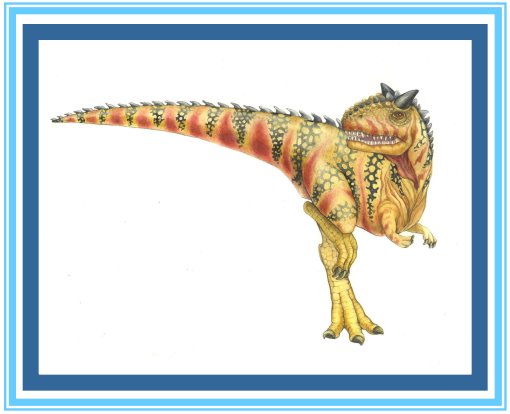
Chapter 7: Deadlocks



Chapter 7: Deadlocks

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

Chapter Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system

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:
  - **request**
  - **use**
  - **release**

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.

Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc.
- See example box in text page 318 for mutex deadlock

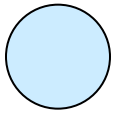
Resource-Allocation Graph

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

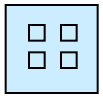
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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

Resource-Allocation Graph (Cont.)

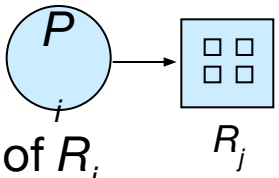
- Process



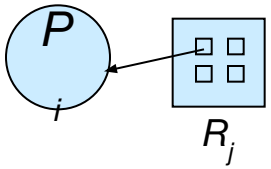
- Resource Type with 4 instances



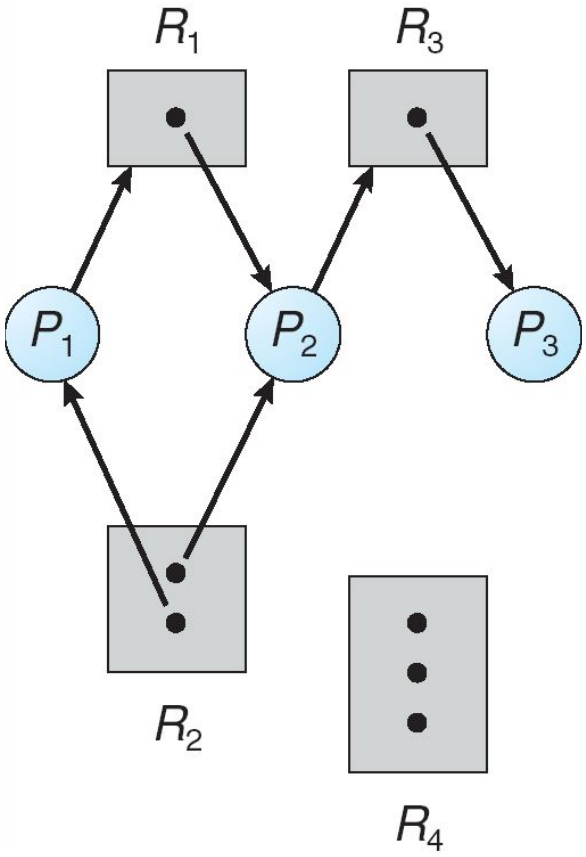
- P_i requests instance of R_j



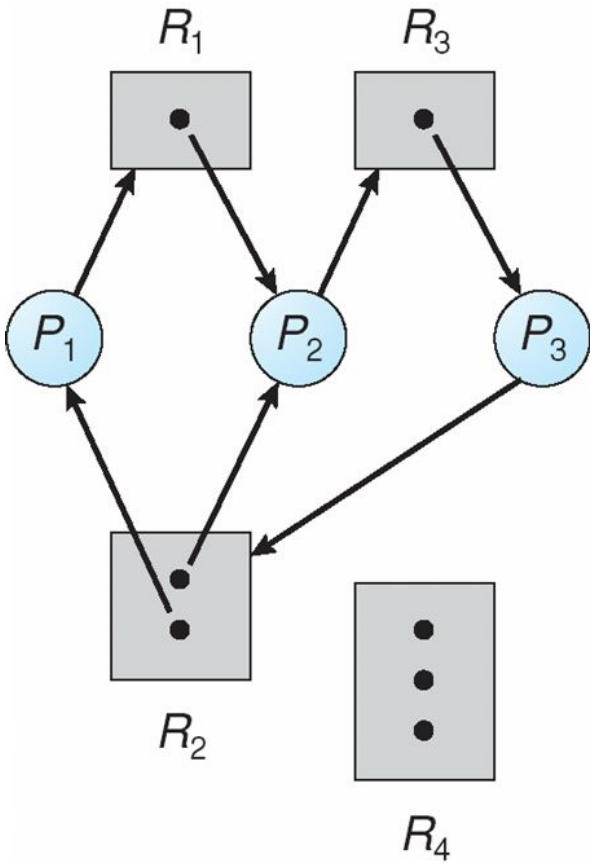
- P_i is holding an instance of R_j



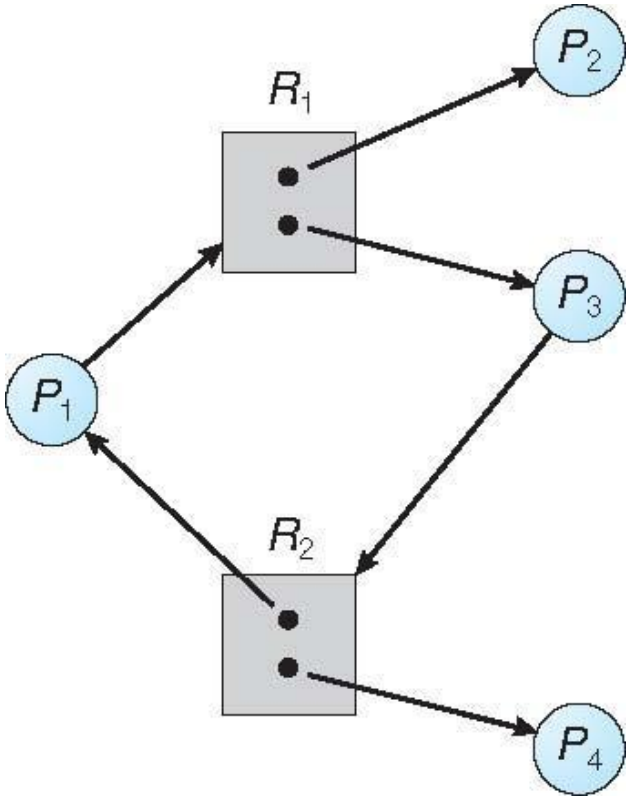
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - 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 avoidence
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made

- **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.)

- **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

Deadlock Example

```
/* 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 Example with Lock Ordering

