MODERN OPERATING SYSTEMS

ANDREW S. TANENBAUM

Chapter 6
Deadlocks

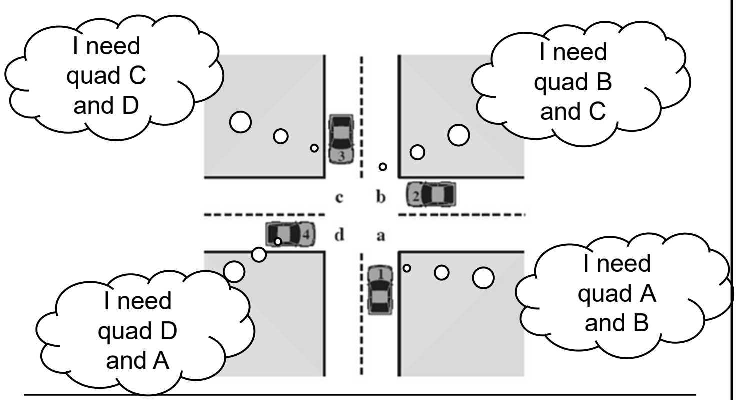
1

Deadlock

- The permanent (lâu dài/thường xuyên) blocking of a set of processes that either compete (canh tranh) for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting (chờ đợi) an event that can only be triggered by another blocked process in the set
- Permanent (Thường xuyên)
- No efficient solution

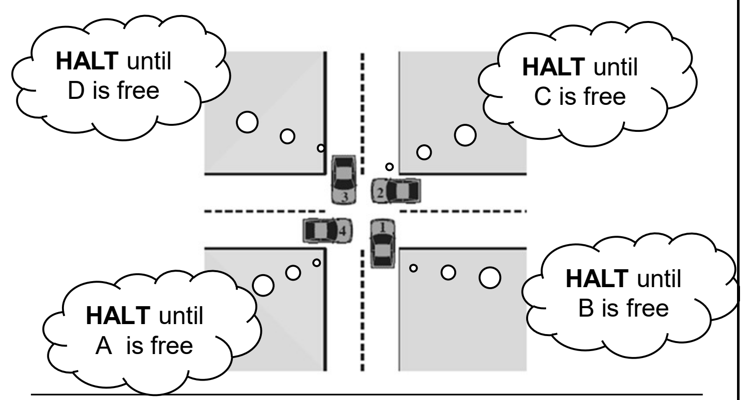
2

Potential Deadlock



3

Actual Deadlock



4

Conditions for Deadlock

Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
• only one process may use a resource at a time	• a process may hold allocated resources while awaiting assignment of others	• no resource can be forcibly (ép buộc) removed from a process holding it	• a closed chain (chuỗi/dây) of processes exists, such that each process holds at least one resource needed by the next process in the chain

5

Dealing (xử sự) with Deadlock

- Three general approaches exist for dealing with deadlock:

Prevent Deadlock
- Adopt (áp dụng/chấp nhận) a policy that eliminates (loại bỏ) one of the conditions

Avoid Deadlock
- make the appropriate (phù hợp) dynamic choices based on the current state of resource allocation

Detect Deadlock
- attempt to detect the presence (sự hiện diện) of deadlock and take (thực hiện) action to recover

6

Resource Allocation Graph

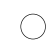
A set of vertices V and a set of edges E

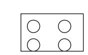
V is partitioned into 2 types
- P = {P1, P2,…,Pn} - the set of processes in the system
- R = {R1, R2,…,Rn} - the set of resource types in the system

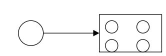
Two kinds of edges
- Request edge - Directed edge Pi ---> Rj
- Assignment edge - Directed edge Rj ----> Pi

