Deadlocks

Deadlock

- 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 can never again change state, because the resources it has requested are held by other waiting threads.
- This situation is called a **deadlock**.

Deadlock

- System consists of resources
- Resource types R_1, R_2, . . ., R_m
  - *CPU cycles, memory space, I/O devices*
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:

✔ **Request:** The process 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.

✔ **Use:** The process can operate on the resource (for example, if the resource is a mutex lock, the thread can access its critical section).

✔ **Release:** The thread releases the resource.

Deadlock

- 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.
- The events with which we are mainly concerned here are resource acquisition and release.
- In the dining-philosophers problem, resources are represented by chopsticks.
- If all the philosophers get hungry at the same time, and each philosopher grabs the chopstick on her left, there are no longer any available chopsticks.
- Each philosopher is then blocked waiting for her right chopstick to become available.

Deadlock with Semaphores

- Data:
  - A semaphore **`S1`** initialized to 1
  - A semaphore **`S2`** initialized to 1
- Two processes P1 and P2
- **`P1:`**

```
    wait(s1)
    wait(s2)
```

- **`P2:`**

```
    wait(s2)
    wait(s1)
```

Deadlock Characterization

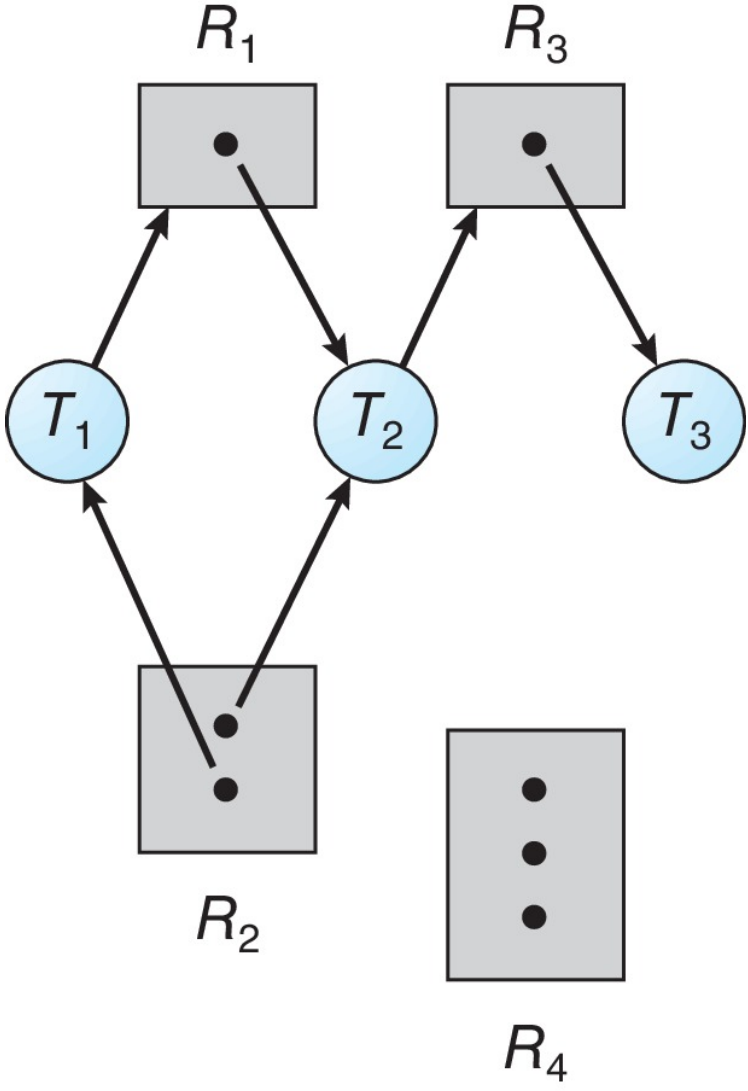
- Deadlock can arise if four conditions hold simultaneously.
- **Mutual exclusion**: only one process at a time can use a resource.
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption**: a resource can be released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

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 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 $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

