Deadlocks

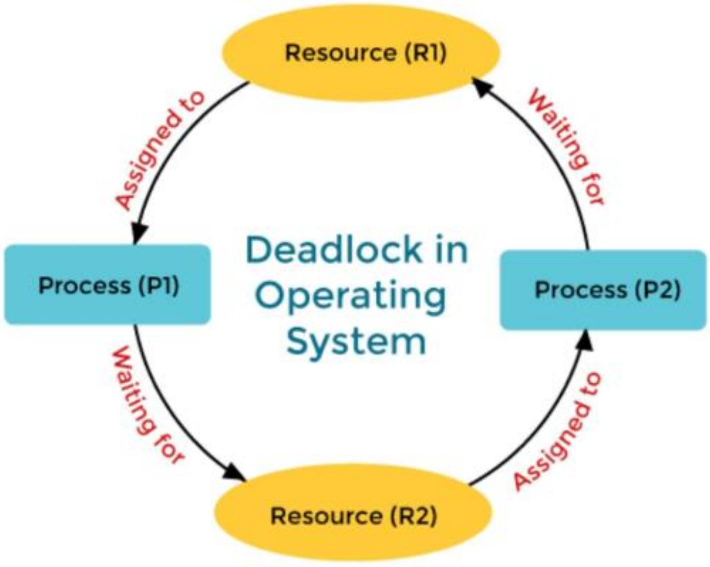
Course Code: CSC 2209 Course Title: Operating Systems

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	10	Week No:	10	Semester:	

Lecture Outline

1. System Model
2. Deadlock in Multithreaded Applications
3. Deadlock Characterization
4. Methods for Handling Deadlocks
5. Deadlock Prevention
6. Deadlock Avoidance
7. Deadlock Detection
8. Recovery from Deadlock



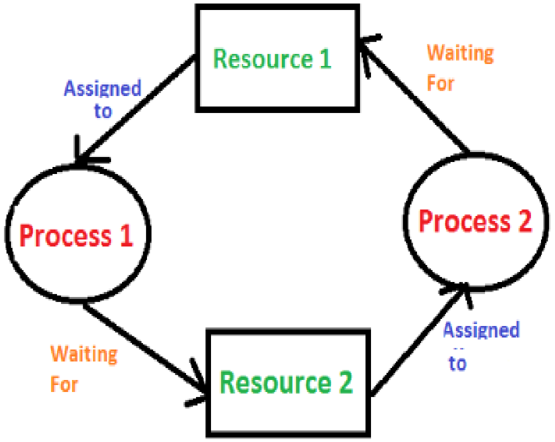
System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
    - *CPU cycles, memory space, I/O devices*
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows (Sequence of use):
    - **Request (ask OS for a resource instance)**
    - **Use (if granted)**
    - **Release (done with the instance)**

Deadlock in Multithreaded Application

- Two mutex locks are created an initialized:

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

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



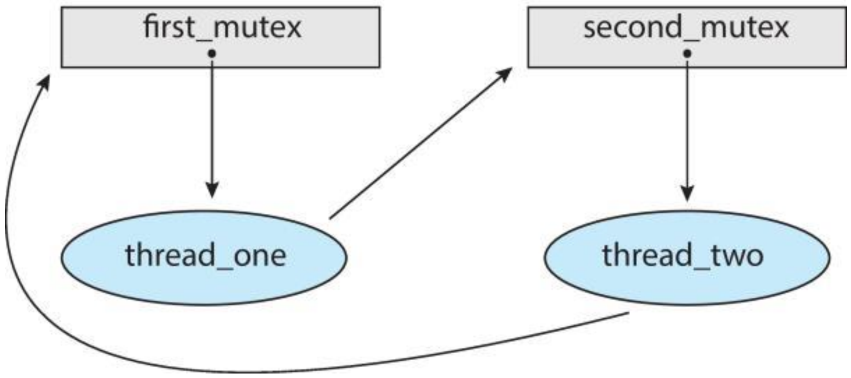
Deadlock in Multithreaded Application

```
/* 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);
}
```

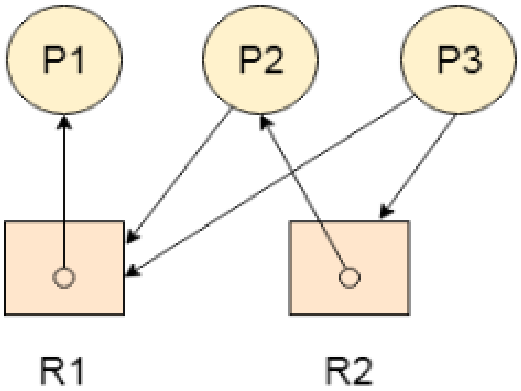
Deadlock in Multithreaded Application

- 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 (RAG)**:



Deadlock



Deadlock Free

The resource allocation graph based on the description:

P1 ----> R2
P2 ----> R3
P3 ----> R1
R1 ----> P1
R2 ----> P2
R3 ----> P3

Constructing and Analyzing a Resource Allocation Graph

You are given the following situation:

- **Process P1** is holding **Resource R1** and requesting **Resource R2**.
- **Process P2** is holding **Resource R2** and requesting **Resource R3**.
- **Process P3** is holding **Resource R3** and requesting **Resource R1**.