7

Resource Allocation Graph

Process 

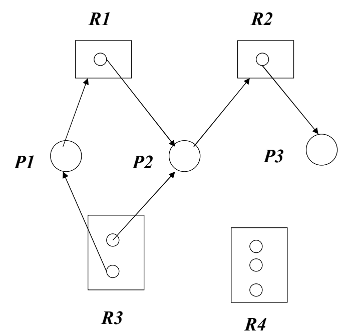
Resource type with 4 instances 

Pi requests instance of Rj 

Pi is holding an instance of Rj 

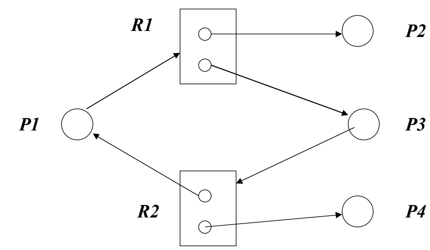
8

Graph with no cycles



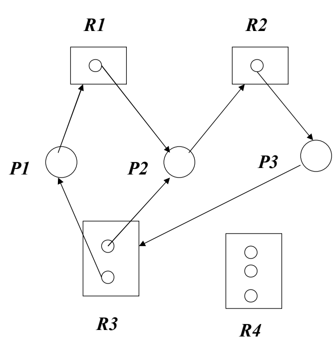
9

Graph with cycles



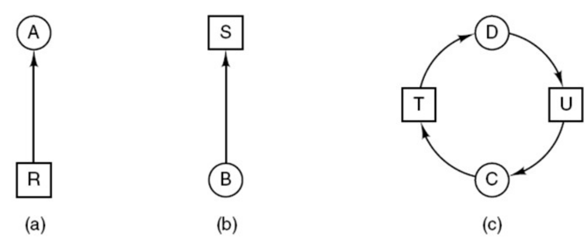
10

Graph with cycles and deadlock



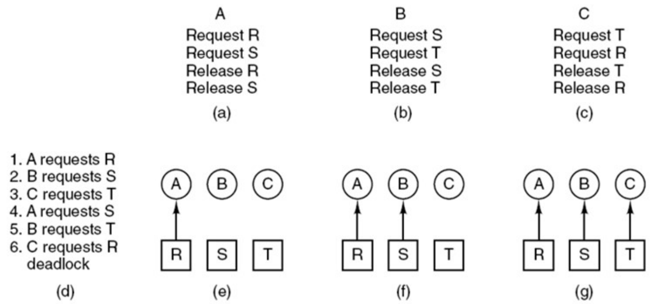
11

Deadlock Modeling (1)



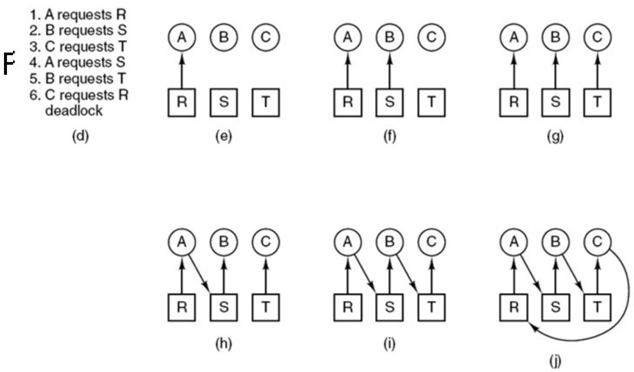
12

Deadlock Modeling (2)



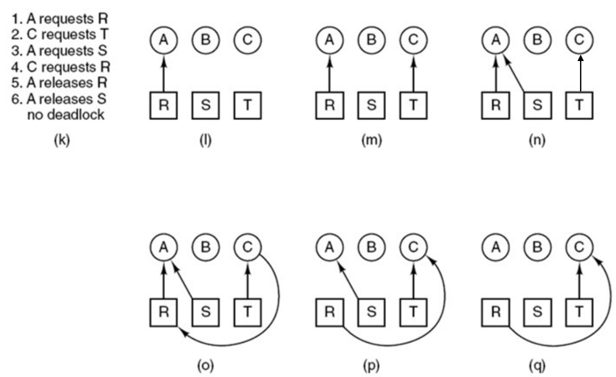
13

Deadlock Modeling (3)



14

Deadlock Modeling (4)



15

Deadlock Modeling (5)

