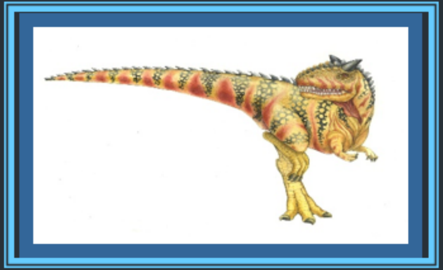
Module 3: L2: Resource allocation and management , RAG, Deadlock



Dr. Rishikeshan C A
SCOPE, VIT Chennai

Outline

- Resource allocation and management
- RAG
- Deadlock
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

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.

Resource Allocation

Resource Allocation may fall under

- Single-tasking Resource Allocation
- Multitasking Resource Allocation
- Real resources Allocation
- Virtual resources Allocation

Resource Allocation
Goals

- Meet resource needs of each program
- Prevent programs from interfering with one another
- Efficiently use hardware and other resources
- Keep detailed records of available resources; know which resources can satisfy which requests
- Schedule resources based on specific allocation policies
- Update records to reflect resource commitment and release by programs and users

Real and Virtual Resources

- Real resources
  - Physical devices and associated system software
- Virtual resources
  - Resources that are apparent to a process or user
  - Meet or exceed real resources by
    - Rapidly shifting resources unused by one program to other programs that need them
    - Substituting one type of resource for another

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**

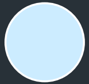
Resource-Allocation Graph

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

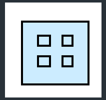
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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

Resource-Allocation Graph (Cont.)

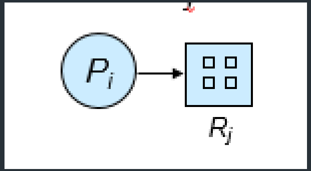
- Process



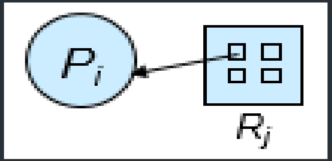
Resource Type with 4 instances



- P_i requests instance of R_j

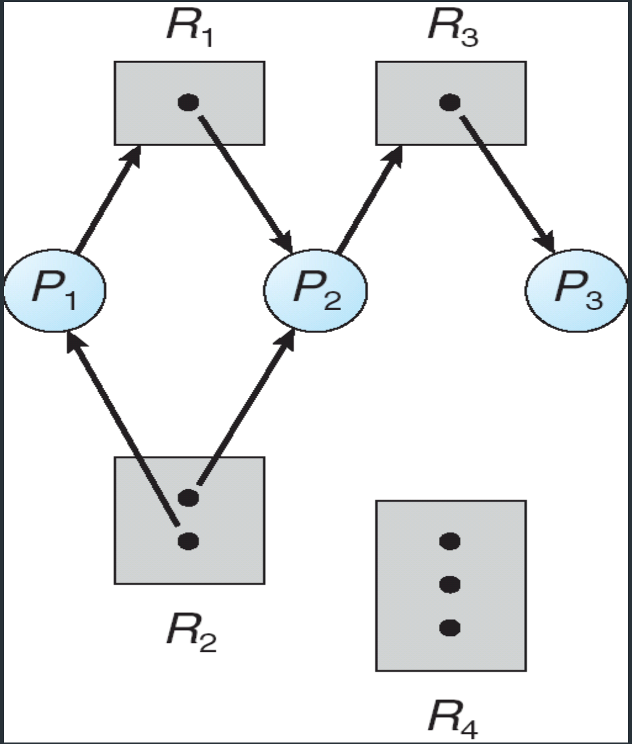


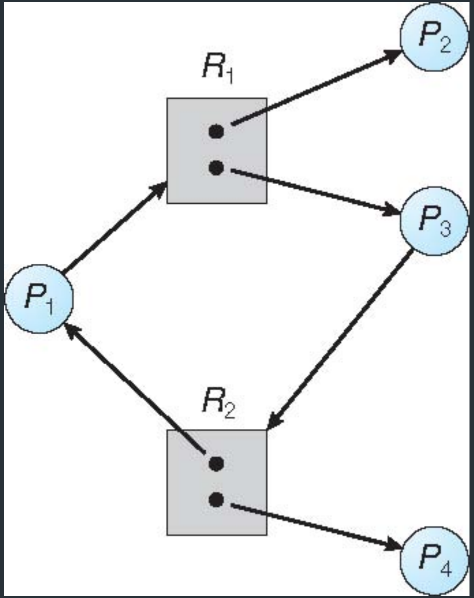
- P_i is holding an instance of R_j



Source: Silberschatz, Gagne, Galvin: Operating System Concepts 9th edition

Example of a Resource Allocation Graph





Example of Graph With A Cycle

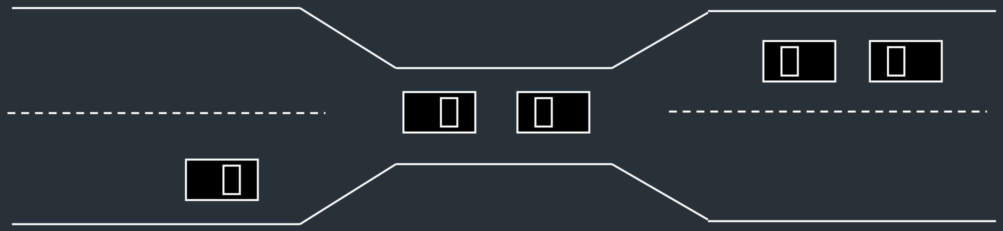
Source: Silberschatz, Gagne, Galvin: Operating System Concepts 9th edition