Question:

1. Construct the resource allocation graph based on the description above.
2. Is there a deadlock? If yes, which processes and resources are involved?

Yes, there is a deadlock. The reasoning:

- **P1** holds **R1** and is waiting for **R2**.
- **P2** holds **R2** and is waiting for **R3**.
- **P3** holds **R3** and is waiting for **R1**.

This creates a **circular wait** between **P1**, **P2**, and **P3**:

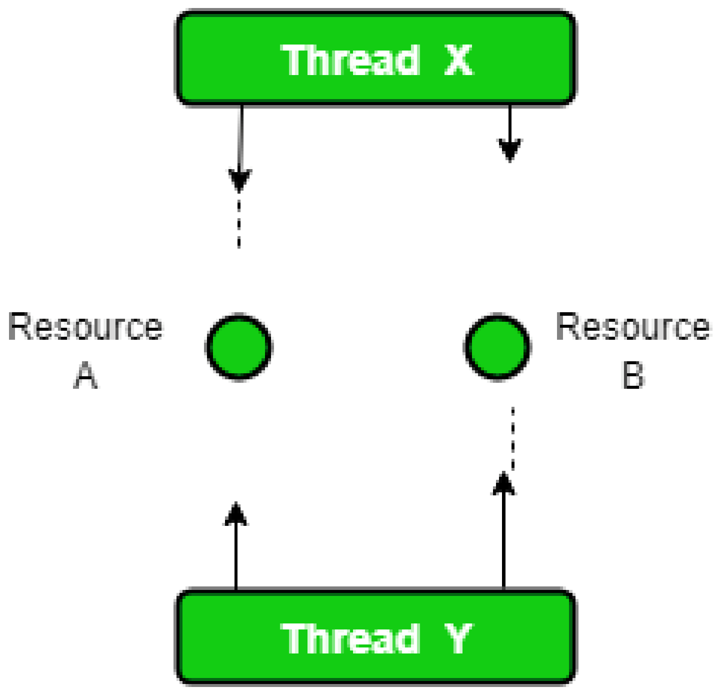
- **P1** is waiting for **R2**, which is held by **P2**.
- **P2** is waiting for **R3**, which is held by **P3**.
- **P3** is waiting for **R1**, which is held by **P1**.

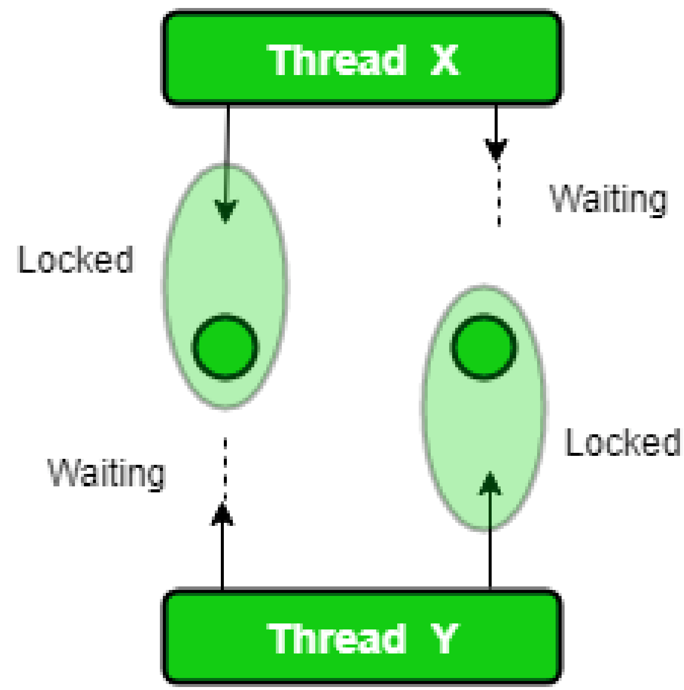
This forms a **deadlock cycle** involving all three processes and resources.

Processes involved: **P1**, **P2**, and **P3**

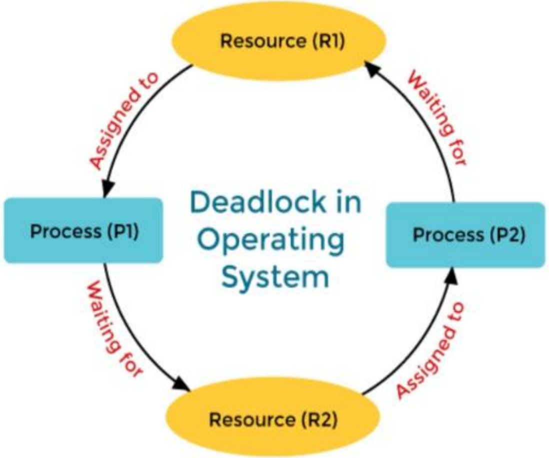
Resources involved: **R1**, **R2**, and **R3**

Deadlock in Multithreaded Application





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 P_0.