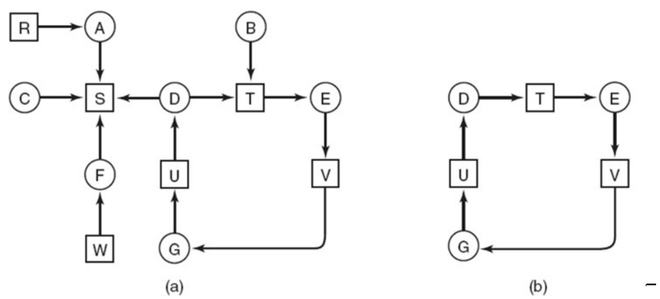
Strategies for dealing with deadlocks:

1. Just ignore (phớt lờ) the problem.
2. Detection and recovery. Let deadlocks occur, detect them, take action.
3. Dynamic avoidance by careful resource allocation.
4. Prevention, by structurally negating one of the four required conditions.

16

Deadlock Detection with One Resource of Each Type (1)

Figure 6-5. (a) A resource graph. (b) A cycle extracted from (a).



17

Deadlock Detection with One Resource of Each Type (2)

Algorithm for detecting deadlock:

1. For each node, N in the graph, perform the following five steps with N as the starting node.
2. Initialize L to the empty list, designate all arcs as unmarked.
3. Add current node to end of L, check to see if node now appears in L two times. If it does, graph contains a cycle (listed in L), algorithm terminates.

…

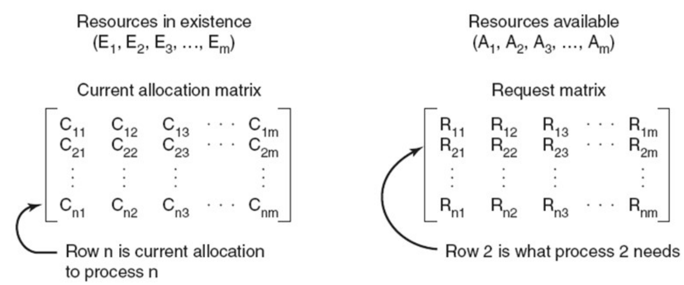
18

Deadlock Detection with One Resource of Each Type (3)

4. From given node, see if any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
6. If this is initial node, graph does not contain any cycles, algorithm terminates. Otherwise, dead end. Remove it, go back to previous node, make that one current node, go to step 3.

19

Deadlock Detection with Multiple Resources of Each Type (1)



20

Deadlock Detection with Multiple Resources of Each Type (2)

Deadlock detection algorithm:

- Data Structures
  - *Available*: Vector of length *m*. If *Available*[*j*] = *k*, there are *k* instances of resource type *Rj* available.
  - *Allocation*: *n* × *m* matrix. If *Allocation*[*i*,*j*] = *k*, then process *Pi* is currently allocated *k* instances of resource type *Rj*.
  - *Request* : An *n* × *m* matrix indicates the current request of each process. If *Request* [*i*,*j*] = *k*, then process *Pi* is requesting *k* more instances of resource type *Rj* .

21

Deadlock Detection with Multiple Resources of Each Type (3)

Deadlock detection algorithm:

- Step 1: Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize
  - *Work* := *Available*
  - For *i* = 1,2,…,*n*, if *Allocation*(*i*) ≠ 0, then *Finish*[*i*] := *false*, otherwise *Finish*[*i*] := *true*.
- Step 2: Find an index *i* such that both:
  - *Finish*[*i*] = *false*
  - *Request_i* ≤ *Work*
  - If no such *i* exists, go to step 4.

22

Deadlock Detection with Multiple Resources of Each Type (4)

Deadlock detection algorithm:

- Step 3: *Work* := *Work* + Allocation_i
  - *Finish*[*i*] := *true*
  - go to step 2
- Step 4: If *Finish*[*i*] = *false* for some *i*, 1 ≤ *i* ≤ *n*, then the system is in a deadlock state. Moreover, if *Finish*[*i*] = *false*, then *Pi* is deadlocked.

Algorithm requires an order of m × (n^2) operations to detect whether the system is in a deadlocked state.