```
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);
}
```

Transactions 1 and 2 execute concurrently. Transaction 1 transfers $25 from account A to account B, and Transaction 2 transfers $50 from account B to account A

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

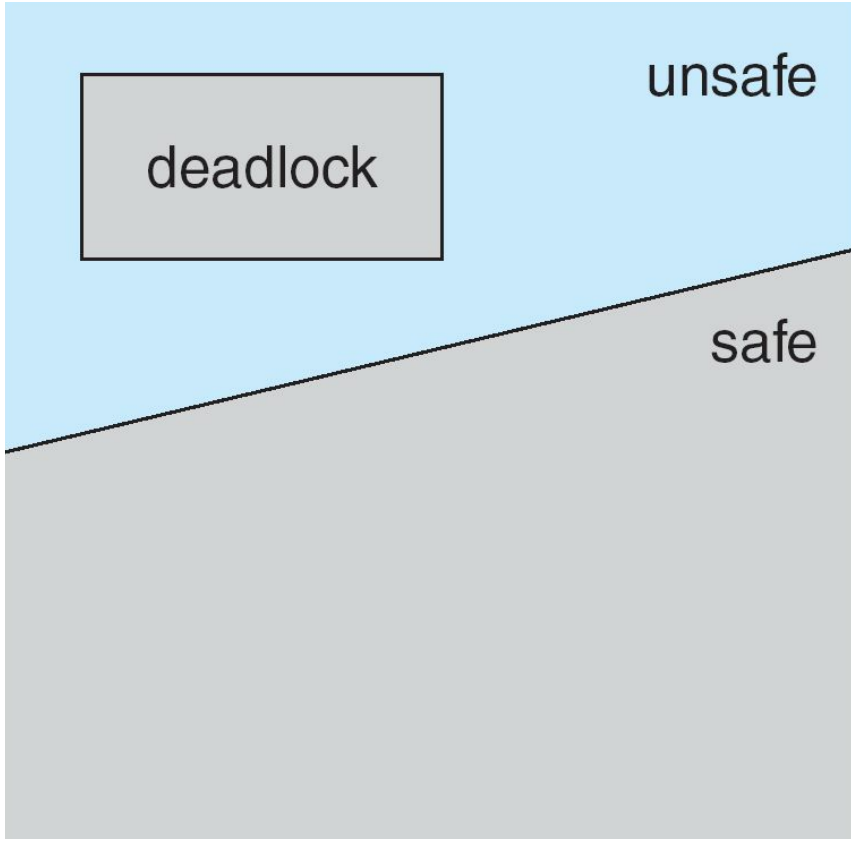
Safe State

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

Basic Facts

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

Safe, Unsafe, Deadlock State



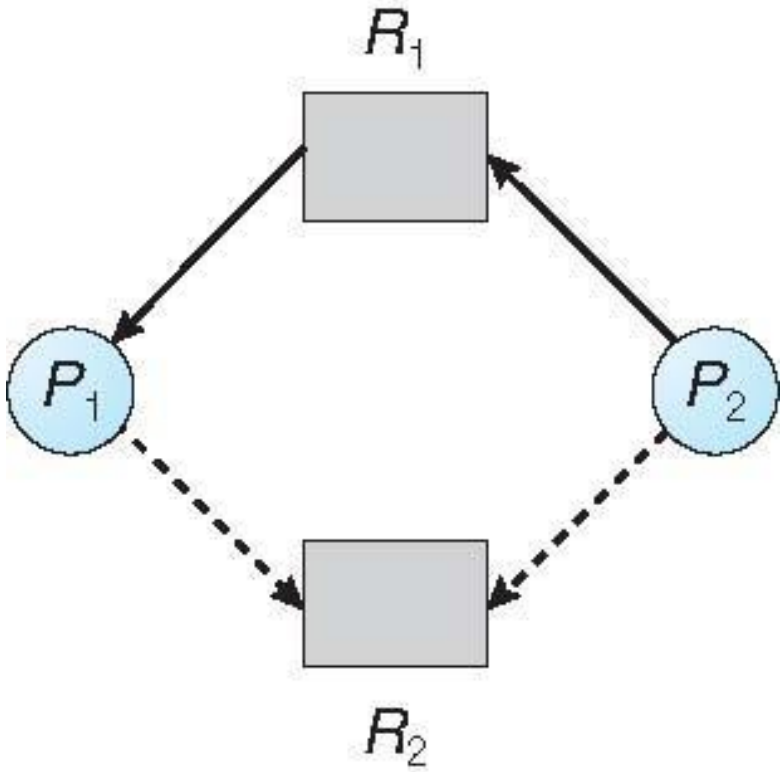
Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

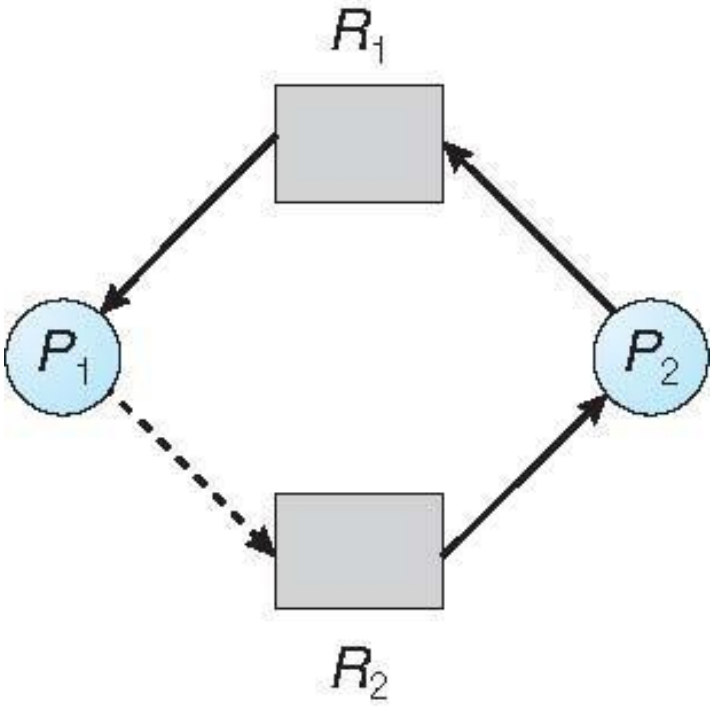
Resource-Allocation Graph Scheme

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

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

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

Banker's Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available
- **Max**: $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: $n \times m$ matrix. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: $n \times m$ matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$

Safety Algorithm

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

   Work* = *Available

   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

   ***Finish* [*i*] = *false* && $Need_i \leq Work$**

   then goto step 3

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

   Each element of Needvector of i^{th} process must be <= each element of work vector

3. *// As ith process i sfinished add allocated resources of ith process to work*

   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

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

1. If $\boldsymbol{Request_i \leq Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\boldsymbol{Request_i \leq Available}$, 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 ⇒ the resources are allocated to $\boldsymbol{P_i}$
- If unsafe ⇒ $\boldsymbol{P_i}$ must wait, and the old resource-allocation state is restored

Advantages of Bankers Algorithm in OS

Some advantages of Bankers Algorithm in OS are mentioned below:

- **Deadlock avoidance:** The Bankers algorithm in OS can detect and avoid deadlocks, ensuring the safe and efficient allocation of resources.