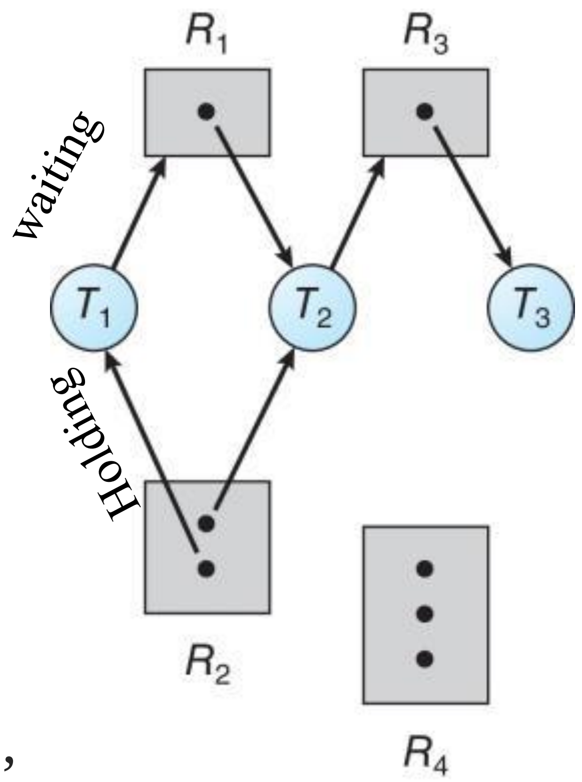
Resource-Allocation Graph (RAG)

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

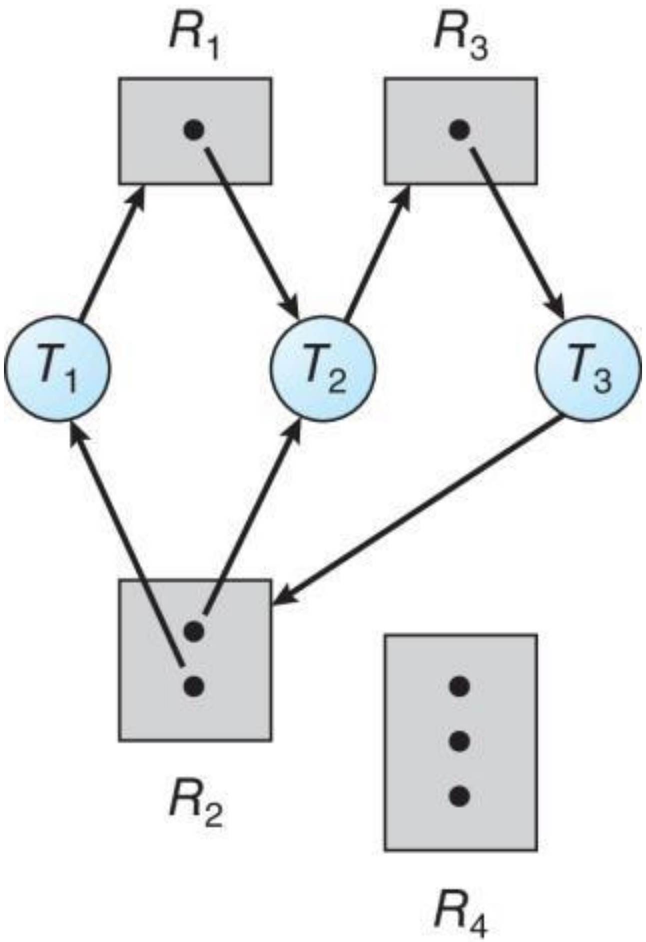
- V is partitioned into **two** types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **Request edge** – directed edge $P_i \rightarrow R_j$
- **Assignment edge** – directed edge $R_j \rightarrow P_i$

Resource Allocation Graph Example

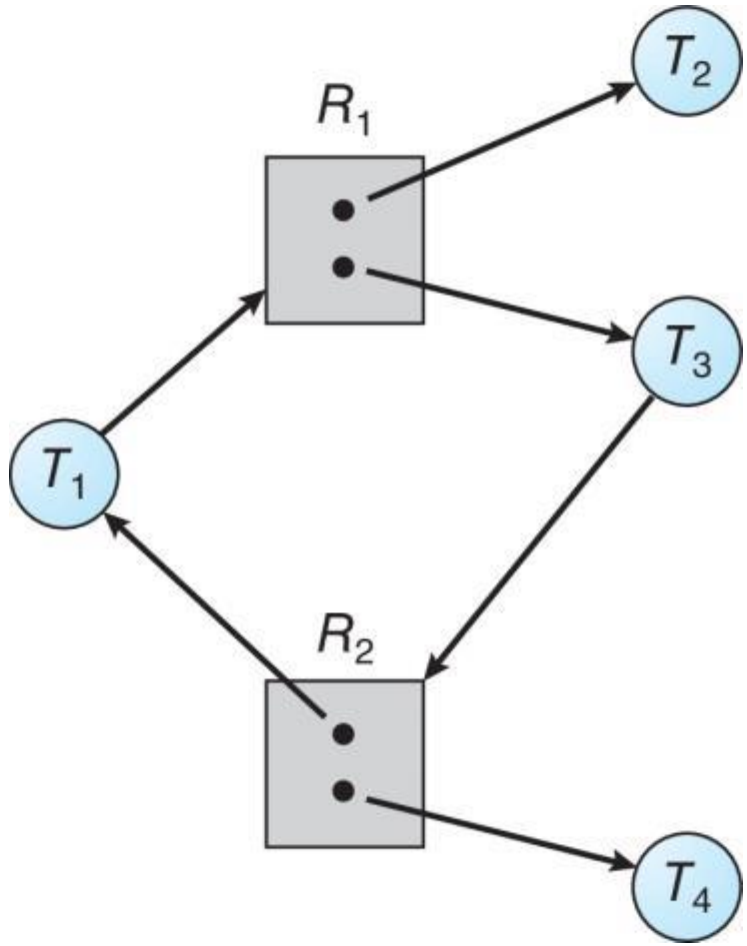
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



