COSC4740-01 Operating Systems Design, Fall 2004, Byunggu Yu

Chapter 7 Deadlock

1. 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 #1: System has 2 tape drives. P_1 and P_2 each hold one tape drive and each needs another one.
- Example #2: semaphores *A* and *B*, initialized to 1

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

2. Assumptions

- Resource types $R_1, R_2, \ldots, R_m$ (e.g., *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

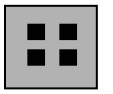
3. Deadlock Situation

Deadlock can arise if the following four conditions hold simultaneously:

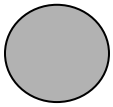
- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a set $\{P_1, P_2, \ldots, P_n\}$ of waiting processes such that P_1 is waiting for a resource that is held by P_2, P_2 is waiting for a resource that is held by P_3, …, 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_1.

4. Resource Allocation Graph (RAG)

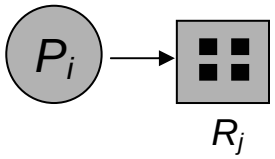
- RAG: a set of vertices (V) and edges (E)
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$ (i.e., processes); $R = \{R_1, R_2, \ldots, R_m\}$ (resource types).
- request edge: directed edge $P_1 \rightarrow R_j$
- assignment edge: directed edge $R_j \rightarrow P_i$



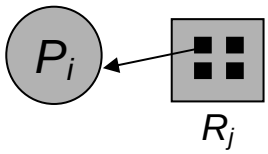
R vertex (a resource type with 4 instances)



P vertex (a process)



Request edge: P_i requests instance of R_j



Assignment edge: P_i is holding an instance of R_j

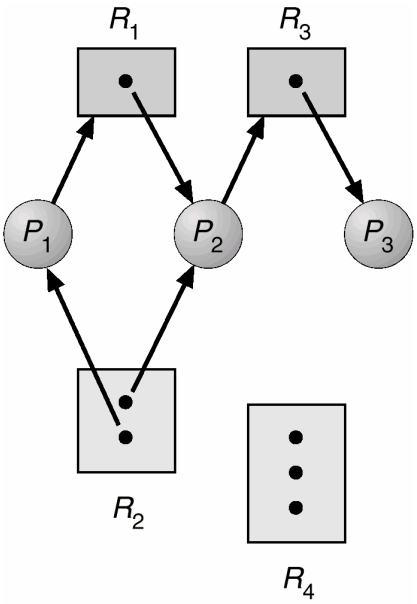
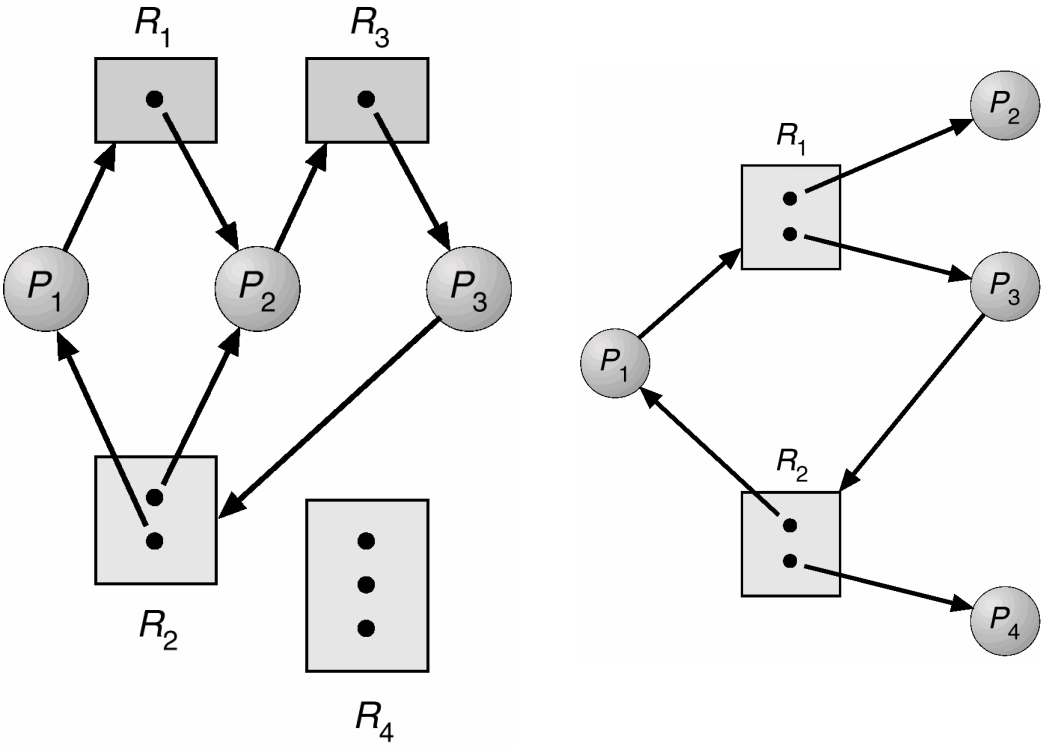


Figure 7.1 Example of RAG

5. RAG and Deadlock



(a) (b)

Figure 7.2 (a) RAG with deadlock and (b) RAG with a cycle but no deadlock

- 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.