- **Resource utilization:** The Bankers algorithm helps to optimize resource utilization by ensuring that resources are only allocated to processes that need them.
- **System stability:** The Bankers algorithm helps to maintain system stability by ensuring that processes do not hold onto resources indefinitely.
- **Safe state detection:** The Bankers algorithm can detect when the system is in a safe state, allowing for the allocation of resources to be safely managed.
- **Simplicity:** The Bankers algorithm is relatively simple and straightforward to implement, making it a popular choice for resource allocation in operating systems.

Drawbacks of Bankers Algorithm in OS

Some drawbacks of Bankers Algorithm in OS are as follows:

- **Overhead:** The Bankers algorithm requires additional computation and memory overhead to maintain data structures and check for safe states, affecting system performance.
- **Inflexible resource allocation:** The Bankers algorithm follows a set of predetermined rules for resource allocation, which may not be flexible enough to handle complex or dynamic resource requirements.
- **Limited applicability:** The Bankers algorithm in OS is limited in its applicability to only those systems with a finite number of resources, and may not be suitable for systems with continuous or infinite resources.
- **Blocking processes:** The Bankers algorithm may lead to processes being blocked, waiting for resources to become available, causing decreased system performance.
- **Static resource allocation:** The Bankers algorithm in OS uses a static approach to resource allocation, and does not account for changes in resource requirements over time, leading to sub-optimal resource utilization.

Example of Banker's Algorithm

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

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

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

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

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

- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Let process P_0 made a request <2,3,0>

(i) **Construct new AllocationMatrix, New available and new need matrix**

(ii) Check Whether the system is Safe or NOT

(iii) Was P_0 request is granted or not?

Example: P_0 Request (2,3,0)

