Work (7,4,3)
- So **T$_4$** **is kept in safe sequence**

	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	

Safety Algorithm - 5

- Work = Work + $Allocation_4$
- Work = (7,4,3) + (0,0,2)=(7,4,5)
- Finish[5] = {false, true, false, true, 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	

Safety Algorithm - 6

- For i=0, **Need$_0$** = (7,4,3)
- Finish[0] = false && Need$_0$ (7,4,3) < Work (7,4,5)
- So **T$_0$** **is kept in safe sequence**

	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	

Safety Algorithm - 6

- Work = Work + $\text{Allocation}_{\mathbf{0}}$
- Work = (7,4,5) + (0,1,0)=(7,5,5)
- Finish[5] = {true, true, false, true, 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	

Safety Algorithm - 7

- For i=2, **Need$_2$** = (6,0,0)
- Finish[2] = false && Need$_2$ (6,0,0) < Work (7,5,5)
- So **T$_2$** **is kept in safe sequence**

	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	

Safety Algorithm - 7

- Work = Work + Allocation$_2$
- Work = (7,5,5) + (3,0,2)=(10,5,7)
- Finish[5] = {true, true, true, true, 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	

Safe Sequence

The system is in a safe state since the sequence

$< T_1, T_3, T_4, T_2, T_0 >$ satisfies safety criteria

Try these as your homework !!!

1. **Following the previous safe state,** T_4 *requests **(3,3,0)** instances of resource type **(A,B,C)** respectively.* Prove that this request cannot be granted.

2. **Following the previous safe state,** T_0 *requests **(0,2,0)** instances of resource type **(A,B,C)** respectively.* Prove that after completing this request, the system won't be in a safe state anymore.

Deadlock Detection

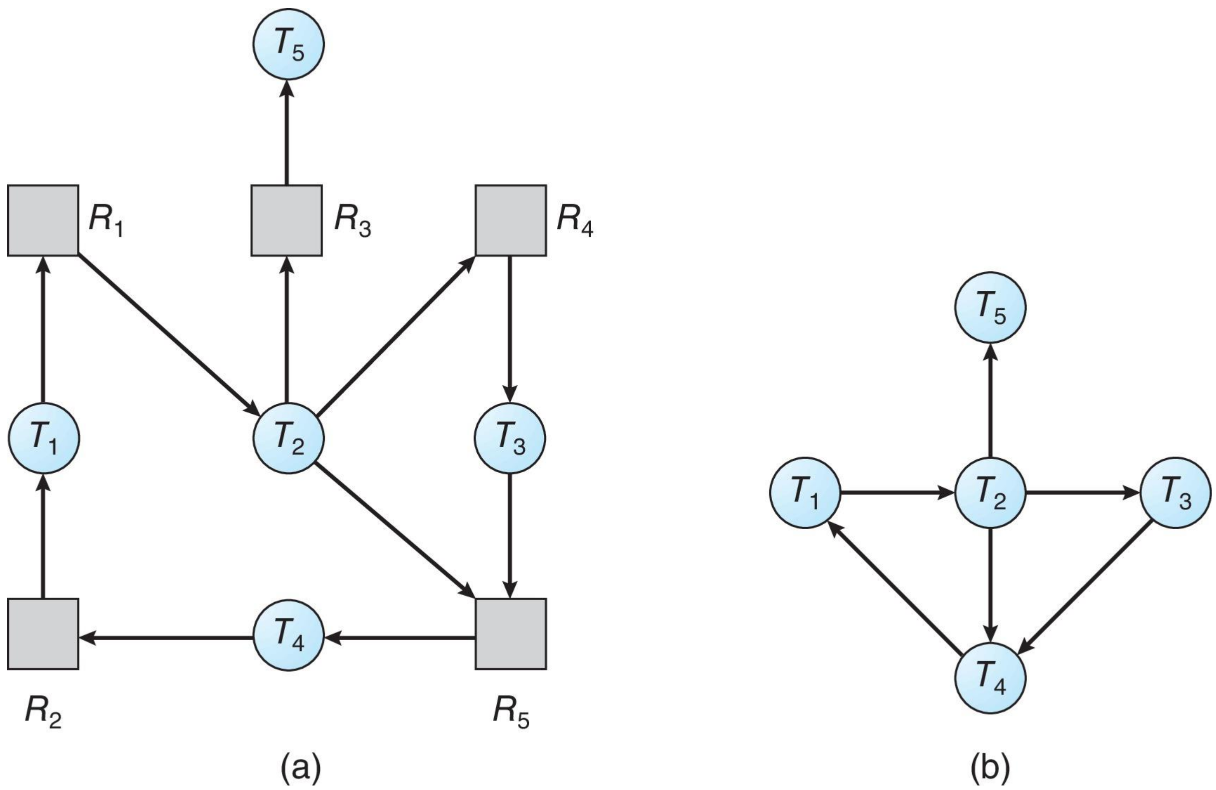
- Allow the system to enter a deadlock state, then use:
  1. Deadlock Detection Algorithm
  2. Deadlock Recovery Algorithm

- A lot of overhead is incurred in this scheme:
  1. Run-time cost of maintaining needed information
  2. Executing detection algorithm
  3. Potential losses due to deadlock recovery algorithm

Deadlock Detection - Single Instance of Each Resource Type

- Maintain **wait-for** graph (a variant of RAG.
  - **Nodes** are the **Threads**
  - $T_i \rightarrow T_j$ if T_i is waiting for T_j to release a resource that T_i needs
- 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 $\mathbf{O(n^2)}$ 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**

Deadlock Detection - Multiple Instances of Each Resource Type

Essential Data structures:

- **Available:** A vector of length m indicates the number of available resources of each type. If *Available* $[j] = k$, then k instances of resource type R_j are available.

- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each thread. If *Allocation* $[i][j] = k$, then thread T_i is currently allocated k instances of resource type R_j.

- **Request:** An $n \times m$ matrix indicates the current request of each thread. If *Request* $[i][j] = k$, then thread T_i is requesting k more instances of resource type R_j.

Deadlock Detection Algorithm - Multiple Instances of Each Resource Type

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively.

   Initialize:

   a) ***Work = Available***

   b) For ***i* = 0,2, …, *n-1***, 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.**

Deadlock Detection Algorithm - Multiple Instances of Each Resource Type

3. ***Work*** **=** ***Work*** **+** ***Allocation$_i$***
   Finish*[i] = *true
   go to step **2**

4. If ***Finish*[i] == *false***, for some ***i***, $1 \leq i \leq n$, then the system is in a deadlock state. Moreover, if ***Finish*[i] == *false***, then thread T_i is deadlocked

This algorithm requires an order of O(m x n^2) operations to detect whether the system is in a deadlocked state

Deadlock Detection Algorithm - Multiple Instances of Each Resource Type - Example

- Five threads T_0 through T_4;
- Three resource types
  A (7 instances), ***B*** (2 instances), and **C** (6 instances)

- **Snapshot of the current state of 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	

Deadlock Detection Algorithm - Multiple Instances of Each Resource Type - Example

Try executing threads is sequence:<T_0, T_2, T_3, T_1, T_4>

Thread(i)	$Allocation_i$ A B C	$Request_i$ A B C	Finish	Work A B C	Finish
				0 0 0	
T_0	0 1 0	0 0 0	false	0 1 0	**true**
T_2	3 0 3	0 0 0	false	3 1 3	**true**
T_3	2 1 1	1 0 0	false	5 2 4	**true**
T_1	2 0 0	2 0 2	false	7 2 4	**true**
T_4	0 0 2	0 0 2	false	7 2 6	**true**

- **Safe State Sequence** <T_0, T_2, T_3, T_1, T_4> will result in ***Finish[i] = true*** for all ***i***

Deadlock Detection Algorithm - Multiple Instances of Each Resource Type - Example

Suppose T_2 requests an additional instance of type **C**

- **Snapshot of 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 1**	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

State of the system? **Unsafe**

- ❑ Can reclaim resources held by thread T_0, but insufficient resources to fulfill other threads requests.
- ❑ **Deadlock exists**, consisting of threads T_1, T_2, T_3, and T_4 .

Thread(i)	Allocation$_i$ A B C	Request$_i$ A B C	Finish	Work A B C	Finish
				0 0 0	
T_0	0 1 0	0 0 0	false	0 1 0	**true**
T_2	3 0 3	**0 0 1**	false		**false**

Deadlock 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 when deadlock happens?
- If the detection algorithm is invoked frequently
  - Consumes a lot of CPU cycles
- If the detection algorithm is invoked rarely
  - The number of threads in the deadlock cycle may grow, causing a decrease in system throughput
- If the detection algorithm is invoked arbitrarily
  - There may be many cycles in the RAG so hard to detect which of the many deadlocked threads “caused” the deadlock.

Deadlock Detection-Algorithm Usage

- **Good strategies:**

- ❑ Scheduled periodically (e.g., Once per hour)
- ❑ Resource-utilization based (e.g., When CPU utilization drops below 50%)
- ❑ On-Demand (e.g., User or Admin request)
- ❑ Threshold-Based (e.g., Memory usage, Network traffic)
- ❑ Automated Triggers (e.g., After a certain number of transactions)
- ❑ Event-driven (e.g., Exception or Error detected)
- ❑ Priority-Based (e.g., Before running High-Priority transactions)
- ❑ Reactive (e.g., After Blocking/Waiting condition is detected)
- ❑ Combined Strategies

Recovery from Deadlock: Thread Termination

- Abort all deadlocked threads.

- Abort one thread at a time until the deadlock cycle is eliminated.

- In which order should we choose to abort? **Minimum Cost**
  1. Priority of the thread
  2. How long has the thread computed, and how much longer to completion
  3. Resources that the thread has used
  4. How many more resources thread need to complete it's execution
  5. Is the thread interactive or batch?
  6. How many threads will need to be terminated

Recovery from Deadlock: Resource Preemption

Successively preempt some resources from processes and give these resources to other processes **until the deadlock cycle is broken**.

- **Selecting a victim** – the aim is to minimize cost.
  - ❑ Number of resources held
  - ❑ Amount of time consumed
- **Rollback victim** – return to some safe state, and restart it from that state.
- **Starvation** – the same thread may always be picked as a victim.
  - ❑ include a count of number of rollbacks in the cost factor.