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 disk drives.
  - P_1 and P_2 each hold one disk drive and each needs another one.
- Example
  - semaphores *A* and *B*, initialized to 1

```
      P0              P1
wait (A);          wait(B)
wait (B);          wait(A)
```

Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

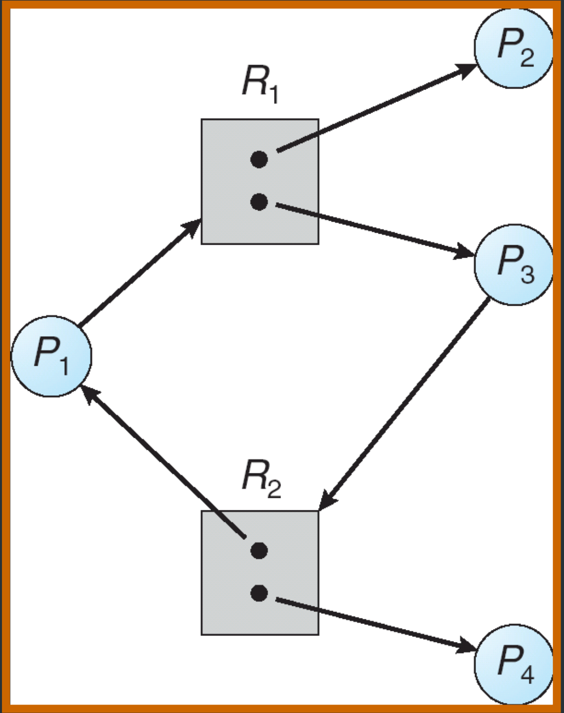
- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ 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

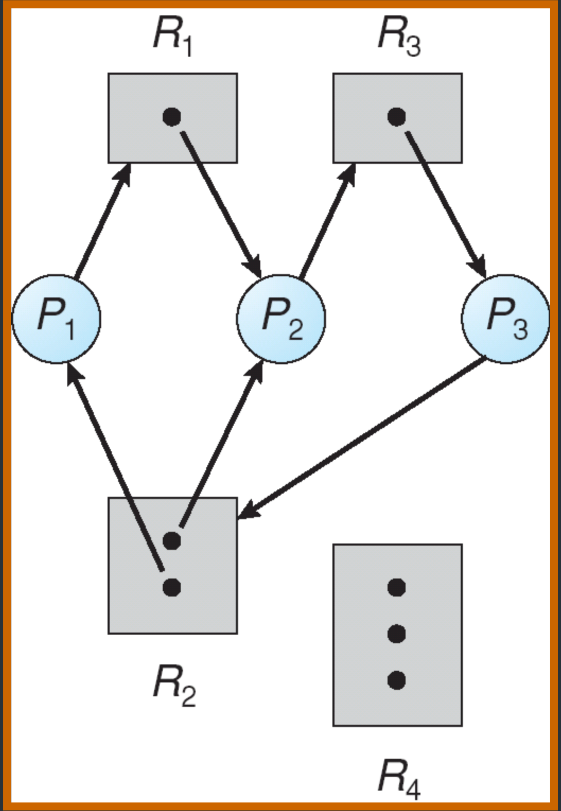
  $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by
  P_n, and P_0 is waiting for a resource that is held by P_0.

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.

Graph With A Cycle But No Deadlock





Resource Allocation Graph With A Deadlock

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
- 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; 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.
  - 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 Avoidance

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

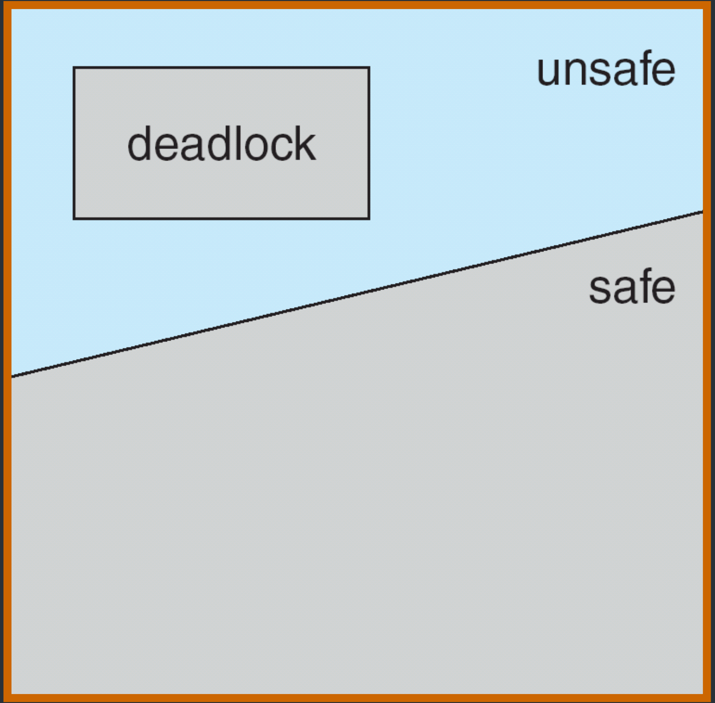
- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *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 is 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