- Check that P_0 Request ≤ Available (that is, **(2,3,0)** ≤ (3,3,2)) . Assume the request is granted then P_0 then P_0 allocation <0,1,0> + <2,3,0> = <2,4,0> and P_0 need becomes < 7,5,3> - <2,4,0> = <5,1,3> and Available become <3,3,2> - <2,3,0> = <1,0,2>

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	2 4 0	5 1 3	1 0 2
P_1	2 0 0		1 2 2
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

With this sate none of the process Need will be satisfied thus system is unsafe.

Hence, the request of P_0 will not be granted

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

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

- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Let process P_0 made a request <2,3,0>
(i) Construct new need matrix
(ii) Check Wether the system is Safe or NOT
(iii) Was P_0 request is granted or not?

Example: P_1 Request (1,0,2)

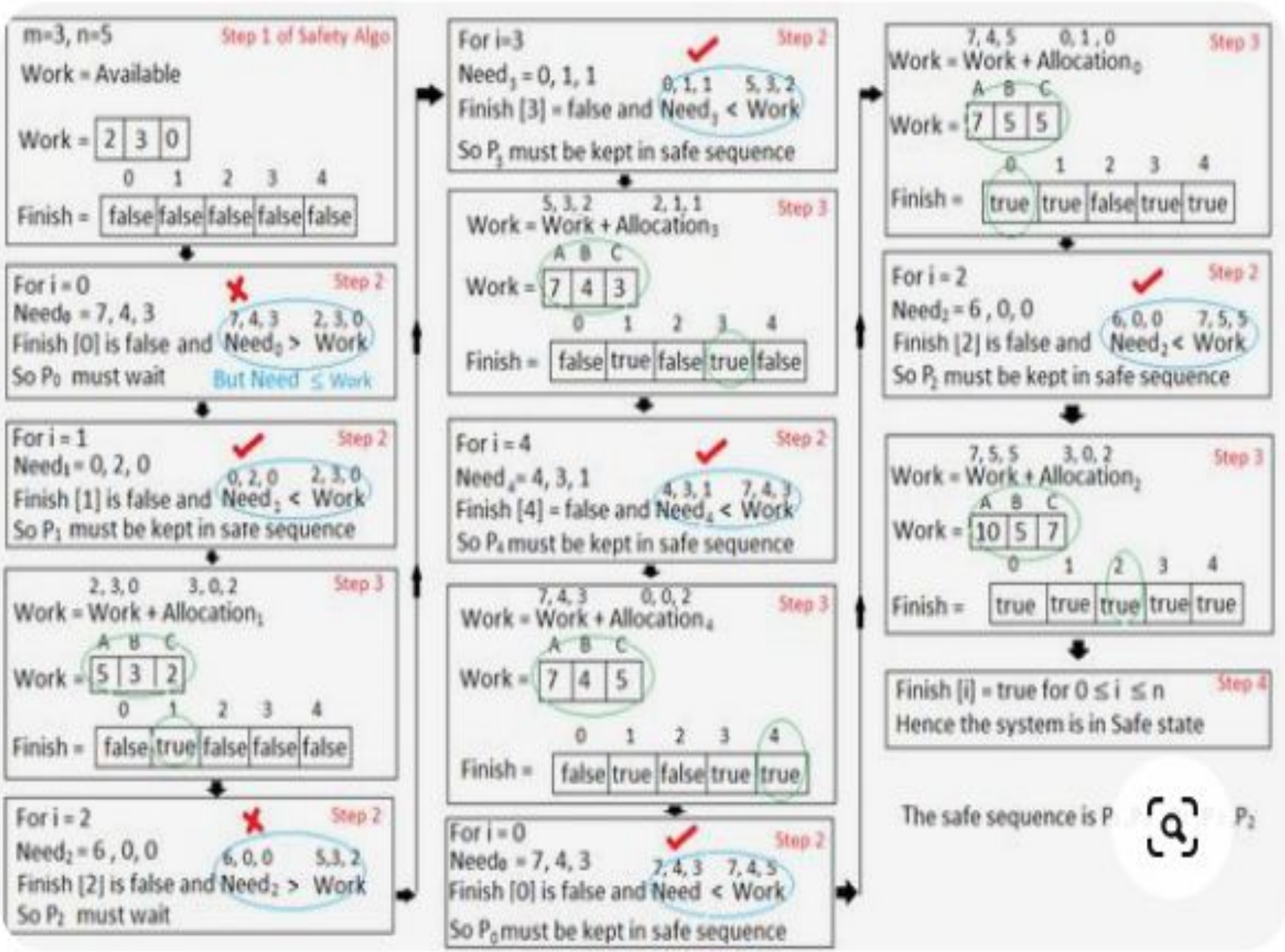
- As P_1 Request$_1$ ≤ Need$_1$ and Request$_1$ ≤ Available$_1$ ((1,0,2) <= (1,2,2) && **(1,0,2)** ≤ (3,3,2)), presume allocation to P_1 has done and compute new state with this allocation. Allocation$_1$=Allocation$_1$+(1,0,2), Need$_1$=Need$_1$ +(1,0,2) & available = available - (1,0,2). The new state is

	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 the system . Hence request for procees $\boldsymbol{P_1}$ is granted.

From here

- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted? **NO (why ?)**
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted? **NO (why ?)**


m=3, n=5
Step 1 of Safety Algo
Work = Available
Work = 2 3 0
0 1 2 3 4
Finish = false false false false false
For i = 0
Step 2
Need0 = 7, 4, 3
7, 4, 3 2, 3, 0
Finish [0] is false and Need0 > Work
So P0 must wait
But Need ≤ Work
For i = 1
Step 2
Need1 = 0, 2, 0
0, 2, 0 2, 3, 0
Finish [1] is false and Need1 < Work
So P1 must be kept in sate sequence
2, 3, 0 3, 0, 2
Step 3
Work = Work + Allocation1
A B C
Work = 5 3 2
0 1 2 3 4
Finish = false true false false false
For i = 2
Step 2
Need2 = 6 , 0, 0
6, 0, 0 5,3, 2
Finish [2] is false and Need2 > Work
So P2 must wait
For i=3
Step 2
Need3 = 0, 1, 1
0, 1, 1 5, 3, 2
Finish [3] = false and Need3 < Work
So P3 must be kept in safe sequence
5, 3, 2 2, 1, 1
Step 3
Work = Work + Allocation3
A B C
Work = 7 4 3
0 1 2 3 4
Finish = false true false true false
For i = 4
Step 2
Need4 = 4, 3, 1
