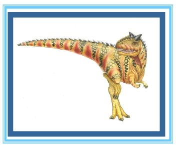
Chapter 6: Deadlocks



1

Chapter 6: Deadlocks

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

2

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

3

The Deadlock Problem

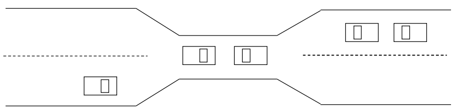
- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs another one
- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

acquire: อยากครอบครอง

4

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
- Note – Most OSes do not prevent or deal with deadlocks

5

System Model

- 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**

6

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.

simultaneously : เกิดขึ้นในเวลาเดียวกัน
voluntarily : กระทำเสร็จแล้ว

Resource-Allocation Graph

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

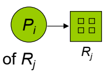
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge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

Resource-Allocation Graph (Cont.)

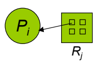
- Process

- Resource Type with 4 instances

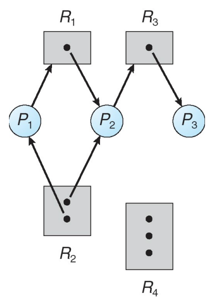
- P_i requests instance of R_j



- P_i is holding an instance of R_j

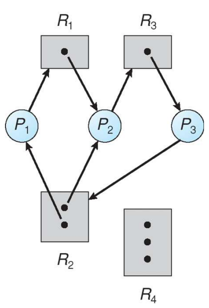


Example of a Resource Allocation Graph



ได้กราฟ E = {P1->R1,
P2->R3,
R1->P2,
R2->P1,
R2-> P2,
R3->P3}

Resource Allocation Graph With A Deadlock



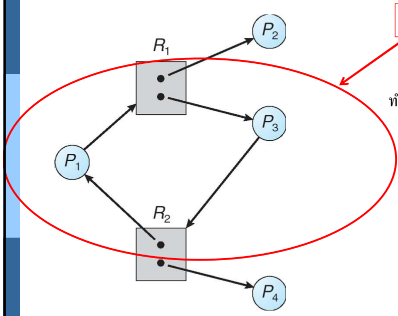
ได้กราฟ **E** คือ ?

={ P1->R1, P2->R3, P3->R2,
R1->P2, R2->P1, R2->P2,
R3->P3 }

กราฟนี้มีการเกิดเป็น Cycle (วงรอบ) 2 วง ได้แก่
วงที่ 1
P1->R1->P2->R3->P3->R2->P1

วงที่ 2 = P2->R3->P3->R2->P2

Graph With A Cycle But No Deadlock



A Cycle

ทำไมถึงไม่เกิด Deadlock ???

หาก P4 ปล่อย R2 ให้กับระบบ
P3 จะได้ครอบครองทรัพยากร
และไม่เกิด Cycle

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

13

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state
  (กำหนดเงื่อนไขในการใช้ **Resource)**
- Allow the system to enter a deadlock state and then recover
  (เมื่อเกิดปัญหา ตามแก้ไขทีหลัง)
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

  (มองข้ามปัญหา ทำเสมือนไม่มีการเกิด **Deadlock**ในระบบ
  **** วิธีนี้เป็น 1 วิธีการที่ใช้ใน **OS** ส่วนใหญ่******)

14

Deadlock Prevention (ป้องกัน)

พิจารณาถึง การเกิด Deadlock ต้องมีเงื่อนไข ทั้ง 4 กรณีเกิดขึ้นพร้อมกัน

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable 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

  (หาก Resource ว่างแต่ไม่สามารถให้ Process ถือครองได้เวลานานๆ สามารถนำ Resource มาใช้ประโยชน์เมื่อใช้เสร็จต้องรีบคืน เมื่อจะใช้ใหม่ต้อง Request ใหม่ หากมี Process ต้องการใช้ Resource ที่ได้รับความนิยมมากๆ จะเกิด Starvation)

15

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

Impose : กำหนด

16

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.

priori : บางส่วนก่อนหน้า

17

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 safe sequence of all processes.
- Sequence $<P_1, P_2, \ldots, P_n>$ is safe if 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 $j < i$.
  - 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.

18

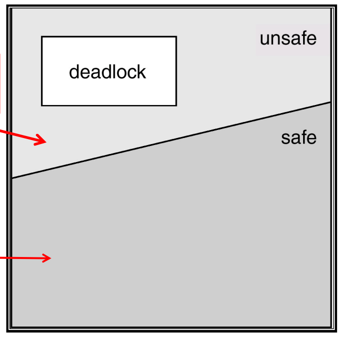
Basic Facts

- 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

อยู่ในสถานะ **unsafe** อาจจะไม่เกิด **Deadlock** ได้

อยู่ในสถานะ **safe** ไม่เกิด **Deadlock** แน่นอน



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

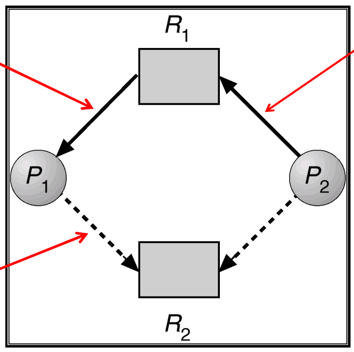
Claim edge : เส้นความต้องการ (แสดงโดยใช้เส้นประ)