23

Deadlock Detection with Multiple Resources of Each Type (5)

Figure 6-7. An example for the deadlock detection algorithm.

$$\begin{array}{lcccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ E = (& 4 & 2 & 3 & 1\ ) \end{array} \qquad \begin{array}{lcccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ A = (& 2 & 1 & 0 & 0\ ) \end{array}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

24

Detection-Algorithm Use

- 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
- How often --
  - Every time a request for allocation cannot be granted immediately
    - Allows us to detect set of deadlocked processes and process that “caused” deadlock. Extra overhead.
    - Every hour or whenever CPU utilization drops.
  - With arbitrary invocation there may be many cycles in the resource graph and we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

25

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 the 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?

26

Recovery from Deadlock: Resource Preemption

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

27

Deadlock Avoidance

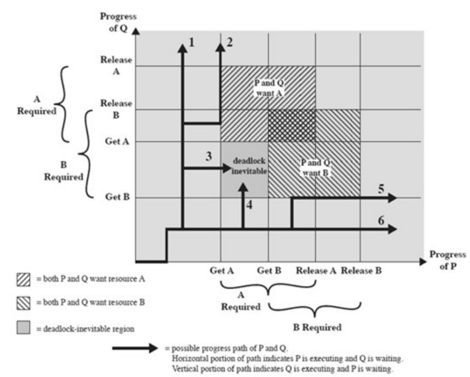


Figure 6.2 Example of Deadlock

28

Deadlock Avoidance

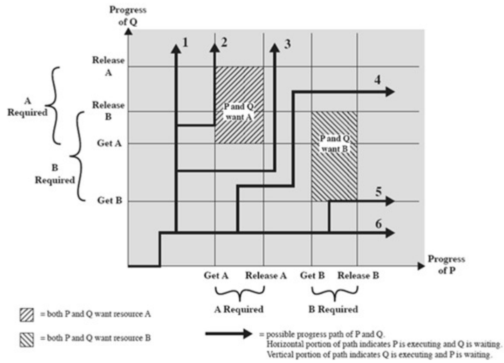


Figure 6.3 Example of No Deadlock [BACO03]

29

Deadlock Avoidance

- Set of resources, set of customers, banker
- Rules
  - Each customer tells banker maximum number of resources it needs.
  - Customer borrows resources from banker.
  - Customer returns resources to banker.
  - Customer eventually pays back loan.
- Banker only lends resources if the system will be in a *safe state* after the loan.

30

Deadlock Avoidance

- Requires that the system has some additional apriori 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.

31

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 <P1, P2, …Pn> is safe, if for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by Pj with j<i.
  - If Pi resource needs are not available, Pi can wait until all Pj have finished.
  - When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate.
  - When Pi terminates, Pi+1 can obtain its needed resources...

32

Basic Facts

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

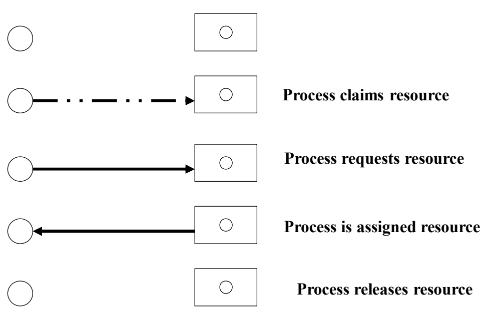
33

Resource Allocation Graph Algorithm

- Used for deadlock avoidance when there is only one instance of each resource type.
  - Claim edge: Pi → Rj indicates that process Pi may request resource Rj; represented by a dashed line.
  - Claim edge converts to request edge when a process requests a resource.
  - When a resource is released by a process, assignment edge reconverts to claim edge.
  - Resources must be claimed a priori in the system.
- If request assignment does not result in the formation of a cycle in the resource allocation graph - safe state, else unsafe state.