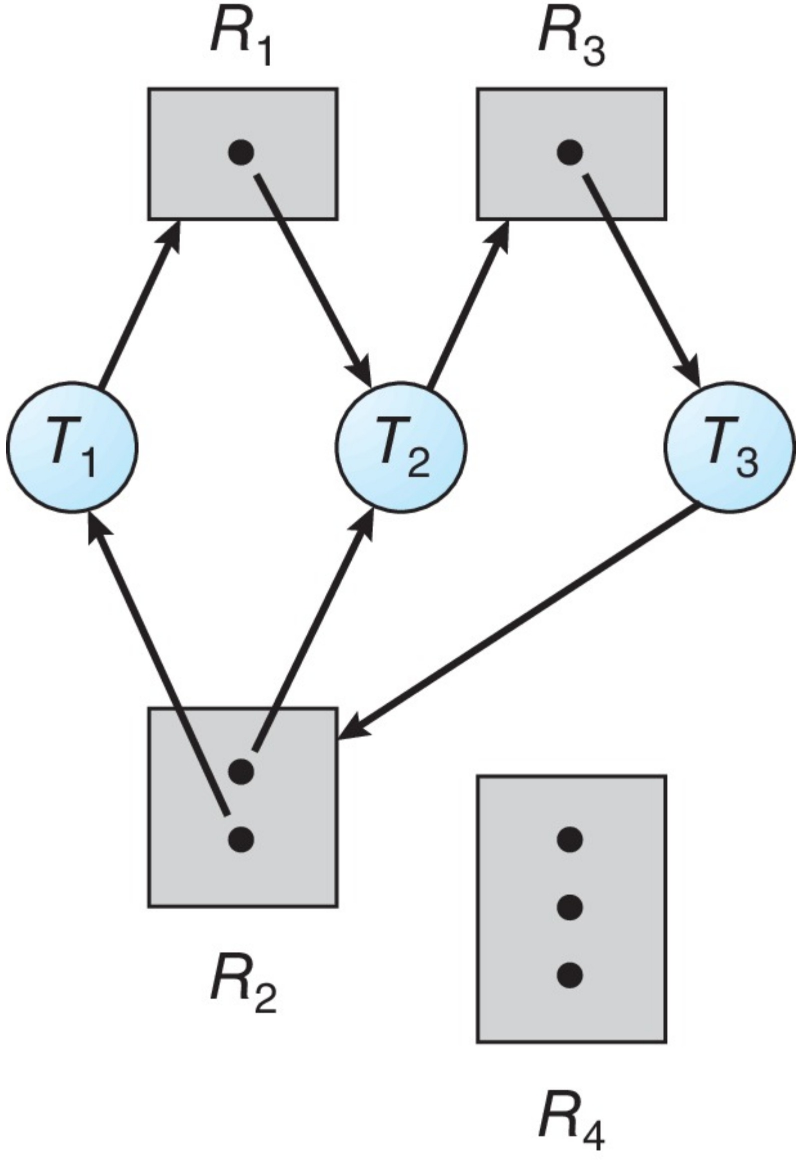
Resource Allocation Graph Example

- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- 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