Resource-Allocation Graph For Deadlock Avoidance

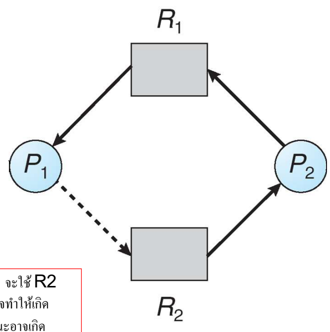
Assignment edge

Request edge

Claim edge



Unsafe State In Resource-Allocation Graph



หากมีความต้องการของ P1 จะใช้ R2 ระบบจะไม่อนุญาต เพราะอาจทำให้เกิด Deadlock (ดูจากลักษณะอาจเกิดวงรอบ ขึ้นได้)

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

25

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.

26

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. (เก็บ Resourceที่ว่าง)
- *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. (เก็บจน. สูงสุดของ Resource ที่กระบวนการแต่ละตัวต้องการใช้)
- *Allocation: n* x *m* matrix. If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of R_j. (เก็บจน. Resource ที่แต่ละกระบวนการครอบครองอยู่)
- *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. (เก็บจน. Resourceที่เหลืออยู่ ที่แต่ละกระบวนการยังคงต้องการใช้ เพื่อทำงานให้เสร็จสมบูรณ์)

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

(ที่ขอใช้ Resource สูงสุด – ที่ได้ Resource ครอบครองแล้ว)

27

Safety Algorithm

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

   Work = *Available*

   Finish [*i*] = *false* for *i* = 1,2, …, *n*.

2. Find and *i* such that both:

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

   (b) $Need_i \le Work$

   ตรวจสอบเงื่อนไข **If (Finish[i]==false AND $Need_i$ <= 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 มีสถานะที่ avilable เพื่อรอในการทำงานครั้งต่อไป

28

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 $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \le Available$, 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:

   $Available = Available - Request_i$;

   $Allocation_i = Allocation_i + Request_i$;

   $Need_i = Need_i – Request_i$;

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

exceed: มากเกิน
pretend: การอ้างถึง

29

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	

30

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

7 5 3 – 0 1 0

4 3 3 – 0 0 2

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

31

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	

2 0 0 + 1 0 2

3 3 2 – 1 0 2

122 - 102

- 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?

32

Practice: Banker's Algorithm

- 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	

- จาก Snapshot T_0 จงแสดงวิธีทำเพื่อที่จะตรวจสอบว่า ระบบจะอยู่ในสถานะที่ปลอดภัย (safe state) หรือไม่ ?? เมื่อมีลำดับของโพรเซส คือ P_3, P_2, P_1, P_4, P_0

33

Deadlock Detection

หากไม่มีการ Protection และ Avoidance

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

34

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.

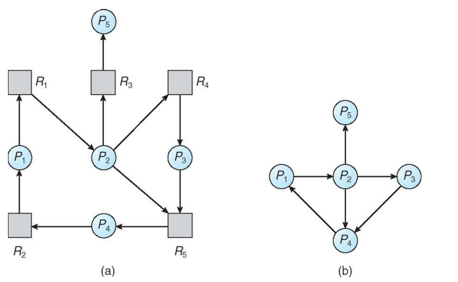
แปลงเป็น กราฟการรอคอยทรัพยากร (wait-for graph)

จะไม่มีโนดของ Resource แล้ว

- Periodically invoke an algorithm that searches for a **cycle in the graph**.
- 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.

35

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

36

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.

37

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*.

   ของ Banker's Algo
   For i=1,2 ..n then
   Finish[i]=false;

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

   ตรวจสอบเงื่อนไข
   If (Finish[i]==false AND $Need_i$ <= work)

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

38

Detection Algorithm (Cont.)

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($m \times n^2$) operations to detect whether the system is in deadlocked state

39

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*

40

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?
  - 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

insufficient : ไม่เพียงพอ

41

Practice: Deadlock Detection

- 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 1 0	
P_3	2 1 1	1 2 3	
P_4	0 0 2	0 0 2	

- จาก Snapshot T_0 เมื่อ P_3 ต้องการ (Request) ทรัพยากรชนิดต่างๆ (Resource type) A = 1, B=2, และ C =3 เมื่อลำดับการทำงานของโพรเซส คือ P_0, P_2,P_3, P_1 และ P_4 ให้นักศึกษาแสดงวิธีทำในการตรวจสอบสถานะของระบบว่าจะมีสถานะเป็นอย่างไร ??

42

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

arbitrarily : ไม่มีกฎเกณฑ์

Recovery from Deadlock: Process Termination

(ยกเลิกกระบวนการที่เกิด deadlock)

- 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

การเลือกใช้ จะต้องพิจารณาผลที่จะเกิดขึ้น 3 ข้อ ดังนี้

- 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

Victim : ผู้รับเคราะห์ (process)

End of Chapter 6