4, 3, 1 7, 4, 3
Finish [4] = false and Need4 < Work
So P4 must be kept in safe sequence
7, 4, 3 0, 0, 2
Step 3
Work = Work + Allocation4
A B C
Work = 7 4 5
0 1 2 3 4
Finish = false true false true true
For i = 0
Step 2
Need0 = 7, 4, 3
7, 4, 3 7, 4, 5
Finish [0] is false and Need < Work
So P0 must be kept in safe sequence
7, 4, 5 0, 1, 0
Step 3
Work = Work + Allocation0
A B C
Work = 7 5 5
0 1 2 3 4
Finish = true true false true true
For i = 2
Step 2
Need2 = 6 , 0, 0
6, 0, 0 7, 5, 5
Finish [2] is false and Need2 < Work
So P2 must be kept in safe sequence
7, 5, 5 3, 0, 2
Step 3
Work = Work + Allocation2
A B C
Work = 10 5 7
0 1 2 3 4
Finish = true true true true true
Finish [i] = true for 0 ≤ i ≤ n
Step 4
Hence the system is in Safe state
The safe sequence is P
P2

Deadlock Detection

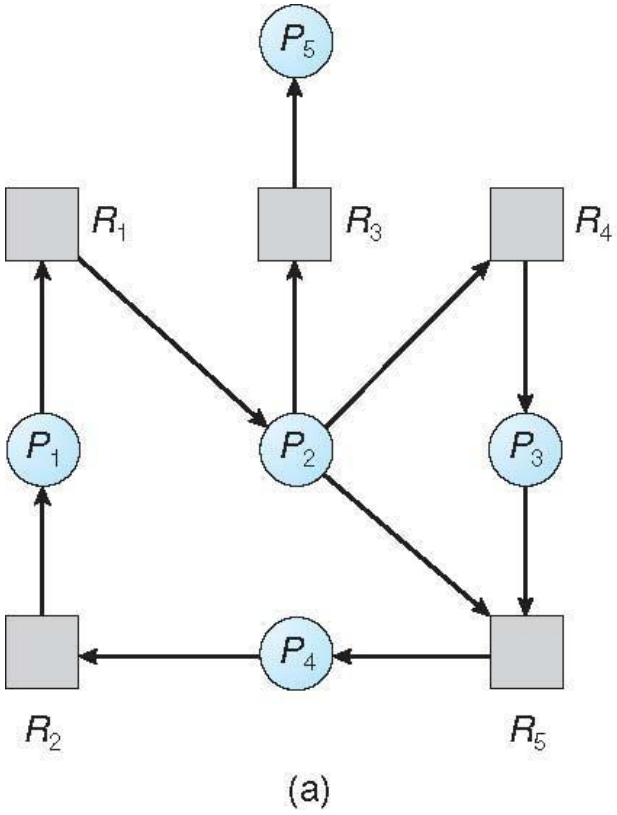
Allow system to enter deadlock state then detect deadlock and Recover from it.

- Detection algorithm (cycle in wait-for-graph)
- Recovery scheme

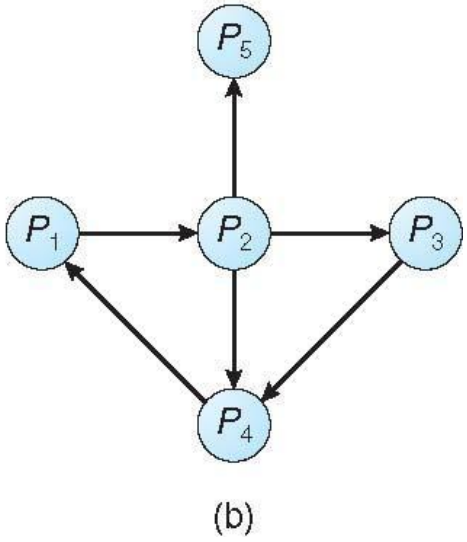
Single Instance of Each Resource Type

- 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 $\boldsymbol{Allocation_i \neq 0}$, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:
   (a) ***Finish*[*i*] == *false***
   (b) $\boldsymbol{Request_i \leq Work}$

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

Detection Algorithm (Cont.)

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

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

Algorithm requires an order of O(m 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.)

- 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

- 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 safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

Let's consider an example below where the **Allocation**, **Max**, and **Available** fields are given to us. This example shows four P0, P1, P2, and P3 processes and three resources, Type A, B, and C. There is also an Available field that shows the currently available resource.

Process	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2			
P1	2	0	0	3	2	2						
P2	3	0	2	7	0	2						
P3	2	1	0	2	2	2						

First we will calculate the Need. The Need field can be calculated using the formula:

Need[i] = Max[i] - Allocation[i]

After applying the formula, we found the following result