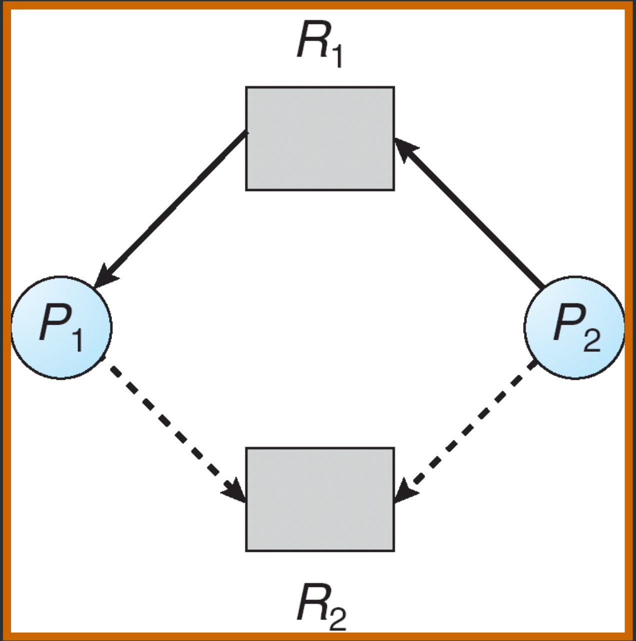
Avoidance algorithms

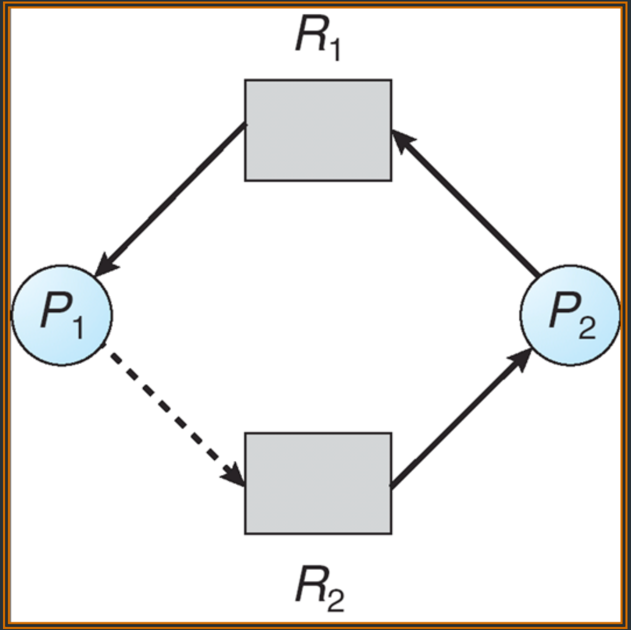
- Single instance of a resource type. Use a resource-allocation graph
- Multiple instances of a resource type. Use the banker’s algorithm

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 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*].

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i. If $Request_i [j] = k$ then process P_i wants k instances of resource type 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 P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

   $Available = Available - Request;$

   $Allocation_i = Allocation_i + Request_i;$

   $Need_i = Need_i - Request_i;$

- *If safe $\Rightarrow$ the resources are allocated to Pi.*
- *If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored*

Safety Algorithm (Banker's 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 and *i* such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

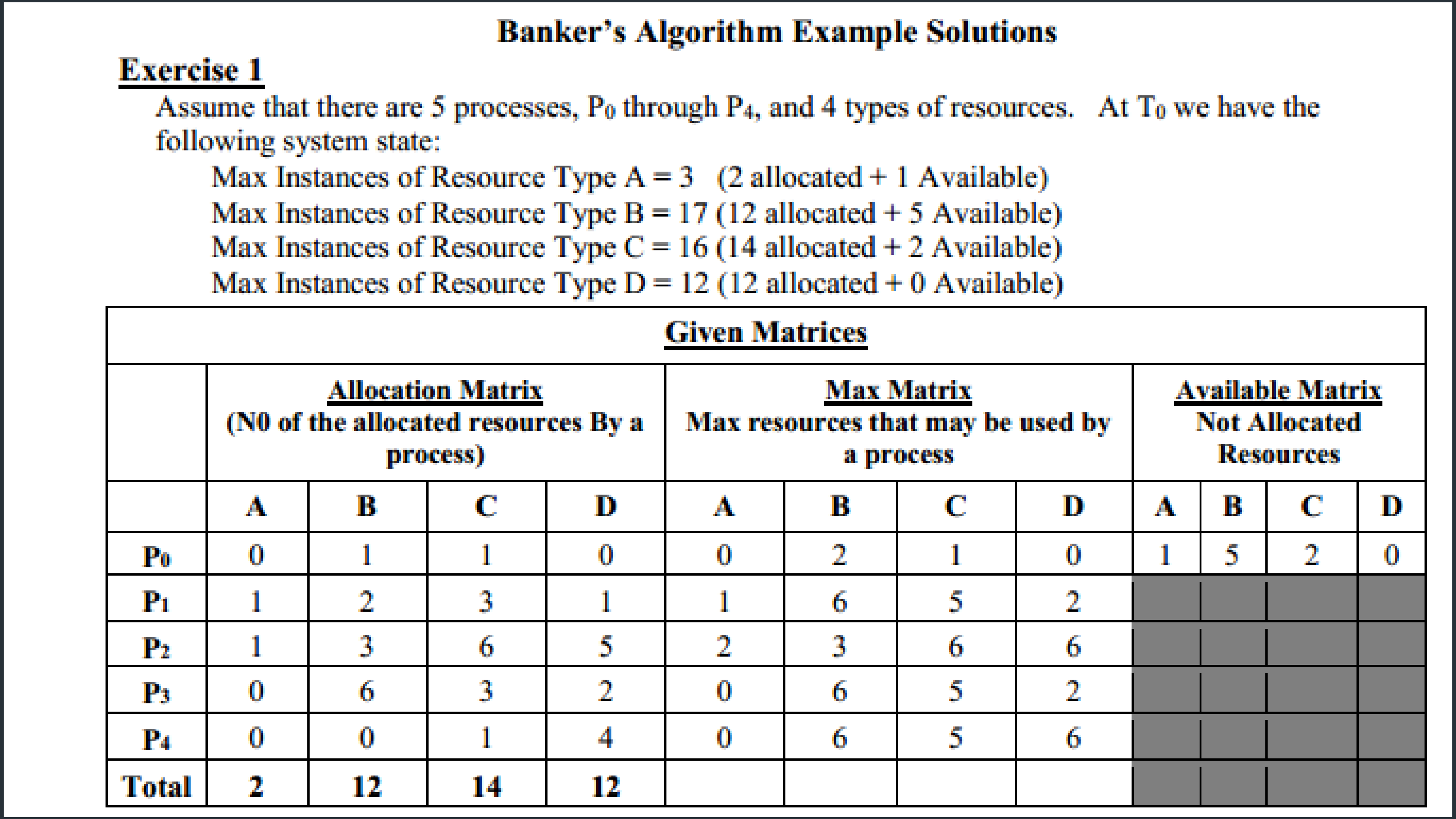
   If no such *i* exists, go to step 4.