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 and then recover
- 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 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 (cont'd)

- **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
  - 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

Circular Wait

- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e. mutex locks) a unique number.
- Resources must be acquired in order.
- 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 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

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks

- If a system is in unsafe state $\Rightarrow$ possibility of deadlock

- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

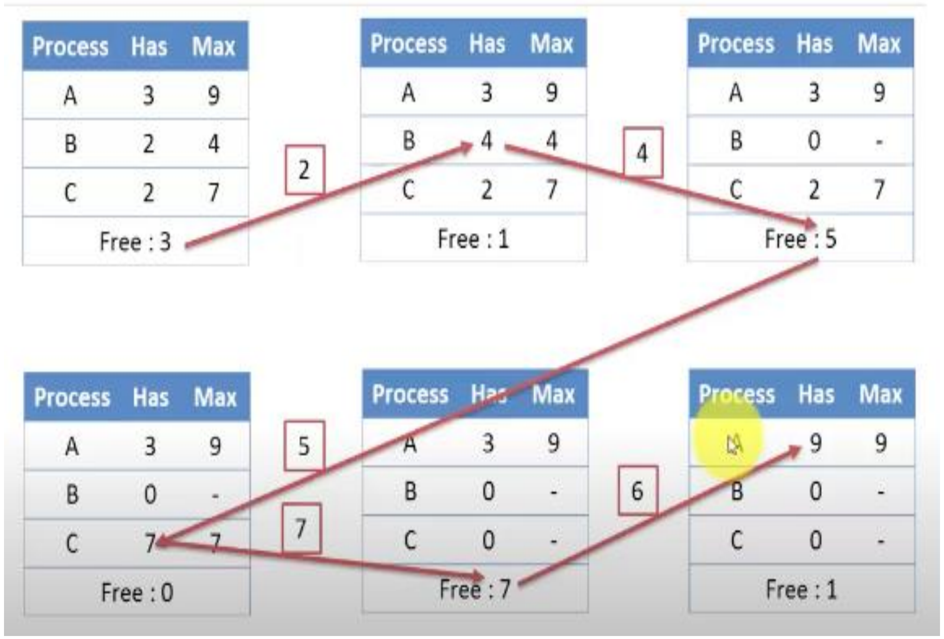
Safe State

- When a process requests an available resource, system/OS must decide if immediate allocation leaves the system in a safe state

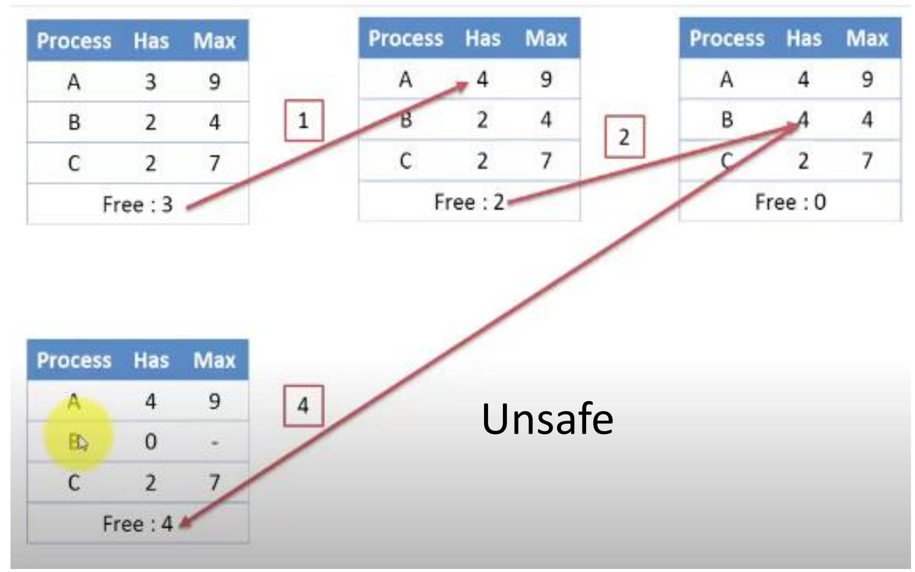
- System is in **Safe State** if there exists a sequence $\langle P_1, P_2, \ldots, P_n \rangle$ of ALL the processes in the systems **such that** for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$