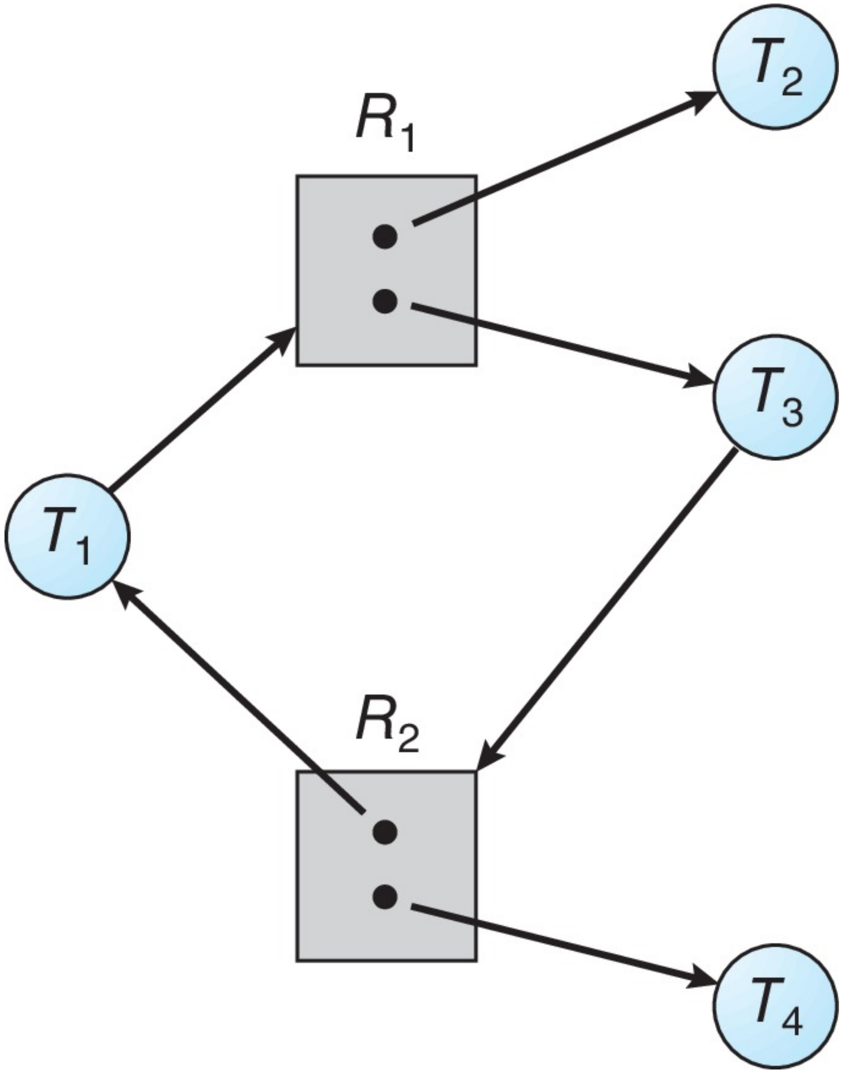
Resource Allocation Graph with a Deadlock

- Processes T1, T2, and T3 are deadlocked.
- Process T2 is waiting for the resource R3, which is held by thread T3.
- Thread T3 is waiting for either thread T1 or thread T2 to release resource R2.
- In addition, thread T1 is waiting for thread T2 to release resource R1.



Graph with a Cycle But no Deadlock

- In this example, we also have a cycle.
- However, there is no deadlock.
- Observe that thread T4 may release its instance of resource type R2.
- That resource can then be allocated to T3, breaking the cycle.



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - ✔ If only one instance per resource type, then deadlock
  - ✔ If several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state:
    - Deadlock prevention
    - Deadlock avoidance
- Allow the system to enter a deadlock state, detect it, and then recover
- Ignore the problem and pretend that deadlocks never occur in the system.

Deadlock Prevention

- Deadlock prevention provides a set of methods to ensure that at **least one of the necessary conditions cannot hold**.

Deadlock Prevention

- Invalidate one of the four necessary conditions for deadlock:
- **Mutual Exclusion** – at least one resource must be non-sharable. Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock.
- **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.
  - Low resource utilization; starvation possible

Deadlock Prevention

- **No Preemption**:
  - If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted.
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait:**
  - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