Process	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2	7	4	3
P1	2	0	0	3	2	2				1	2	2
P2	3	0	2	7	0	2				4	0	0
P3	2	1	0	2	2	2				0	1	2

Now, let's check the safe sequence.

For Process P0, Need is (7,4,3), and Available is (3,3,2). It shows that the resource needed is more than the available so the system will move to the subsequent process request.

For Process P1, Need is (1,2,2), and the Available is (3,3,2). It shows that the resource needed is less than the Available. So the system will move forward with the request of P1.

Available = Available + Allocation = (3,3,2) + (2,0,0) =(5,3,2)

Now New Available is (5,3,2)

For Process P2, Need is (4,0,0), and the Available is (5,3,2). It shows that the resource needed is less than the Available. So the system will move forward with the request of P2.

Available = Available + Allocation =(5,3,2) + (3,0,2) = (8,3,4)

Now the New Available is (8,3,4)

For Process P3, Need is (0,1,2), and the Available is (8,3,4). It shows that the resource needed is less than the Available. So the system will move forward with the request of P3.

Available = Available + Allocation =(8,3,4) + (2,1,0) = (10,4,4)

Now the New Available is (10,4,4)

For Process P0, Need is (7,4,3), and the Available is (10,4,4). It shows that the resource needed is less than the Available. So the system will move forward with the request of P0.

Available = Available + Allocation =(10,4,4) + (0,1,0) = (10,5,4)

Therefore the Safe Sequence <P1,P2,P3,P0>.

GATE QUESTION

GATE | GATE-CS-2014-(Set-1) | Question 65

An operating system uses the Banker's algorithm for deadlock avoidance when managing the allocation of three resource types X, Y, and Z to three processes P0, P1, and P2. The table given below presents the current system state. Here, the Allocation matrix shows the current number of resources of each type allocated to each process and the Max matrix shows the maximum number of resources of each type required by each process during its execution.

	Allocation			Max		
	X	Y	Z	X	Y	Z
P0	0	0	1	8	4	3
P1	3	2	0	6	2	0
P2	2	1	1	3	3	3

There are 3 units of type X, 2 units of type Y and 2 units of type Z still available. The system is currently in a safe state. Consider the following independent requests for additional resources in the current state:

```
REQ1: P0 requests 0 units of X,
      0 units of Y and 2 units of Z
REQ2: P1 requests 2 units of X,
      0 units of Y and 0 units of Z
```

Which one of the following is TRUE?

(A) Only REQ1 can be permitted.

(B) Only REQ2 can be permitted.

(C) Both REQ1 and REQ2 can be permitted.

(D) Neither REQ1 nor REQ2 can be permitted

Answer (B)

Example of Bankers Algorithm in OS

Let we have a process table with a number of processes that has an allocation column (to show how many resources of type A, B, and C are allocated to each process in the table), a max field (to show how many resources of type A, B, and C can be allotted to each process), and an available field (for showing the currently available resources of each type in the table).

Processes	Allocation	Max	Available
	A B C	A B C	A B C
P0	2 1 0	8 6 3	4 3 2
P1	1 2 2	9 4 3	
P2	0 2 0	5 3 3	
P3	3 0 1	4 2 3	

Safe sequence is

A) P0,P1,P2,P3 B) P1, P0,P2,P3 C) P3, P0,P1,P2

D) P3,P2, P0,P1

Processes	Need
	A B C
P0	6 5 3
P1	8 2 1
P2	5 1 3
P3	1 2 2

Available <4,3,2>

Ans 2 - Let us now check for a safe sequence:

1. In the case of Process P0, Need = (6, 5, 3) and Available = (4, 3, 2) Clearly, the resources required outnumber the ones available. As a result, the system will proceed to the next request.

2. In the case of Process P1, Need = (8, 2, 1) and Available = (4, 3, 2) Clearly, the resources required outnumber the ones available. As a result, the system will proceed to the next request.

3. In the case of Process P2, Need = (5, 1, 3) and Available = (4, 3, 2) Clearly, the resources required outnumber the ones available. As a result, the system will proceed to the next request.

4. In the case of Process P3, Need = (1, 2, 2) and Available = (4, 3, 2) Clearly, the resources required are less than the resources available in the system. As a result, P3's request is approved.