- 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

Safe & Unsafe State Cont. (Example)

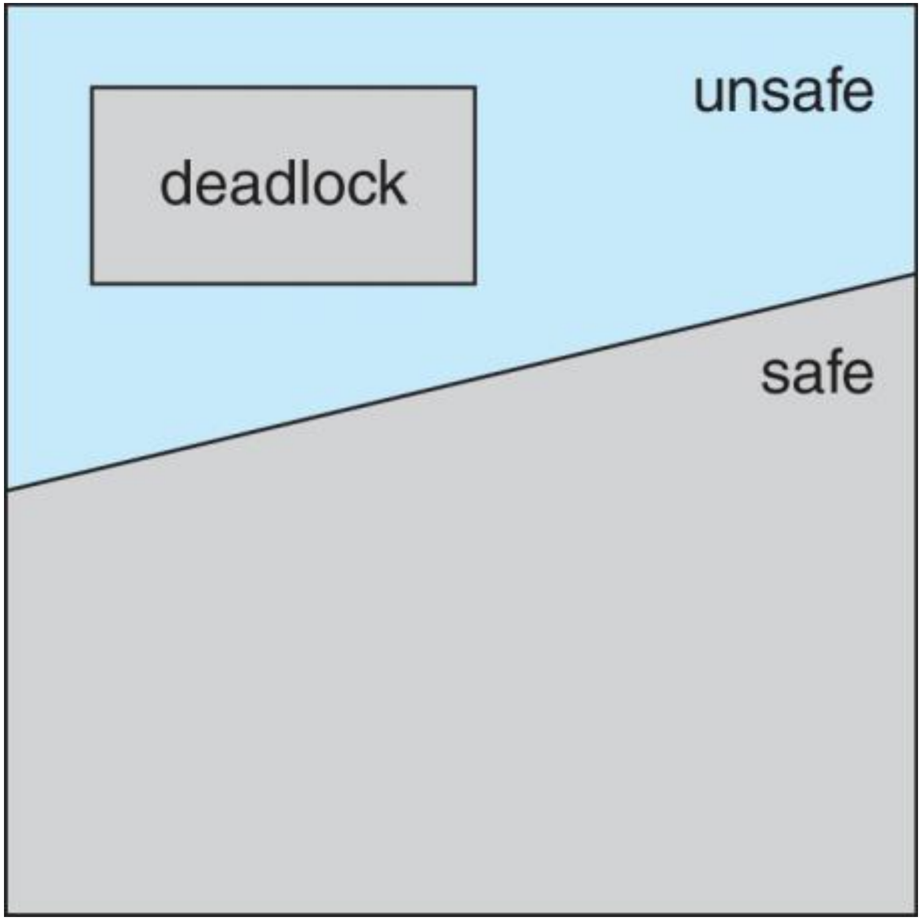


Safe



Unsafe

Safe, Unsafe, Deadlock State



Avoidance Algorithms

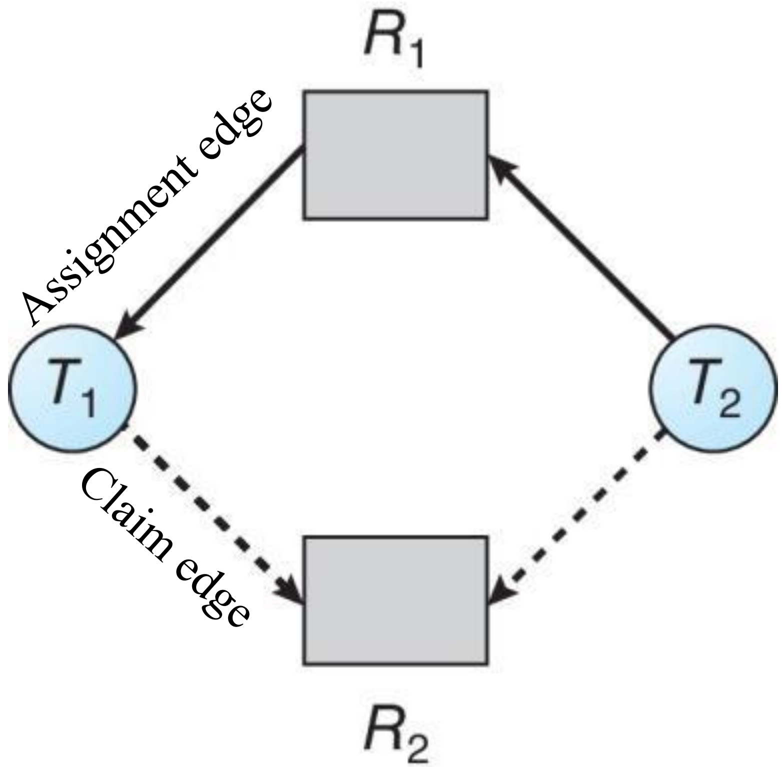
- Single instance of a resource type
  - Use a **Resource Allocation Graph** (RAG)