6. Method for Handling Deadlocks

Choice #1: Ensure that the system will never enter a deadlock state

a. Deadlock Prevention
b. Deadlock Avoidance

Choice #2: Allow the system to enter a deadlock state and then recover

c. Deadlock Detection & Recovery

Choice #3: Ignore and don't care

6.1 Deadlock Prevention: ensure that the system will *never* enter a deadlock state.

Technique #1: Ensure that "Mutual Exclusion" condition never holds

- Impossible

Technique #2: Ensure that "Hold-and-Wait" condition never holds

- Protocol #1: A new process requests every required resources before it begins execution, OR
- Protocol #2: OS allows a process to request resources only when it has no resource
- Problems: low resource utilization, starvation

Technique #3: Ensure that "No Preemption" condition never holds

- Protocol #1: Pi requests a resource at point in time. If the resource is not available then all resources currently being held by Pi are preempted, OR
- Protocol #2: Pi request a resource at anytime. If the resource is held by Pj waiting for another resource then preempt the resource and allocate it to Pi.
- Problem: for preemptable resources such as CPU registers and memory space. Not applied to such resources as printers and tape drives.

Technique #4: Ensure that "Circular Wait" condition never holds

- Pi can request Rj only after Ri has been allocated to Pi (note, F(Rj) > F(Ri))
- Pi release Ri before it requests Rj (note, F(Rj) >= F(Ri)).
- Problem: Resource ordering

6.2 Deadlock Avoidance: Requires that the system has some additional *a priori* information available.

- Requires each process to declare the *maximum number* of resources of each type that it may need.

- Safe State:
  - System is in safe state if there exists a safe sequence of all processes (i.e., the system can allocate all the requested resources in some order).
  - Safe Sequence for the current allocation state: $<P_1, P_2, \ldots, P_n>$, for each P_i, P_i resource request can be satisfied by currently available resources + resources held by all the P_j, with $j<i$.

- Unsafe State: There is no safe sequence

- Deadlock State: the deadlock state is a subset of the unsafe state.

- Deadlock Avoidance ensures that the system will never enter the unsafe state

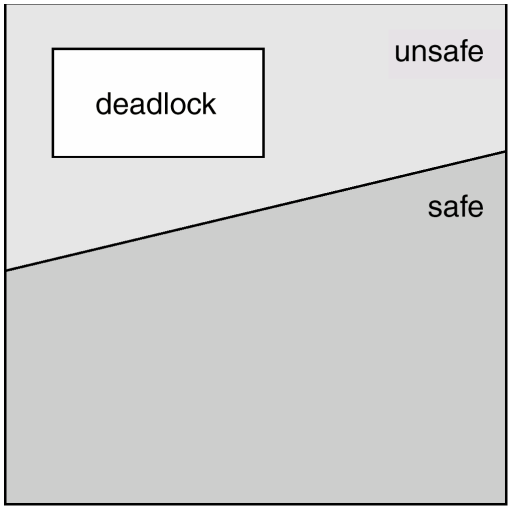
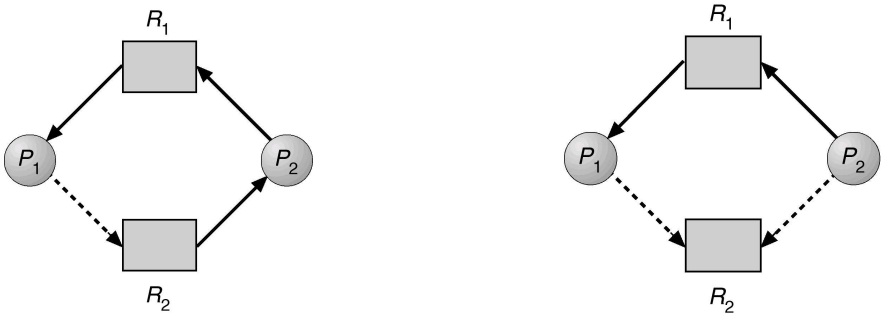


Figure 7.3 Safe/Unsafe/Deadlock State Spaces

- When every resource type has only one instance (RAG Algorithm)
  - Before Pi starts executing, all its claim edges are generated and added to the RAG.
  - *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.
    - When a resource is released by a process, assignment edge reconverts to a claim edge.
  - Resource request is granted only if converting the request edge to an assignment edge does not result in the formation of a circle in the RAG.

Unsafe State In A Resource-Allocation Graph **Resource-Allocation Graph For Deadlock Avoidance**



Fugure 7.4 Unsafe Stage in RAG and Deadlock Avoidance

In the single resource instance case, circle in RAG is both a necessary and a sufficient condition.
Note that the 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.

- When resource types have multiple instances (Banker's Algorithm)
  - Before Pi starts executing, it must declare the maximum number of instances of each resource type that it may need..

