OPERATING SYSTEM
IT-41033

CHAPTER – 7

My Profile

Dr. Naing Win Tun
Associate Professor

minnyithattant@gmail.com

minnyithattant

www.facebook.com/minnyithattant (မင်းညီသက်တံ့)

Semester - I

Overview

C-1 Introduction

C-2 System Structures

Process Management

C-3 Process Concept

C-4 Multithreaded Programming

C-5 Process Scheduling

Process Coordination

C- 6 Synchronization

C-7 Deadlocks

7.1 System Models

7.2 Deadlock Characterization

7.3 Methods for Handling Deadlocks

7.4 Deadlock Prevention

7.5 Deadlock Avoidance

7.6 Deadlock Detection

7.7 Recovery from Deadlock

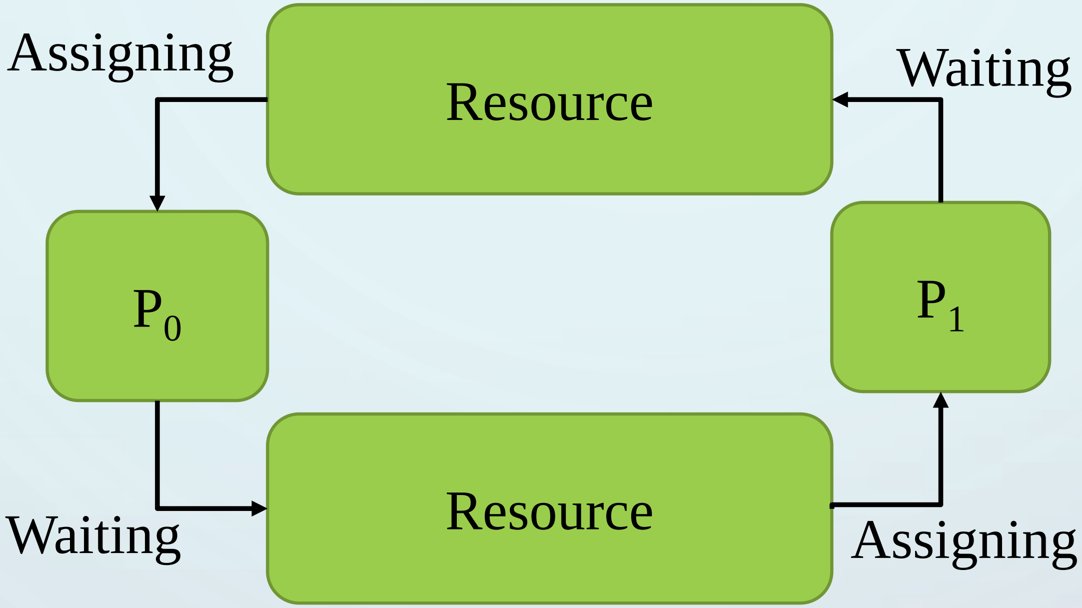
7.8 Summary

CHAPTER (7): DEADLOCKS

- 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.

7.1 SYSTEM MODELS

- A situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. (Deadlock)



- Semaphores *A* and *B*, initialized to 1 P_0 P_1

wait (A);	wait(B);
wait (B);	wait(A);

7.2 DEADLOCK CHARACTERISTICS

- 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

$P_2, \ldots, P_{n-1}$ 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

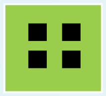
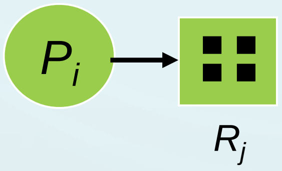
- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system

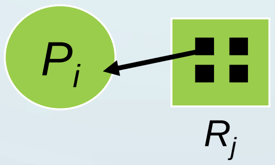
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