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

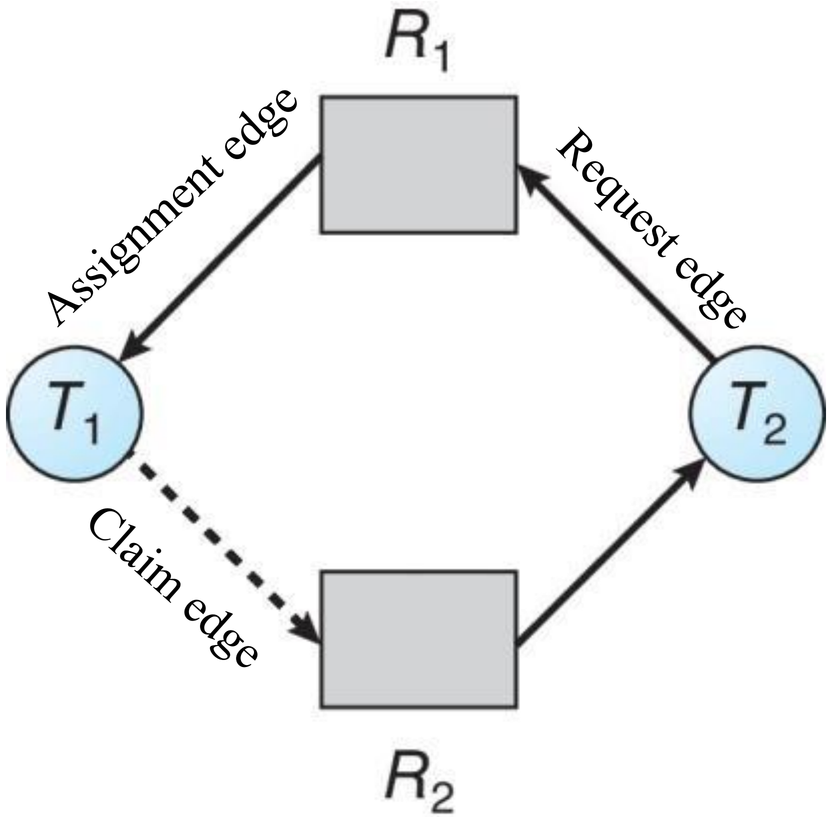
Resource-Allocation Graph Scheme

- **Claim edge** P_i - - - - $\rightarrow R_j$ indicated that process P_i may request resource R_j; represented by a **dashed line**
- Claim edge converts to **request edge** (**solid directed line**) when a process requests a resource $P_i \rightarrow R_j$
- Request edge converted to an **assignment edge** when the resource is allocated to the process $R_j \rightarrow P_i$
- 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



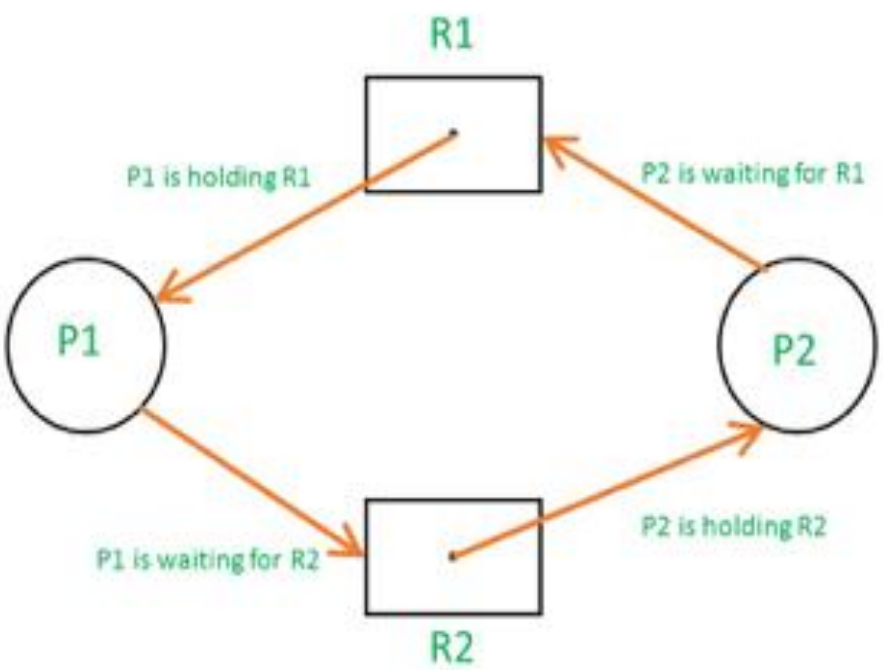
Unsafe State In Resource-Allocation Graph



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

Resource-Allocation Graph Algorithm (Example- Single Instance)