```
Available=Available+Allocation = (4, 3, 2) + (3, 0, 1) = (7, 3, 3)
```

5. Now, Check again for Process P0, Need = (6, 5, 3) and Available = (7, 3, 3) Clearly, the resources required outnumber the ones available. As a result, the system will proceed to the next request.

6. Check again for Process P1, Need = (8, 2, 1) and Available = (7, 3, 3) Clearly, the resources required outnumber the ones available. As a result, the system will proceed to the next request.

7. Check again for Process P2, Need = (5, 1, 3) and Available = (7, 3, 3) Clearly, the resources required are less than equal the resources available in the system. As a result, P2's request is approved.

```
Available=Available+Allocation = (7, 3, 3) + (0, 2, 0) = (7, 5, 3)
```

8. Now,only two processes are left, lets check again for Process P0, Need = (6, 5, 3) and Available = (7, **5** 3). Clearly, the resources required are less than equal the resources available in the system. As a result, P0's request is approved.

```
Available=Available+Allocation = (7, 5, 3) + (2, 1, 0) = (9, 6 3)
```

9. Now,only one process is left, lets check for Process P1, Need = (8, 2, 1) and Available = (9, **6**, 3). Clearly, the resources required are less than equal the resources available in the system. As a result, P1's request is approved.

```
Safe sequence: < P3,P2,P0,P1>
```

End of Chapter 7