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.



Banker's Algorithm Example Solutions

Exercise 1

Assume that there are 5 processes, P_0 through P_4, and 4 types of resources. At T_0 we have the following system state:

Max Instances of Resource Type A = 3 (2 allocated + 1 Available)
Max Instances of Resource Type B = 17 (12 allocated + 5 Available)
Max Instances of Resource Type C = 16 (14 allocated + 2 Available)
Max Instances of Resource Type D = 12 (12 allocated + 0 Available)

Given Matrices												
	Allocation Matrix (N0 of the allocated resources By a process)				**Max Matrix** Max resources that may be used by a process				**Available Matrix** Not Allocated Resources			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P_0	0	1	1	0	0	2	1	0	1	5	2	0
P_1	1	2	3	1	1	6	5	2	▨	▨	▨	▨
P_2	1	3	6	5	2	3	6	6	▨	▨	▨	▨
P_3	0	6	3	2	0	6	5	2	▨	▨	▨	▨
P_4	0	0	1	4	0	6	5	6	▨	▨	▨	▨
Total	**2**	**12**	**14**	**12**					▨	▨	▨	▨

Need(i) = Max(i) – Allocated(i)
(i=0) (0,2,1,0) - (0,1,1,0) = (0,1,0,0)
(i=1) (1,6,5,2) - (1,2,3,1) = (0,4,2,1)
(i=2) (2,3,6,6) - (1,3,6,5) = (1,0,0,1)
(i=3) (0,6,5,2) - (0,6,3,2) = (0,0,2,0)
(i=4) (0,6,5,6) - (0,0,1,4) = (0,6,4,2)

Ex. Process P1 has max of (1,6,5,2) and allocated by (1,2,3,1)
Need(p1) = max(p1)- allocated(p1) = (1,6,5,2) – (1,2,3,1) = (0,4,2,1)