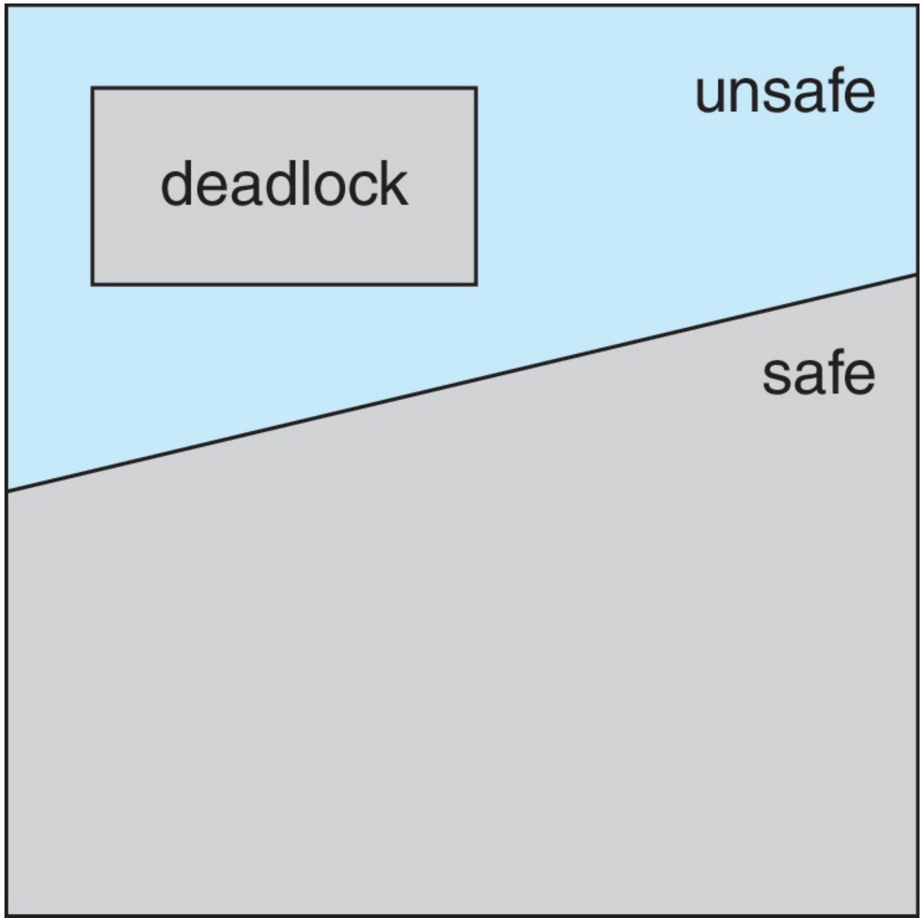
- Requires that the system has some additional a priori information available
- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure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



Avoidance Algorithms

- Single instance of a resource type
  - Use a modified resource-allocation graph

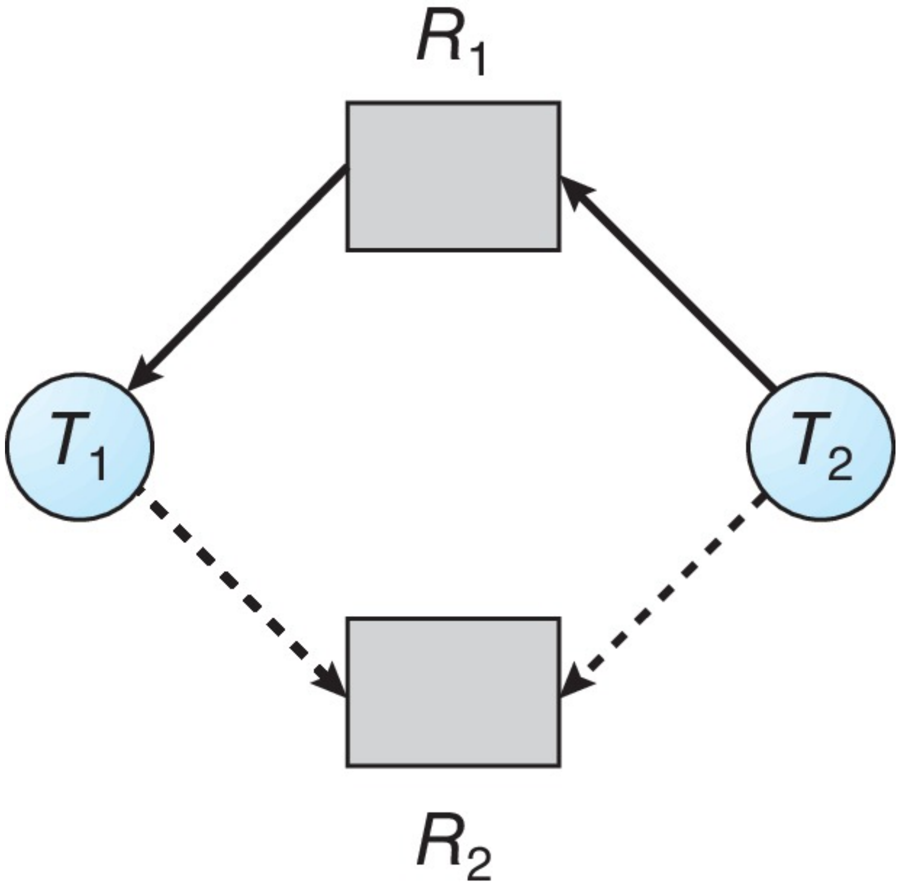
- Multiple instances of a resource type
  - Use the Banker’s Algorithm