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

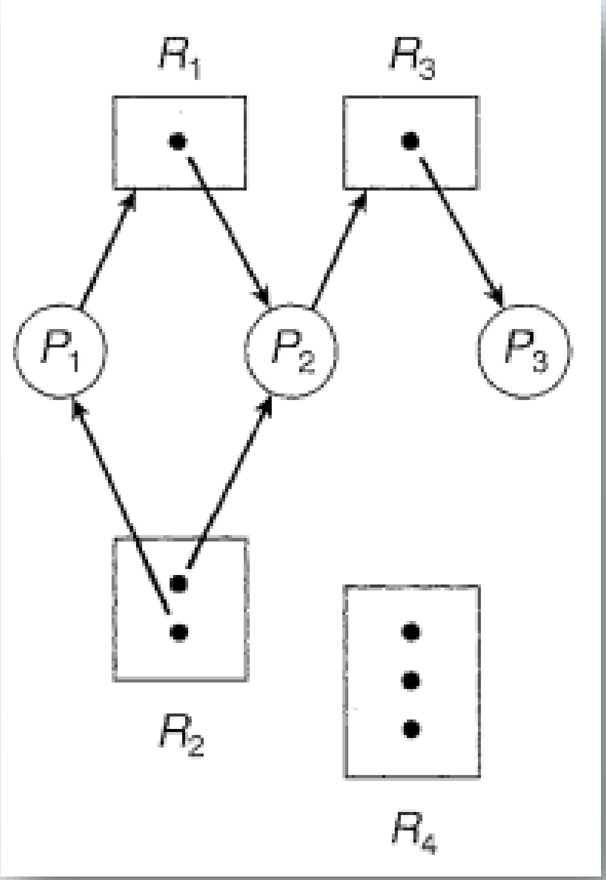
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Symbols)

- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

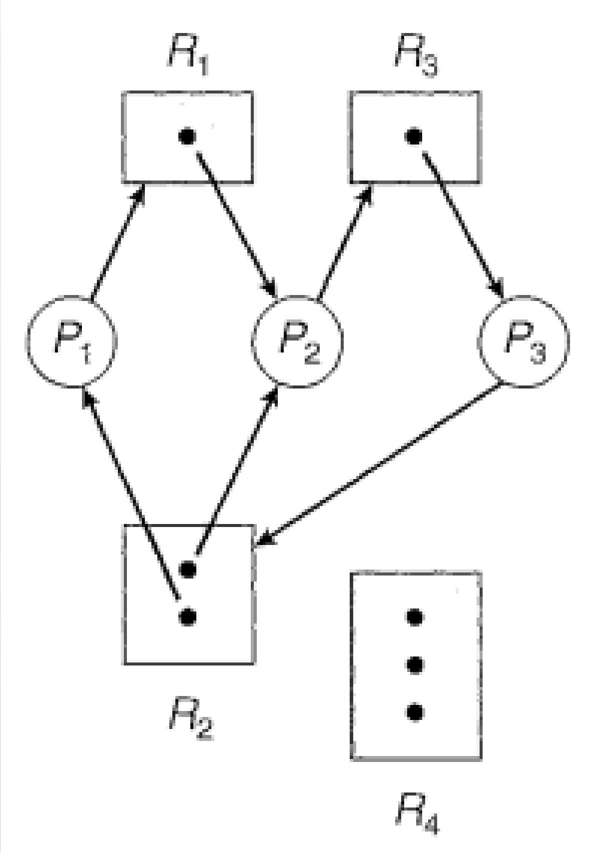
- P_i is holding an instance of R_j 

Resource-Allocation Graph (Cont…)



- The sets P, K and E:
  - P == {P1, P2, P3}
  - R== {R1, R2, R3, R4}
  - E == {P1 -> R1, P2 -> R3, R1 -> P2, R2 -> P2, R2 -> P1, R3 -> P3}
- Resource instances:
  - One instance of resource type R1
  - Two instances of resource type R2
  - One instance of resource type R3
  - Three instance of resource type R4
- Process states:
  - Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1, so on…

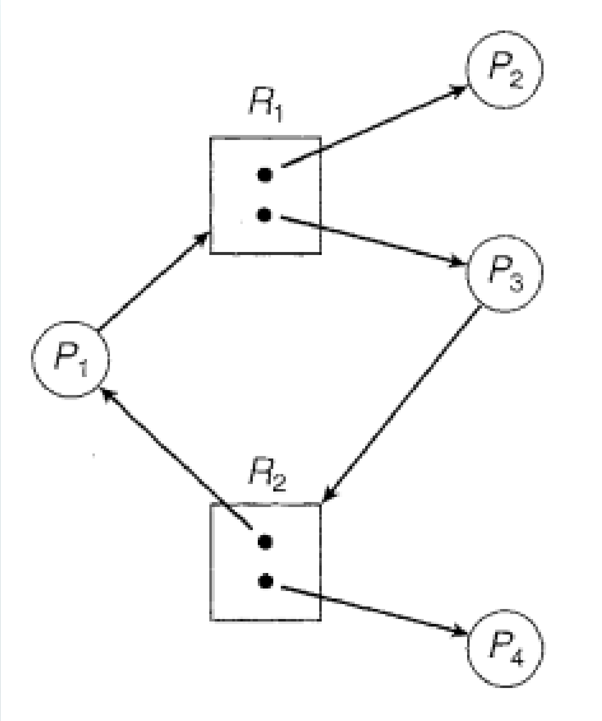
Resource-Allocation Graph with deadlock



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

- P2 is waiting for the resource R3. (held by P3)
- P3 is waiting for P1 or P2 to release resource R2.
- P1 is waiting for P2 to release resource R1.