Need Matrix = Max Matrix – Allocation Matrix

	A	**B**	**C**	**D**
P_0	0	1	0	0
P_1	0	4	2	1
P_2	1	0	0	1
P_3	0	0	2	0
P_4	0	6	4	2

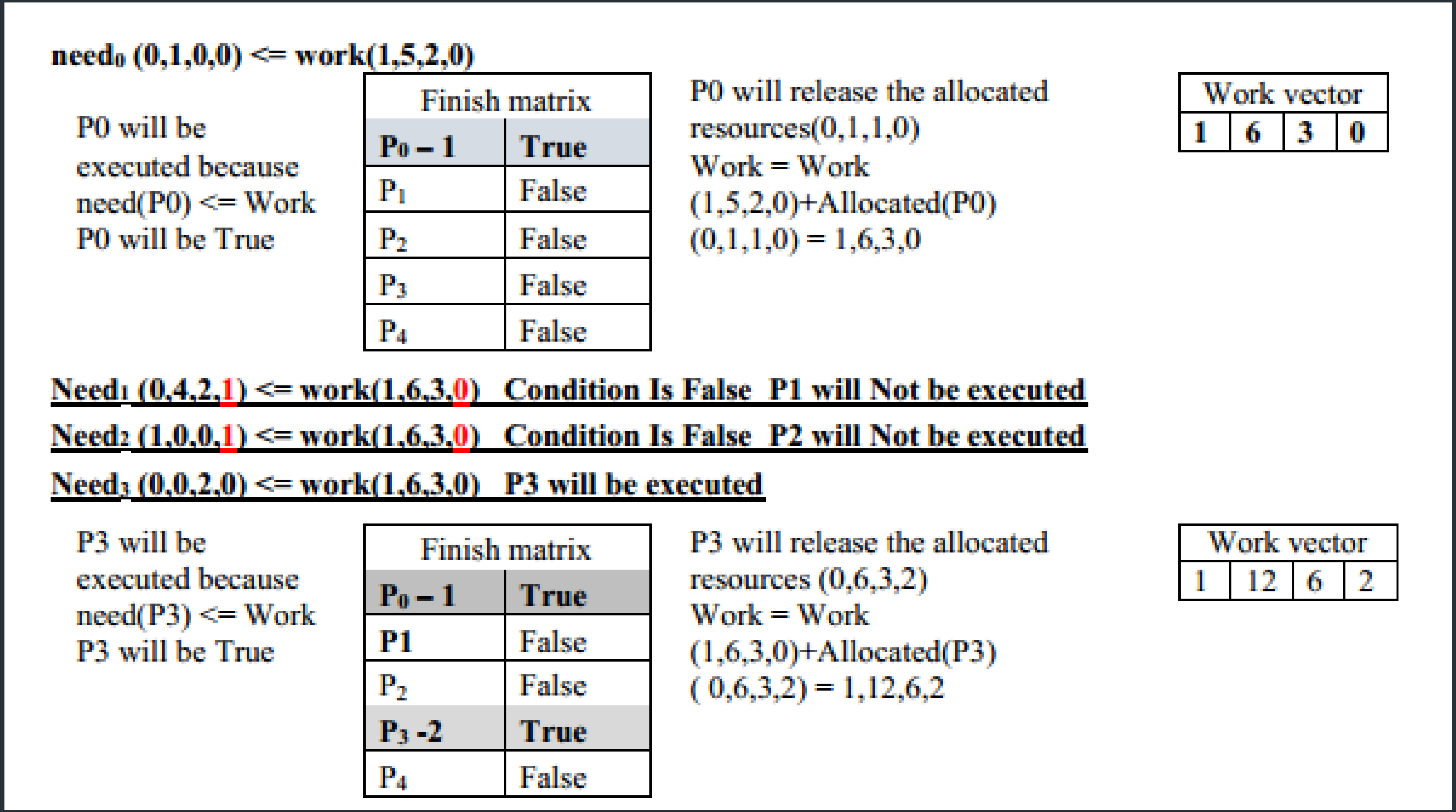
2. **Use the safety algorithm to test if the system is in a safe state or not?**
   a. **We will first define work and finish:**

Initially work = available = (1, 5 , 2, 0)
Finish = False for all processes

Finish matrix	
P_0	False
P_1	False
P_2	False
P_3	False
P_4	False

Work vector			
1	**5**	**2**	**0**

   b. **Check the needs of each process [needs(pi) <= Max(pi)], if this condition is true:**
      - **Execute the process , Change Finish[i] =True**
      - **Release the allocated Resources by this process**
      - **Change The Work Variable = Allocated (pi) + Work**



need$_0$ (0,1,0,0) <= work(1,5,2,0)

P0 will be executed because need(P0) <= Work P0 will be True

Finish matrix	
P$_0$ – 1	**True**
P$_1$	False
P$_2$	False
P$_3$	False
P$_4$	False

P0 will release the allocated resources(0,1,1,0) Work = Work (1,5,2,0)+Allocated(P0) (0,1,1,0) = 1,6,3,0

Work vector			
1	**6**	**3**	**0**

Need$_1$ (0,4,2,1) <= work(1,6,3,0) Condition Is False P1 will Not be executed

Need$_2$ (1,0,0,1) <= work(1,6,3,0) Condition Is False P2 will Not be executed

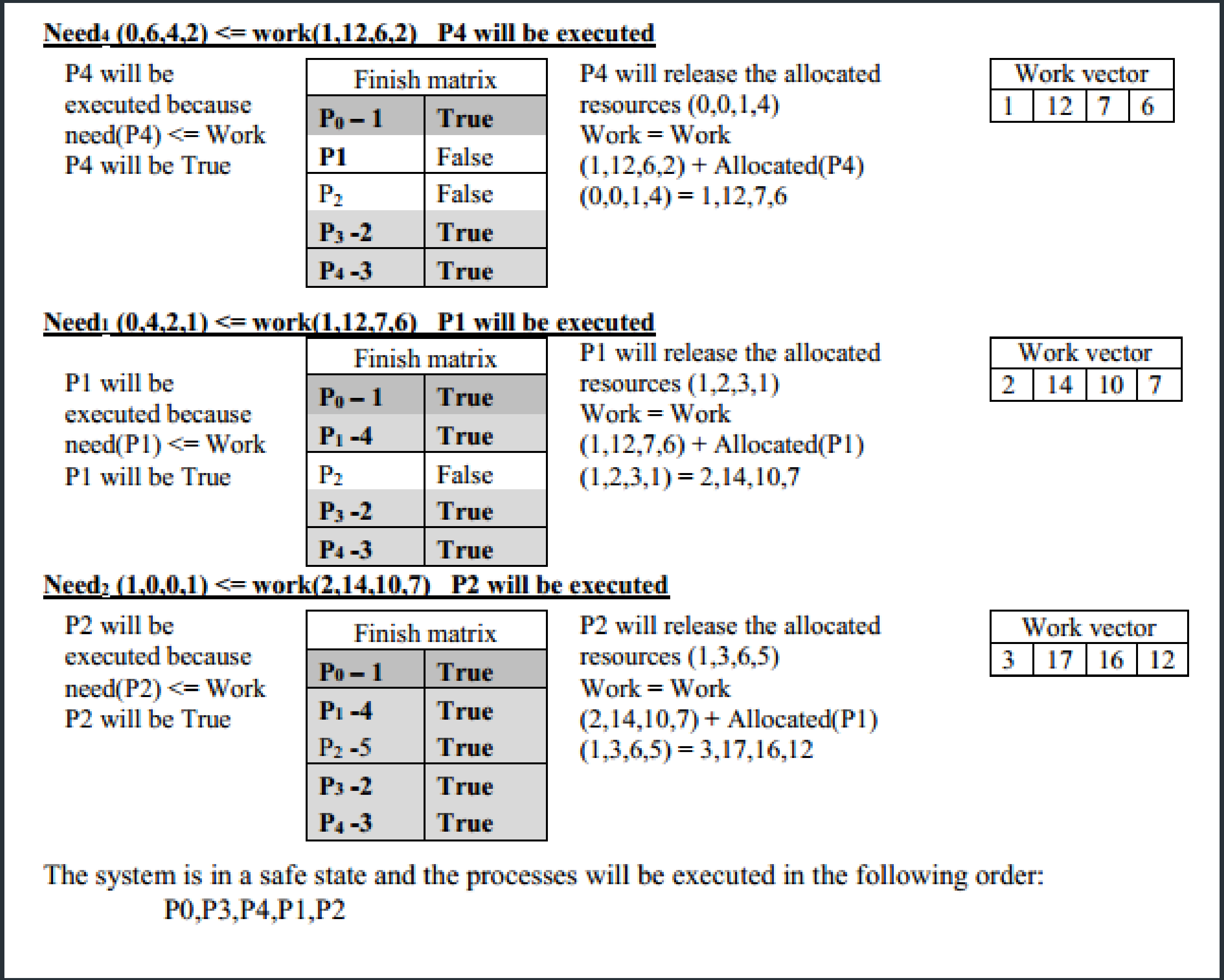
Need$_3$ (0,0,2,0) <= work(1,6,3,0) P3 will be executed