Modified Resource-Allocation Graph Scheme

- **Claim edge** P_i --> R_j indicates that process P_i may request resource R_j
- **Request edge** $P_i \rightarrow R_j$ indicates that process P_i requests resource R_j
  - Claim edge converts to request edge when a process requests a resource.
- **Assignment edge** $R_j \rightarrow P_i$ indicates that resource R_j was allocated to process P_i
  - Request edge converts 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.

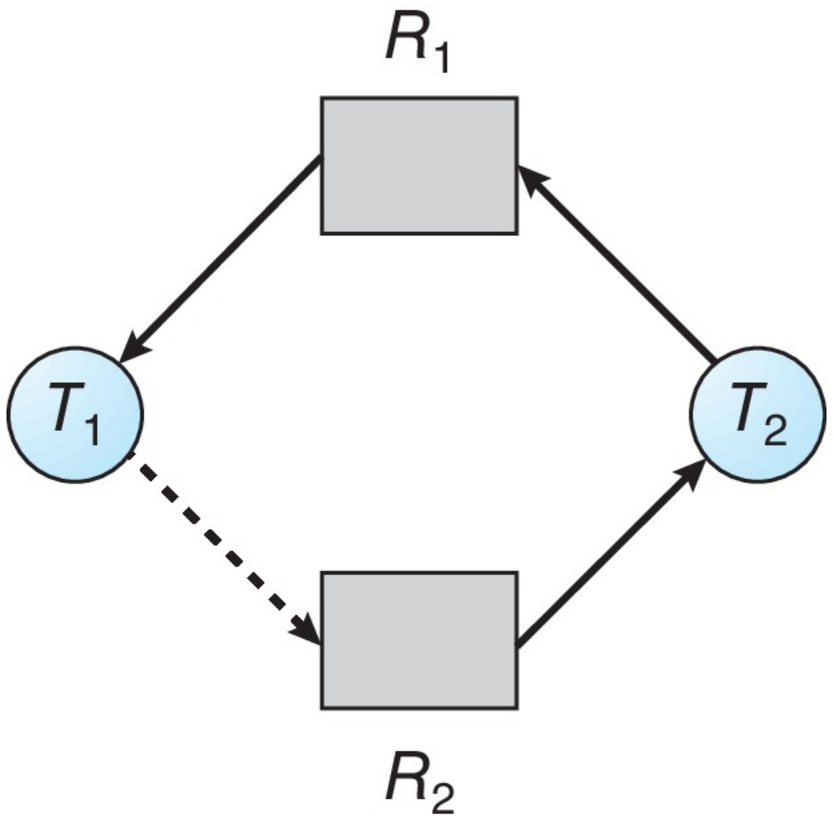
Resource-Allocation Graph

- Process Ti requests resource Rj.
- The request can be granted only if converting the request edge Ti → Rj to an assignment edge R j → Ti does not result in the formation of a **cycle** in the resource-allocation graph.



Unsafe State In Resource-Allocation Graph

- If no cycle exists, then the allocation of the resource will leave the system in a **safe state**.
- If a cycle is found, then the allocation will put the system in an **unsafe state**.