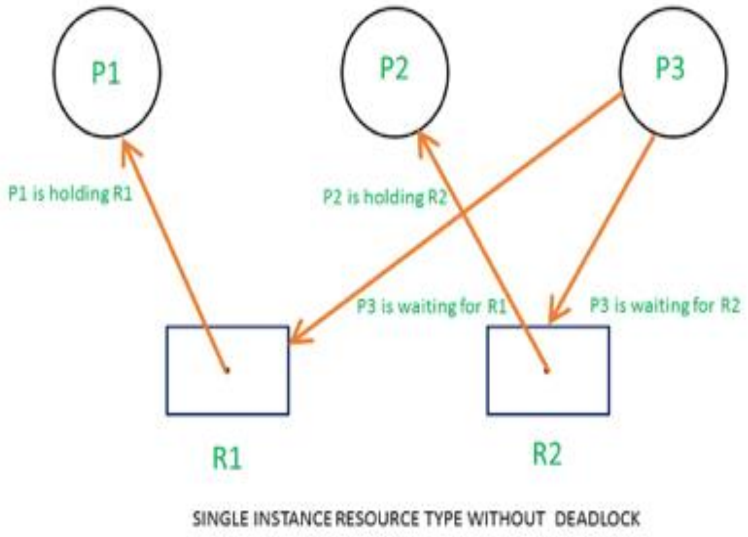
SINGLE INSTANCE RESOURCE TYPE WITH DEADLOCK

Process	Allocate/Hold	Request
	R1 ----------- R2	R1 ------------- R2
P1	1--------------0	0----------------1
P2	0--------------1	1-----------------0

	R1 ----------- R2
Check Availability	0----------------0
P1(0,1)	Can not satisfy
P2(1,0)	Can not satisfy

So, No available resource so deadlock

Resource-Allocation Graph Algorithm (Example- Single Instance)

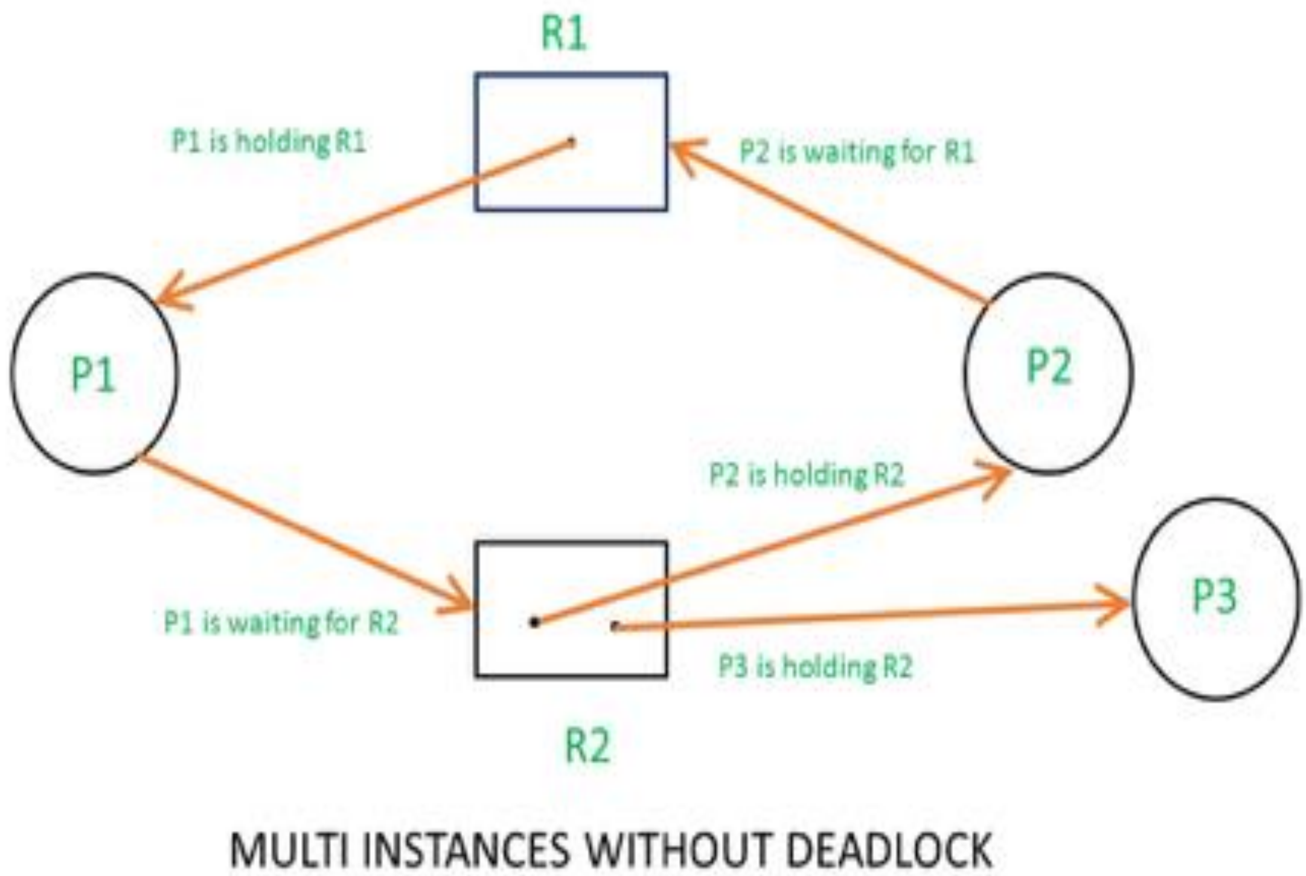


Process	Allocate/Hold	Request
	R1 ----------- R2	R1 ------------- R2
P1	1--------------0	0-----------------0
P2	0--------------1	0-----------------0
P3	0--------------0	1-----------------1