P3 will be executed because need(P3) <= Work P3 will be True

Finish matrix	
P$_0$ – 1	**True**
P1	False
P$_2$	False
P$_3$ -2	**True**
P$_4$	False

P3 will release the allocated resources (0,6,3,2) Work = Work (1,6,3,0)+Allocated(P3) (0,6,3,2) = 1,12,6,2

Work vector			
1	12	6	2



Need$_4$ (0,6,4,2) <= work(1,12,6,2)   P4 will be executed

P4 will be executed because need(P4) <= Work
P4 will be True

Finish matrix	
P$_0$ – 1	**True**
P1	False
P$_2$	False
P$_3$ -2	**True**
P$_4$ -3	**True**

P4 will release the allocated resources (0,0,1,4)
Work = Work
(1,12,6,2) + Allocated(P4)
(0,0,1,4) = 1,12,7,6

Work vector			
1	12	7	6

Need$_1$ (0,4,2,1) <= work(1,12,7,6)   P1 will be executed

P1 will be executed because need(P1) <= Work
P1 will be True

Finish matrix	
P$_0$ – 1	**True**
P$_1$ -4	**True**
P$_2$	False
P$_3$ -2	**True**
P$_4$ -3	**True**

P1 will release the allocated resources (1,2,3,1)
Work = Work
(1,12,7,6) + Allocated(P1)
(1,2,3,1) = 2,14,10,7

Work vector			
2	14	10	7

Need$_2$ (1,0,0,1) <= work(2,14,10,7)   P2 will be executed

P2 will be executed because need(P2) <= Work
P2 will be True

Finish matrix	
P$_0$ – 1	**True**
P$_1$ -4	**True**
P$_2$ -5	**True**
P$_3$ -2	**True**
P$_4$ -3	**True**

P2 will release the allocated resources (1,3,6,5)
Work = Work
(2,14,10,7) + Allocated(P1)
(1,3,6,5) = 3,17,16,12

Work vector			
3	17	16	12

The system is in a safe state and the processes will be executed in the following order:
P0,P3,P4,P1,P2

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	

	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

Need is defined to be Max – Allocation

Example (Cont.)

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

  <u>*Need*</u>

	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 $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true.

	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	

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	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?
- Can request for (0,2,0) by P_0 be granted?

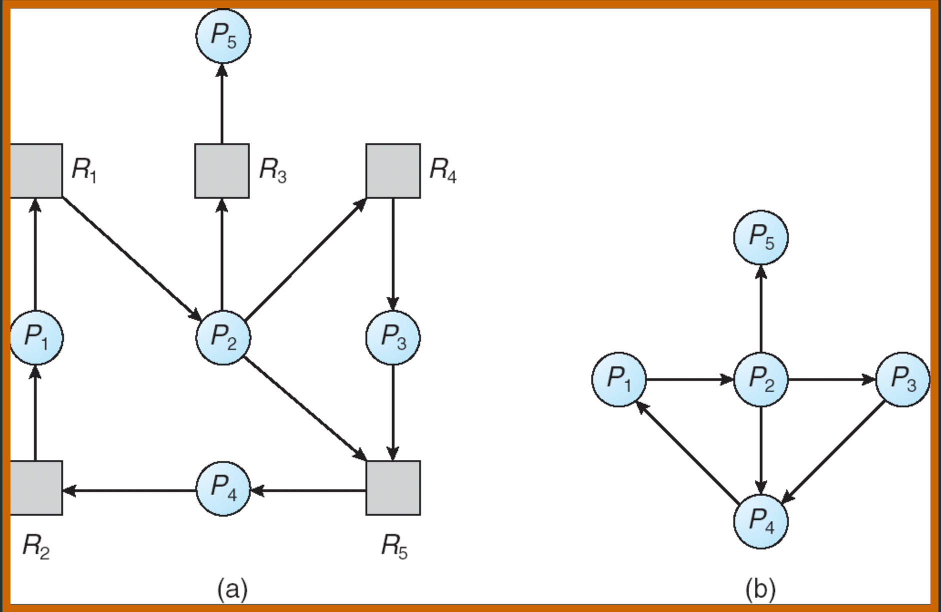
Deadlock Detection

- 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

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 $Allocation_i \neq 0$, then
      Finish[i] = false; otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$
      If no such *i* exists, go to step 4.
3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.