Resource-Allocation Graph no deadlock (*may be*)



- If a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state.
- If there is a cycle, then the system may or may not be in a deadlocked state.

7.3 METHODS FOR HANDLING DEADLOCKS

Deadlock problem with one of following three states:

- 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

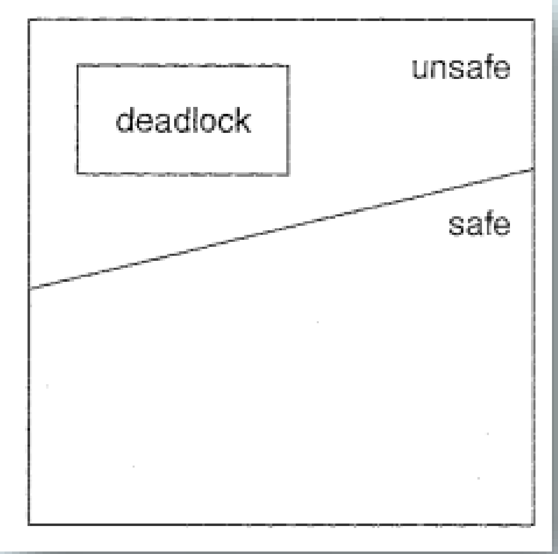
7.4 DEADLOCK PREVENTION

- **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
- **No Preemption** –
  - If a process that is holding some resources and 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.

7.5 DEADLOCK AVOIDANCE

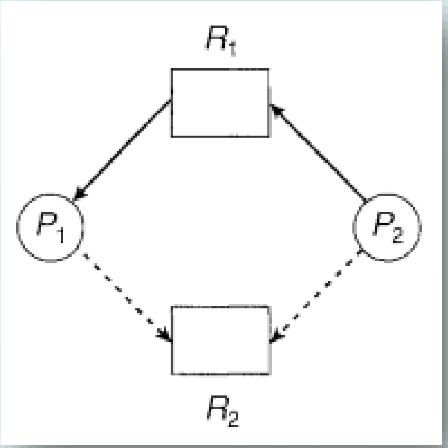
- Alternative method => how resources are to be requested.
  - A tape and a printer => P1 request first tape and then printer before releasing both resources. (P2 will request printer)
- The simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.

Safe State

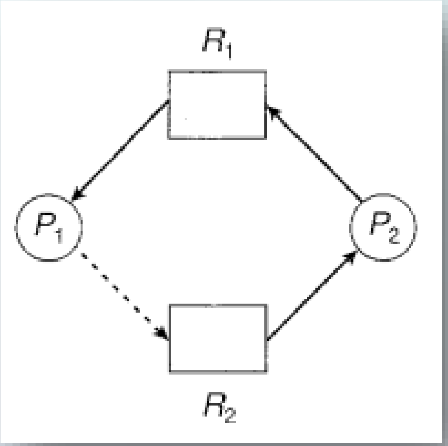


- If a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state.
- If there is a cycle, then the system may or may not be in a deadlocked state.
- To avoid, system remains in a safe state at all times. When a process requests resources, the system checks if granting this request will keep it in a safe state..

Resource-Allocation-Graph Algorithm



Deadlock avoidance



Unsafe state

- **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 when a process requests a resource
- 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, $P_i \rightarrow R_j$
- Resources must be claimed a priori in the system

Banker's Algorithm

n = number of processes, and m = number of resources:

- **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

Note: $Need\ [i,j] = Max[i,j] – Allocation\ [i,j]$

Banker's 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

   Need[i,j] = Max[i,j] – Allocation[i,j]

2. Find an *i* such that both:

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

   (b) $Need_i \leq Work$

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

3. $Work = Work + Allocation_i$

   Finish[*i*] = *true*

   go to step 2

4. If *Finish*[*i*] == true for all *i*, then the system is in a safe state

Banker’s Algorithm (Cont…)

	Allocation				Maximum				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0				
P1	1	0	0	0	1	7	5	0								
P2	1	3	5	4	2	3	5	6								
P3	0	6	3	2	0	6	5	2								
P4	0	0	1	4	0	6	5	6								