Process	Need/Request	Availability	Executed and Terminated
	R1 ----------- R2	R1 -------------- R2	
P1	0-----------------0	0----------------1	Done and resource release
P2	0-----------------0	1----------------0	Done and resource release
P3	1-----------------1	1----------------1	Done and resource release

Resource-Allocation Graph Algorithm (Example- Multiple Instance) Exercise

Deadlock or not? Identify



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 x 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) $\boldsymbol{Need_i \leq Work}$

   If no such ***i*** exists, go to step 4

3. $\boldsymbol{Work = Work + Allocation_i}$

   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$_i$*** $\leq$ ***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 $\boldsymbol{P_i}$ must wait, since resources are not available
3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

$$\boldsymbol{Available = Available - Request_i;}$$

$$\boldsymbol{Allocation_i = Allocation_i + Request_i;}$$

$$\boldsymbol{Need_i = Need_i - Request_i;}$$

- 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 from P_0 to P_4;
  3 resource types:
  A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time T_0:

Process	*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'd)

- 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 < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted immediately? (Home task)
- Can request for (0,2,0) by P_0 be granted immediately? (Home task)

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

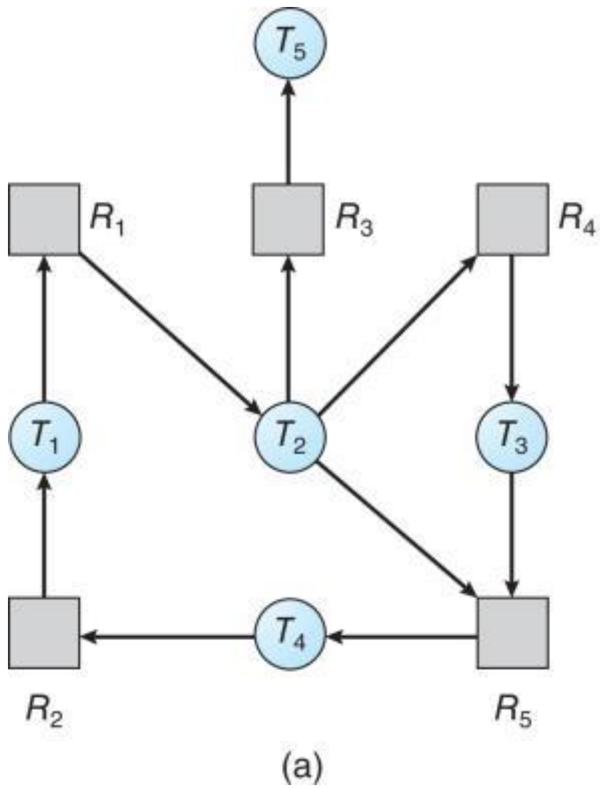
Single Instance of Each Resource Type

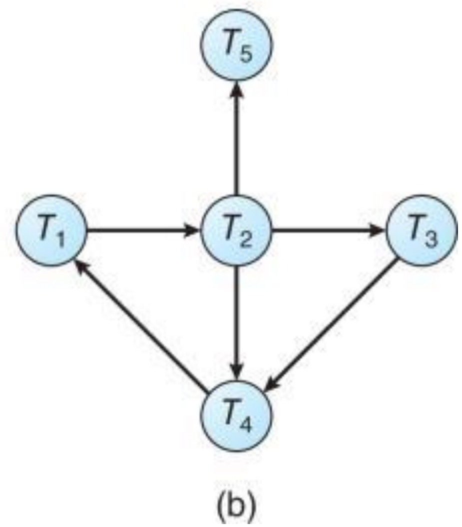
- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j

- 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 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

- **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

Detection Algorithm (cont'd)

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 deadlock state. Moreover, if ***Finish*[*i*] == *false***, then P_i is deadlocked

Algorithm requires an order of O(m x 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'd)

- $\boldsymbol{P_2}$ 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 $\boldsymbol{P_0}$, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes $\boldsymbol{P_1}$, $\boldsymbol{P_2}$, $\boldsymbol{P_3}$, and $\boldsymbol{P_4}$

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

- Abort all deadlocked processes

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

- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

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

Books

- Operating Systems Concept
  - Written by Galvin and Silberschatz
  - Edition: 9th

References

- Operating Systems Concept
  - Written by Galvin and Silberschatz
  - Edition: 9th