4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, 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.)

- P_2 requests an additional instance of type C.

	Request
	$A\ B\ C$
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?

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?
  - 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?

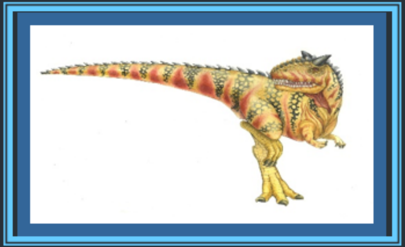
Recovery from Deadlock: Resource Preemption

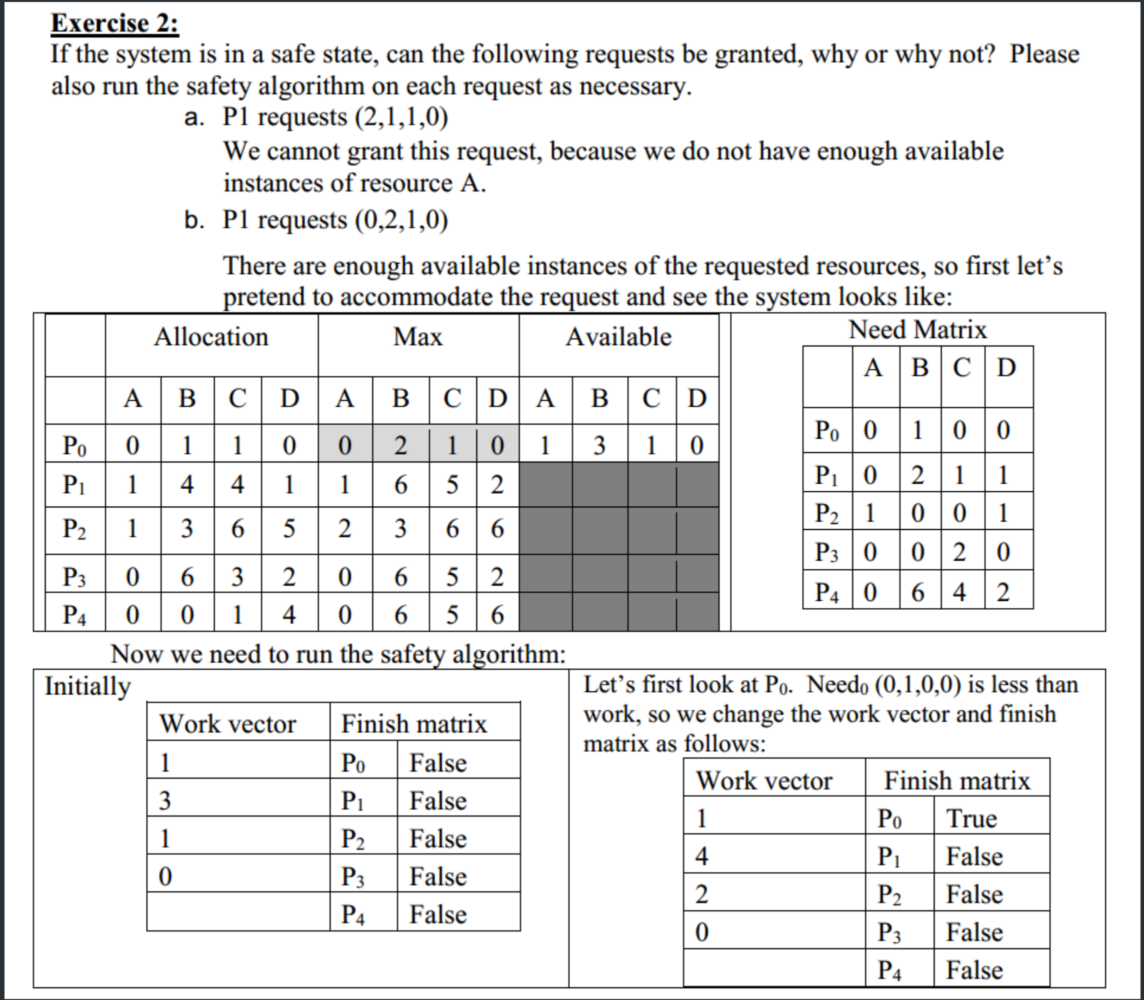
- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

References

- Silberschatz, Gagne, Galvin: Operating System Concepts, 9th Edition

Thank you





Exercise 2:

If the system is in a safe state, can the following requests be granted, why or why not? Please also run the safety algorithm on each request as necessary.

a. P1 requests (2,1,1,0)

   We cannot grant this request, because we do not have enough available instances of resource A.

b. P1 requests (0,2,1,0)

   There are enough available instances of the requested resources, so first let's pretend to accommodate the request and see the system looks like:

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

Need Matrix

	A	B	C	D
P_0	0	1	0	0
P_1	0	2	1	1
P_2	1	0	0	1
P_3	0	0	2	0
P_4	0	6	4	2

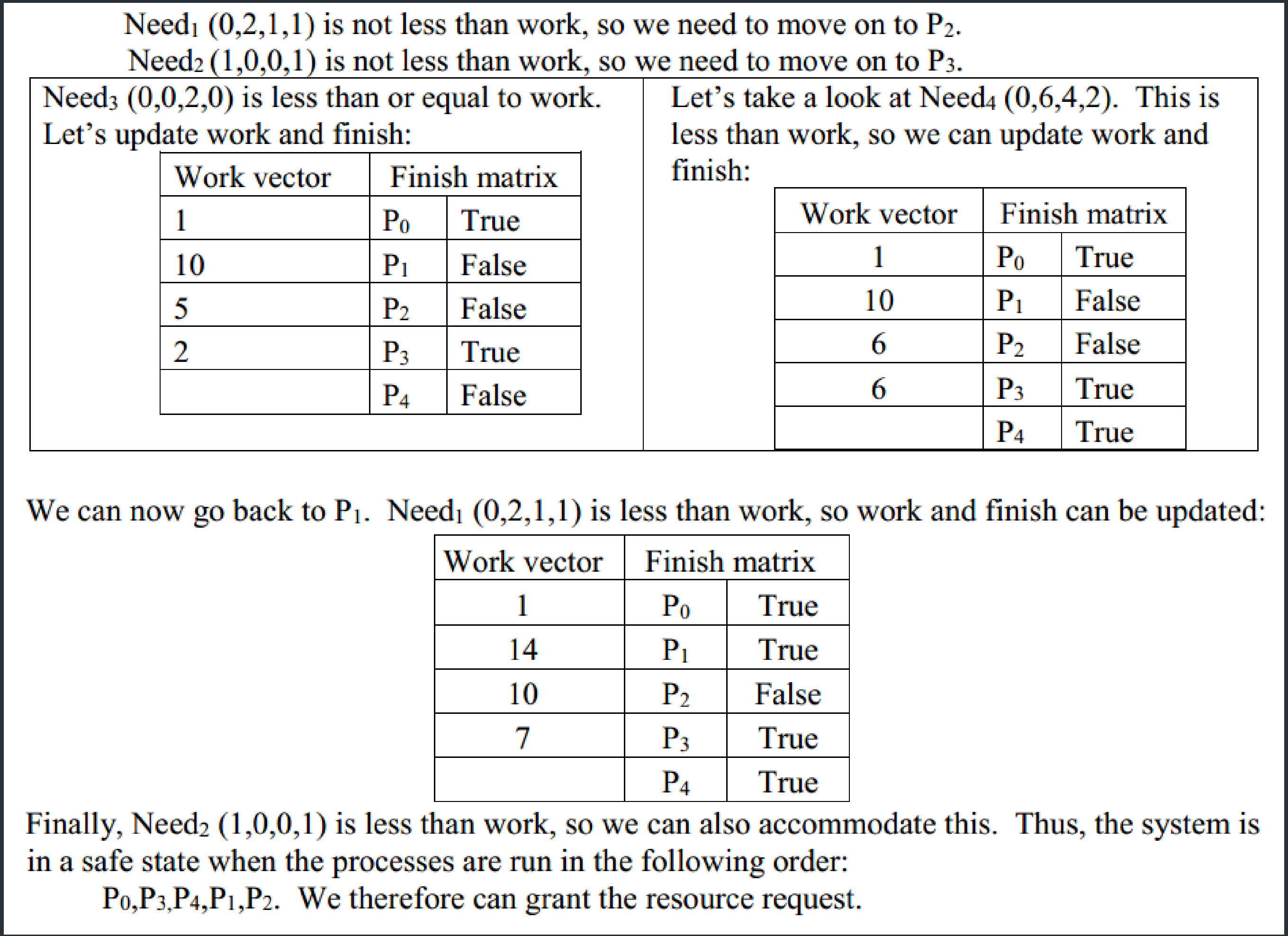
Now we need to run the safety algorithm:

Initially

Work vector	Finish matrix	
1	P_0	False
3	P_1	False
1	P_2	False
0	P_3	False
	P_4	False

Let's first look at P_0. Need_0 (0,1,0,0) is less than work, so we change the work vector and finish matrix as follows:

Work vector	Finish matrix	
1	P_0	True
4	P_1	False
2	P_2	False
0	P_3	False
	P_4	False



Need_1 (0,2,1,1) is not less than work, so we need to move on to P_2.
Need_2 (1,0,0,1) is not less than work, so we need to move on to P_3.

Need_3 (0,0,2,0) is less than or equal to work. Let's update work and finish:

Work vector	Finish matrix	
1	P_0	True
10	P_1	False
5	P_2	False
2	P_3	True
	P_4	False

Let's take a look at Need_4 (0,6,4,2). This is less than work, so we can update work and finish:

Work vector	Finish matrix	
1	P_0	True
10	P_1	False
6	P_2	False
6	P_3	True
	P_4	True

We can now go back to P_1. Need_1 (0,2,1,1) is less than work, so work and finish can be updated:

Work vector	Finish matrix	
1	P_0	True
14	P_1	True
10	P_2	False
7	P_3	True
	P_4	True

Finally, Need_2 (1,0,0,1) is less than work, so we can also accommodate this. Thus, the system is in a safe state when the processes are run in the following order:
P_0,P_3,P_4,P_1,P_2. We therefore can grant the resource request.