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

Local Vectors:
Work: a vector of length m.
Finish: a vector of length n

Step 1. Work = Available;
Finish[i] = false, for all i=1,2,...,n;

Step 2. Find i such that (Finish[i]=false && Need[i] <= Work);
If no such i exist then the system is in an unsafe state; return;

Step 3. Work = Work+Allocation[i];
Finish[i] = true;

Step 4. If Finish[i] = true for all i then the system is in a safe state; return;
Else goto Step 2;

Resource-Request Algorithm for Process P_i

Local Vector:

Request: m-dimensional vector. r[j] is the number of instances of j^{th} resource type P_i is currently requesting.

Step 1. If Request <= Need[i] then goto Step 2;
Else, raise an error; return;

Step 2. If Request<= Available[i] then goto Step 3;
Else, block the current process P_i; return;

Step 3. Backup the current values of Available, Need[i], and Allocation[i];
Available = Available-Request;
Allocation[i] = Allocation[i]+Request;
Need[i] = Need[i]-Request;

Step 4. Call the Safety Algorithm;
If it is safe then physically allocate requested resources; return;
Else restore Available, Allocation[i], and Need[i] and block the process P_i; return;

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
	ABC	*ABC*	*ABC*
P_0	010	753	332
P_1	200	322	
P_2	302	902	
P_3	211	222	
P_4	002	433	

- 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 there are sequences (e.g., $<P_1\ P_3\ P_4\ P_0\ P_2>$).

P1 requests (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ *true*.

	Allocation	Need	Available
	ABC	*ABC*	*ABC*
P_0	010	743	230
P_1	302	020	
P_2	301	600	
P_3	211	011	
P_4	002	431	

- 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?

6.3 Deadlock Detection and Recovery: In this method, OS allows the system to enter deadlock state. Then OS detects the deadlock with a detection algorithm, and runs a recovery scheme.

6.3.1 Deadlock Detection Algorithms

- Single instance case
  - 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.

  Note that in the single resource instance case, circle in RAG is both a necessary and a sufficient condition.
  Note that the algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices of the graph.

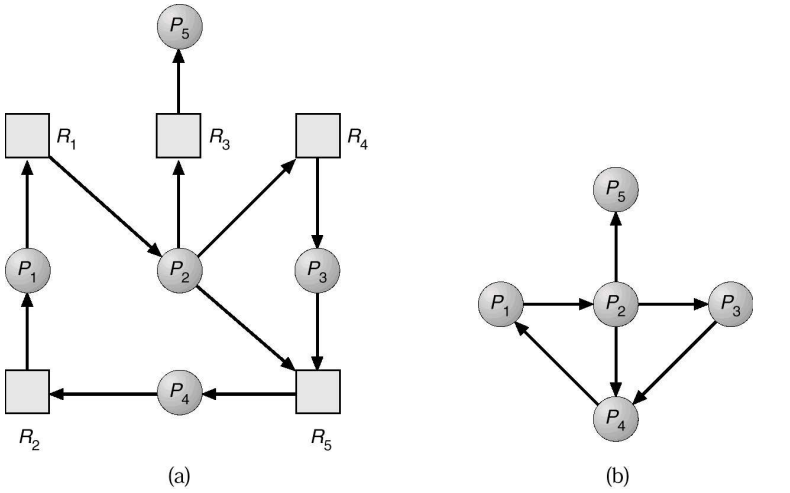


Figure 7.8 Wait-for graph

- Multiple instance case
  - Data Structures:
    - *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* [*ij*] = *k*, then process P_i is requesting *k* more instances of resource type. *Rj*.
  - 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 $Allocation_i$!= 0, then *Finish*[i] = false;
   otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] = *false*
   (c) $Request_i$ <= *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*] = false, for some *i*, 1 <= *i* <= *n*, then the system is in deadlock state.
   Moreover, if *Finish*[*i*] = *false*, then P_i is deadlocked.

   Note, the algorithm requires an order of $m \times n^2$ operations to detect whether the system is in deadlocked state.

- Usage of Deadlock Detection Algorithms
  - When, and how often, to invoke the detection algorithm depends on:
    - How often a deadlock is likely to occur?
    - How many processes will need to be rolled back?
  - If the detection algorithm is invoked at arbitrarily points in time, 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.

6.3.2 Deadlock Recovery

- Process Termination
  - Abort all deadlocked processes, or
  - Abort one deadlocked process at a time until every deadlock cycle is eliminated.
  - In which order should we choose processes to abort?
    - Priority of the process.
    - Process age and remaining time
    - Resources the process has used.
    - Resources the process needs to complete.
    - How many processes will need to terminate.
    - Is the process interactive or batch?

- Resource Preemption
  - Selecting a victim – minimize cost.
  - Rollback – return to some safe state, restart process fro that state.

- Starvation – same process is always picked as a victim, include the number of rollback in cost factor.

6.4 Combined Approach

- Combine the three basic approaches
  - Prevention
  - Avoidance
  - Detection
- Partition resources into hierarchically ordered classes.
- Use most appropriate technique for handling deadlocks within each class.