Banker's Algorithm (Cont…)

	Allocation				Maximum				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P1	1	0	0	0	1	7	5	0	1	5	3	2	0	7	5	0
P2	1	3	5	4	2	3	5	6	2	8	8	6	1	0	0	2
P3	0	6	3	2	0	6	5	2	2	14	11	8	0	0	2	0
P4	0	0	1	4	0	6	5	6	2	14	12	12	0	6	4	2
									3	14	12	12				

P0, P2, P3, P4, P1

Resource-Request Algorithm

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*

Resource-Request Algorithm

	Allocation				Maximum				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P0	2	0	0	1	4	2	1	2	3	3	2	1				
P1	3	1	2	1	5	2	5	2								
P2	2	1	0	3	2	3	1	6								
P3	1	3	1	2	1	4	2	4								
P4	1	4	3	2	3	6	6	5								

- **P1 request for (1,1,0,0) immediately granted.**

Request-Request Algorithm

	Allocation				Maximum				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P0	2	0	0	1	4	2	1	2	3	3	2	1	2	2	1	1
P1	3	1	2	1	5	2	5	2	5	3	2	2	2	1	3	1
P2	2	1	0	3	2	3	1	6	6	6	3	4	0	2	1	3
P3	1	3	1	2	1	4	2	4	7	10	6	6	0	1	1	2
P4	1	4	3	2	3	6	6	5	10	11	8	7	2	2	3	3
									12	12	8	**10**				

- **P1 request for (1,1,0,0) immediately granted.**
- **P4 request for (0,0,2,0) immediately granted.**

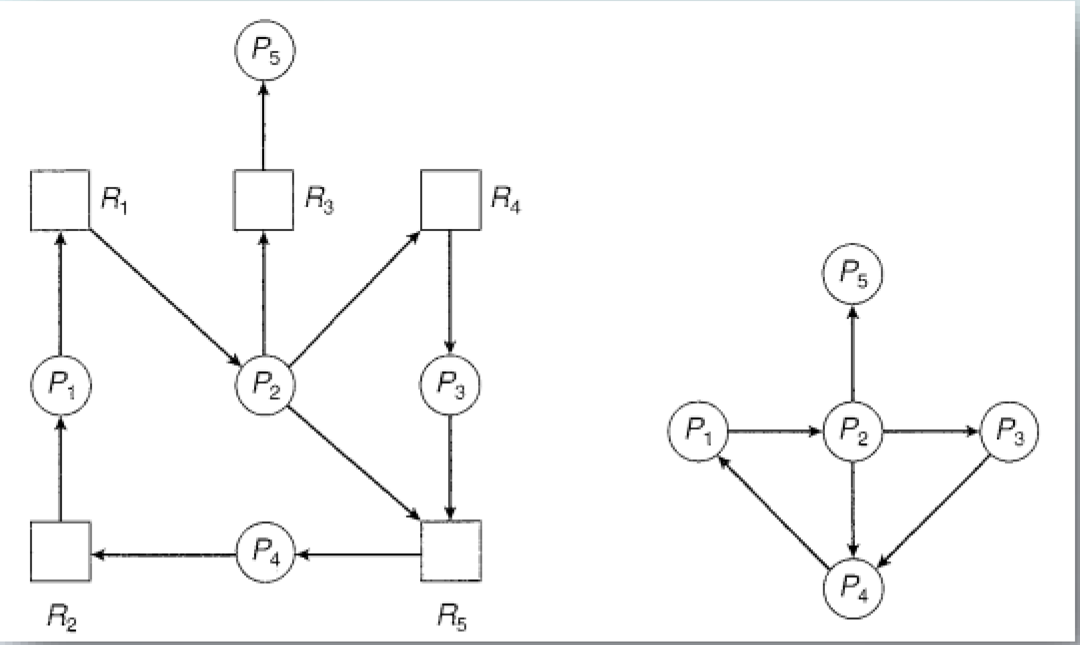
P0, P2, P3, P4, P1

7.6 DEADLOCK DETECTION

Single Instance
(wait for graph)

Multiple Instance
(Banker)

Resource Allocation Graph and Wait for Graph



Deadlock Detection and Recovery

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2	0	1	0
P2	3	0	3	0	0	0	3	1	3
P3	2	1	1	1	0	0	5	2	4
P4	0	0	2	0	0	2	7	2	4

7 2 6

- **P2 request (0,0,1)**

P0, P3, P4, P1, P2

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

References

Images: Internet

Source: Operating System Concepts (8th Edition)