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 x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: n x 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***$_i$ **≤ *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 $\boldsymbol{P_i}$. If ***Request**$_i$* **[*j*] = *k then process $\boldsymbol{P_i}$ wants ***k*** instances of resource type $\boldsymbol{R_j}$

1. If ***Request**$_i$* ≤ ***Need**$_i$* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If ***Request**$_i$* ≤ ***Available***, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available
3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

   ***Available = Available – Request**$_i$;*

   ***Allocation**$_i$ = **Allocation**$_i$ + **Request**$_i$;*

   ***Need**$_i$ = **Need**$_i$ – **Request**$_i$;*

- If safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$
- If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ 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* (5 instances), 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 of Banker's Algorithm

- Snapshot at time T_0:

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		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) $\Rightarrow$ 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 < $\boldsymbol{P_1, P_3, P_4, P_0, P_2}$> satisfies safety requirement

- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?

- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

Deadlock Detection

- Allow system to enter deadlock state
  - ✔ If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.

- Detection algorithm

- Recovery scheme

Single Instance of Each Resource Type

- If all resources have only a single instance, then 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.

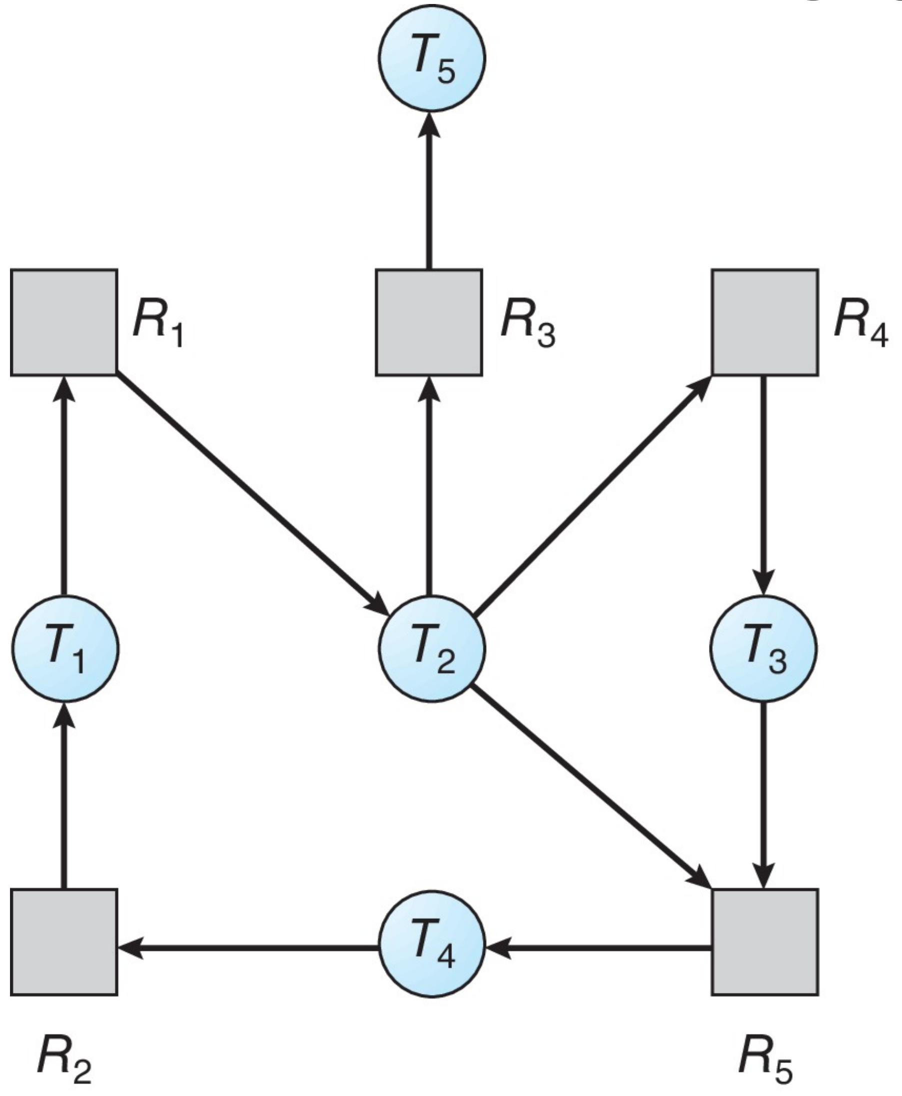
Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are processes
  - $\boldsymbol{P_i} \rightarrow \boldsymbol{P_j}$ if $\boldsymbol{P_i}$ is waiting for $\boldsymbol{P_j}$

