Deadlock (III)

Banker's Algorithm

- Multiple instances
- Each process must claim maximum use in advance
- 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**: $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n 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

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

Safety Algorithm

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

   Work = *Available*

   Finish [*i*] = *false*, for $i = 0, 1, \ldots, n-1$

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

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*

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	

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 ≤ 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 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?

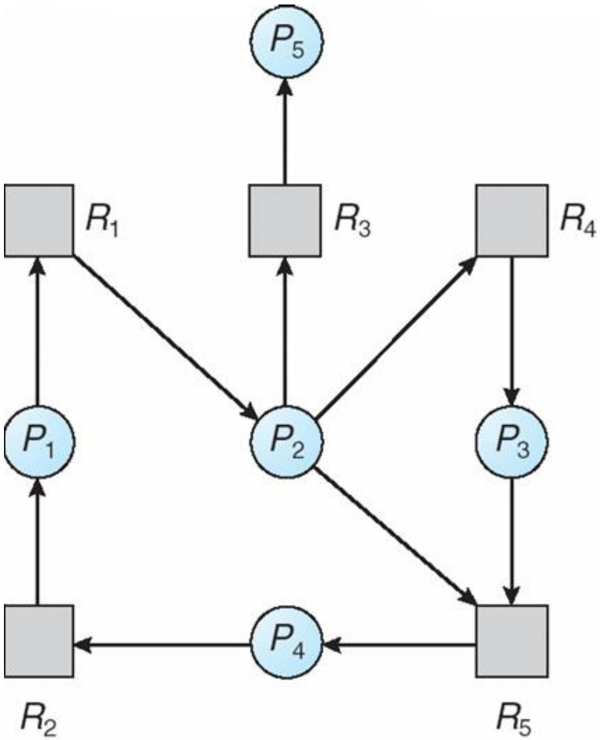
Deadlock Detection

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

Single Instance of Each Resource Type

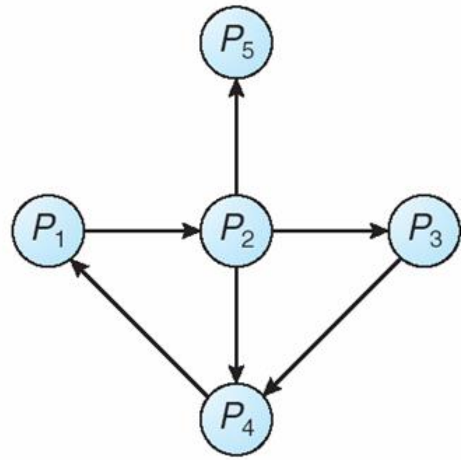
- Maintain *wait-for* graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



(a)

Resource-Allocation Graph



(b)

Corresponding wait-for graph

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.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively
   Initialize:
   (a) $Work = Available$
   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then
   $Finish[i] = \text{false}$; 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

Detection Algorithm (Cont.)

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2

4. If $Finish[i] == \text{false}$, 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	

- Execution sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*. *So, the system is not deadlocked.*

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 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

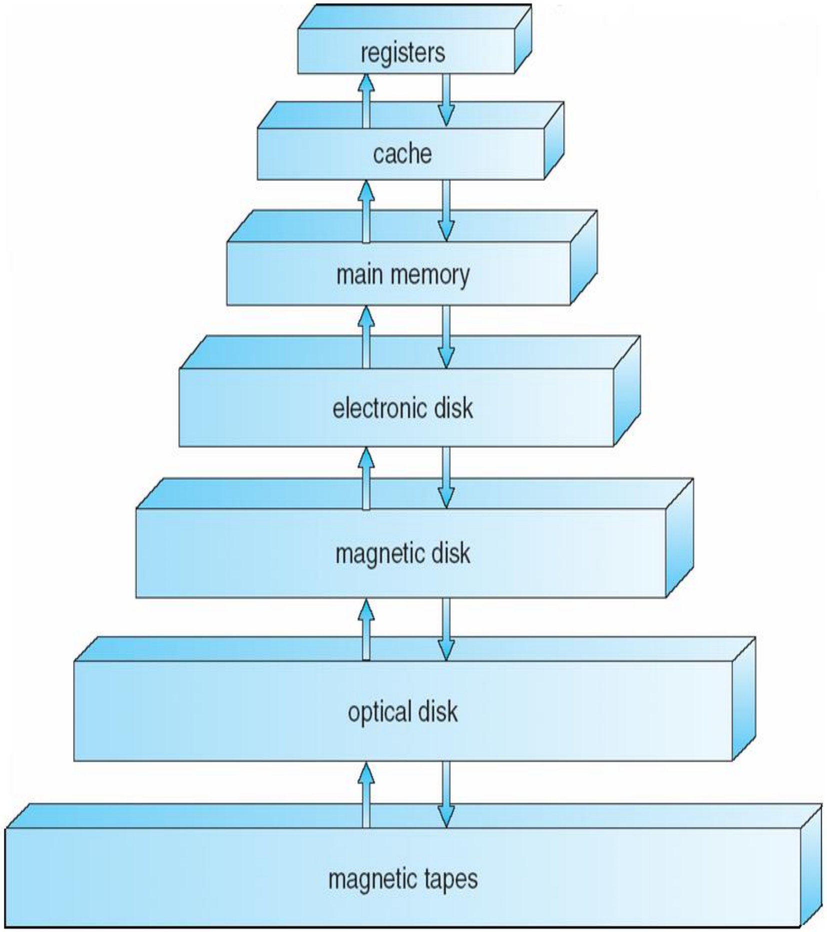
- When, and how often, to invoke detection?
  - Whenever a resource request is made
    - Maybe too frequent
    - If the maximum resource demand of every process is known, this can prevent deadlock
  - Whenever a resource request cannot be satisfied
    - Can identify the process which "finally" causes deadlock
  - Every certain time interval
    - 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