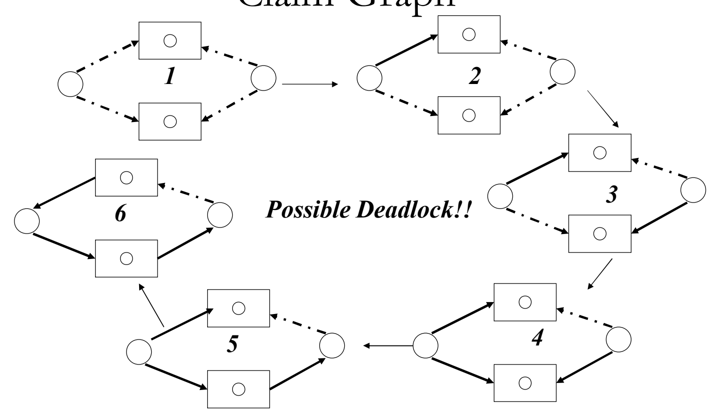
34

Claim Graph



35

Claim Graph



36

Banker's Algorithm

- Used for multiple instances of each resource type.
- Each process must a priori claim maximum use of each resource type.
- 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.

37

Data Structures for the Banker's Algorithm

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

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

38

Safety Algorithm

- Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize
  - *Work* := *Available*
  - *Finish*[i] := *false* for i = 1,2,…,n.
- Find an i (i.e. process Pi) such that both:
  - *Finish*[i] = *false*
  - *Need_i* <= *Work*
  - If no such i exists, go to step 4.
- *Work* := *Work* + Allocation_*i*
  - *Finish*[i] := *true*
  - go to step 2
- If *Finish*[i] = *true* for all i, then the system is in a safe state.

39

Resource-Request Algorithm for Process *Pi*

- Request_i = request vector for process Pi. If Request_i[j] = k, then process Pi wants k instances of resource type Rj.
  - STEP 1: If *Request_i* ≤ *Need_*i, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
  - STEP 2: If *Request_*i ≤ *Available*, go to step 3. Otherwise, Pi must wait since resources are not available.
  - STEP 3: Pretend to allocate requested resources to Pi by modifying the state as follows:
    - *Available* := *Available* – *Request_i;*
    - *Allocation_i* := *Allocation_i* + *Request_i;*
    - *Need_i* := *Need_*i – *Request_i;*
  - If safe ⇒ resources are allocated to *Pi*.
  - If unsafe ⇒ *Pi* must wait and the old resource-allocation state is restored.

40

Safe and Unsafe States (1)

Figure 6-9. Demonstration that the state in (a) is safe.

(a)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(b)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

(c)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

(d)

A	3	9
B	0	–
C	7	7

Free: 0

(e)

A	3	9
B	0	–
C	0	–

Free: 7

41

Safe and Unsafe States (2)

Figure 6-10. Demonstration that the state in (b) is not safe.

(a)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(b)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(c)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(d)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

42

The Banker's Algorithm for a Single Resource

Figure 6-11. Three resource allocation states:
(a) Safe. (b) Safe. (c) Unsafe.

(a)

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(b)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(c)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

43

The Banker's Algorithm for Multiple Resources (1)

Figure 6-12. The banker's algorithm with multiple resources.

Resources assigned

Process	Tape drives	Plotters	Printers	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources still needed

Process	Tape drives	Plotters	Printers	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

E = (6342)
P = (5322)
A = (1020)

44

Combined approach to deadlock handling

- Combine the three basic approaches
  - Prevention
  - Avoidance
  - Detection

  allowing the use of the optimal approach for each class of resources in the system.
- Partition resources into hierarchically ordered classes.
  - Use most appropriate technique for handling deadlocks within each class.

45

Summary

- Deadlock:
  - the blocking of a set of processes that either compete for system resources or communicate with each other
  - Blockage (tắc nghẽn) is permanent (thường xuyên) unless OS takes action
  - may involve (liên quan đến) reusable or consumable resources
    - Consumable = destroyed when acquired by a process
    - Reusable = not depleted/destroyed by use
- Dealing with deadlock:
  - prevention – guarantees (đảm bảo) that deadlock will not occur
  - detection – OS checks for deadlock and takes action
  - avoidance – analyzes each new resource request

46