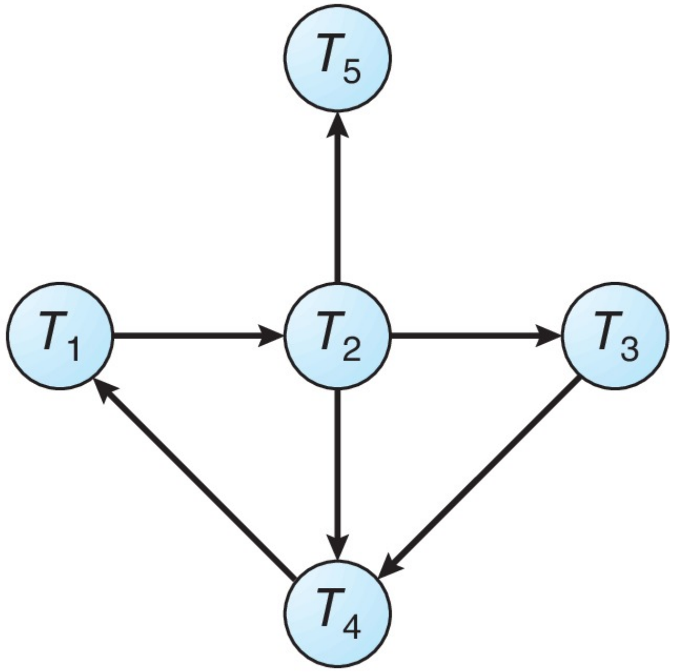
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

- An algorithm to detect a cycle in a graph requires an order of $\boldsymbol{n^2}$ operations, where $\boldsymbol{n}$ is the number of vertices in the graph

Resource-Allocation Graph & Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph

Several Instances of a Resource Type

- The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.
- We turn now to a deadlockdetection algorithm that is applicable to such a system.
- The algorithm employs several time-varying data structures that are similar to those used in the banker’s algorithm.

Several Instances of a Resource Type

- **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 $\boldsymbol{P_i}$ is requesting ***k*** more instances of resource type $\boldsymbol{R_j}$.

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 ≤ ***i*** ≤ ***n***, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then ***P***$_i$ is deadlocked

Detection Algorithm

- Algorithm requires an order of O(m x n2) 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 of Detection Algorithm

- ***P_2*** requests an additional instance of type ***C***

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	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 need to be rolled back?
    - ▪ One for each disjoint cycle

- 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

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system recover from the deadlock automatically.
- There are two options for breaking a deadlock.
  - ✔ One is simply to abort one or more threads to break the circular wait.
  - ✔ The other is to preempt some resources from one or more of the deadlocked threads.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - ✔ Priority of the process
  - ✔ How long process has computed, and how much longer to completion
  - ✔ Resources the process has used
  - ✔ Resources process needs to complete
  - ✔ How many processes will need to be terminated
  - ✔ Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

Summary

- Deadlock: situation in which a set of threads/processes cannot proceed because each requires resources held by another member of the set.
- Prevention: design resource allocation strategies that guarantee that one of the necessary conditions never holds
- Avoidance: don't allocate a resource if it would introduce a cycle.
  - ✔ Safe State
  - ✔ Single instance of a resource type
    - Use a modified resource-allocation graph
  - ✔ Multiple instances of a resource type
    - Use the Banker's Algorithm
- Detection and recovery: recognize deadlock after it has occurred and break it.