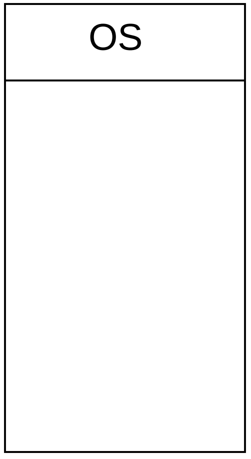
- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
  - In which order should we choose to abort?
    - Priority of the process
    - How long process has computed, and how much longer to completion
    - Resources the process has used
    - Resources process needs to complete
    - Is process interactive or batch?

Storage Hierarchy

- ❑Main memory, cache and registers are the only storages that CPU can access directly
- ❑Program must be brought (from disk) into memory and placed within a process image for it to be run

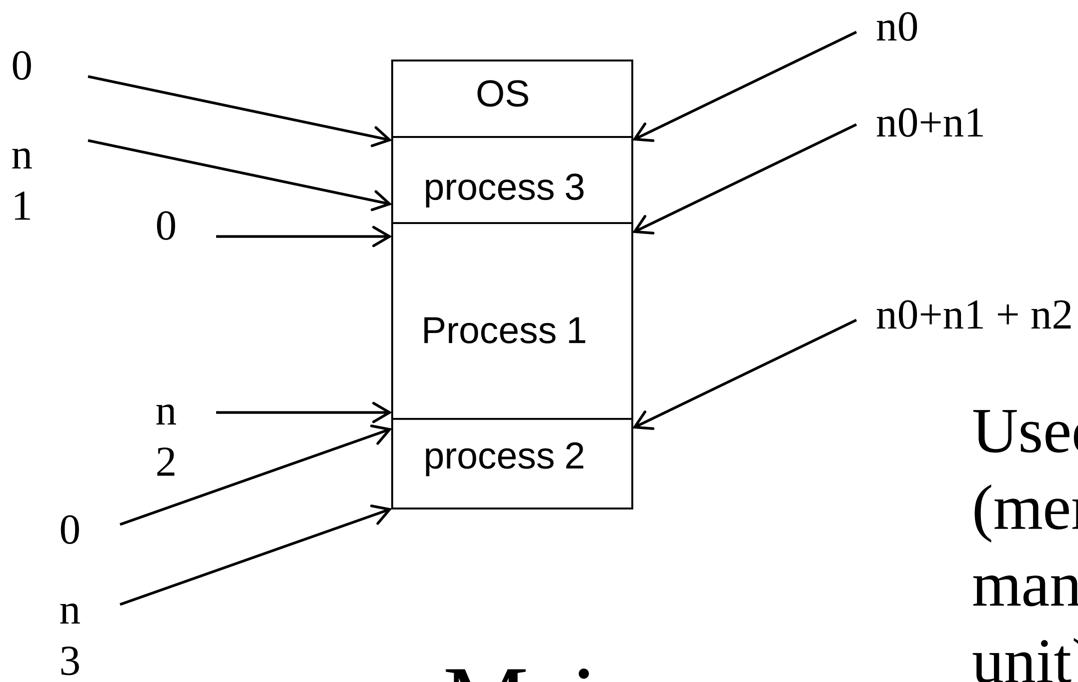


- Main memory is usually divided into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory



Logical Addresses Physical Addresses

Used in code, by CPU

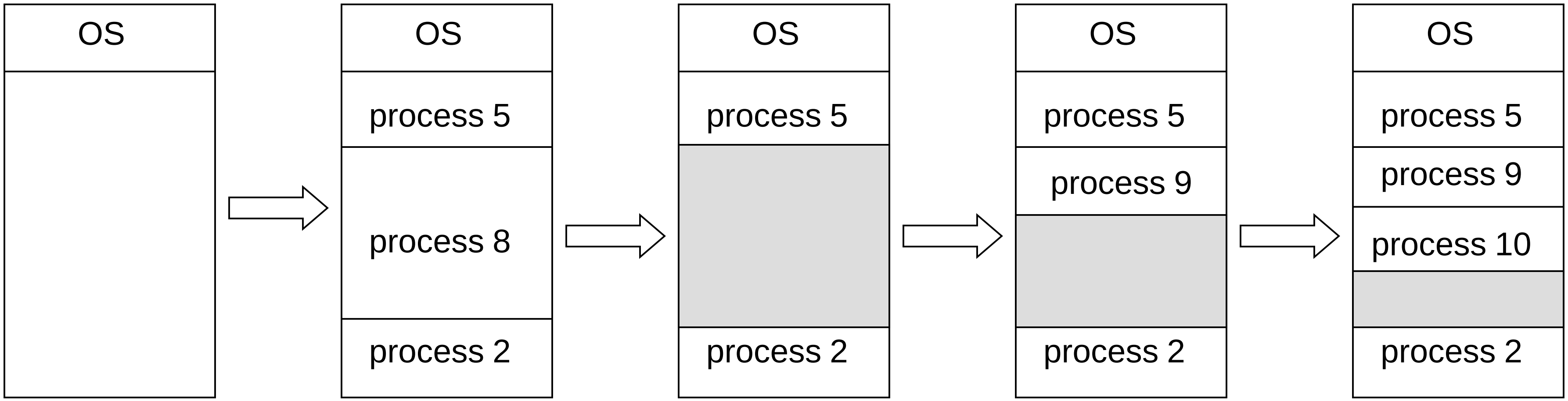


Used by MMU (memory management unit)

Main Memory

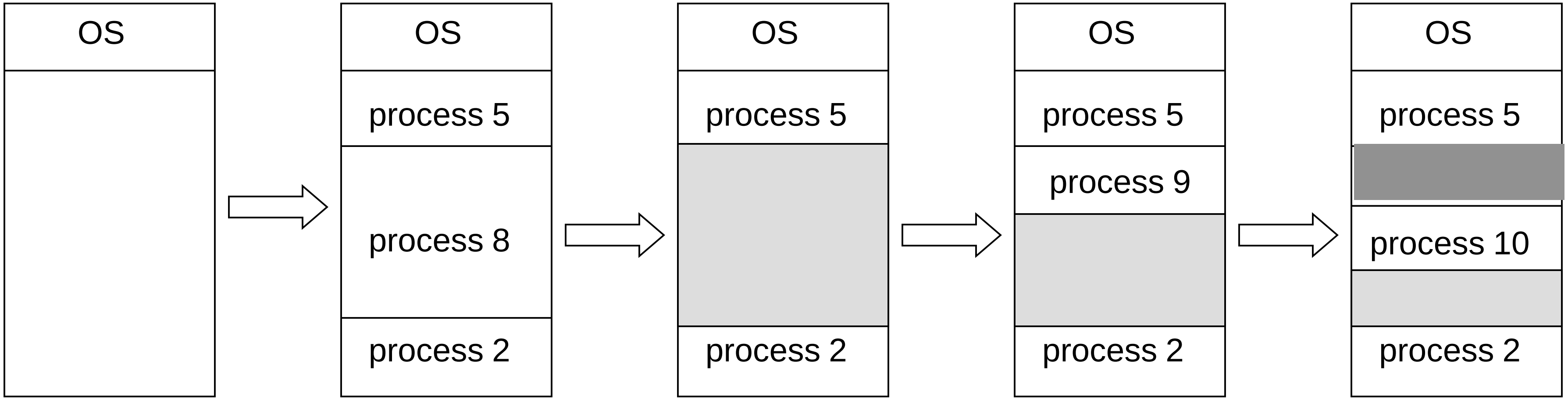
Contiguous Allocation

- In user memory space, each process is stored in a contiguous region (block).
- Hole – block of un-occupied contiguous memory space
- At the beginning, there is a single hole in the memory: the whole space for user processes



Contiguous Allocation

- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Operating system maintains information about:
  a) allocated partitions   b) free partitions (holes)
- Holes of various sizes may be generated later on and are scattered throughout memory



Contiguous Allocation Policies

How to satisfy a request of size *n* from a list of free holes

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Problem: Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time

Paging – Memory management Strategy adopted by modern OSes

- Objective:
  - Logical address space of a process remains contiguous but the physical address space of it needs not be contiguous
  - Process is allocated physical memory whenever the latter is available

Paging: Key Ideas

- Divide physical memory (user memory part) into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory space of a process into blocks of same size called **pages**
- Pages are mapped to frames one-by-one; process-specific page table records the mapping and facilitates the logical to physical address translation
- OS keeps track of free frames and allocates frames to new/swap-in processes

Paging Model and Page Table