Operating Systems

*Narasimhulu M*_{M. Tech.}
Assistant Professor
Department of Computer Science & Engineering

S.No.	Course Outcomes	Cognitive Level
1	Explain the fundamentals of operating systems like process, memory, storage, file system, security and protection.	Understand
2	Illustrate various operating System services, interfaces and system calls.	Apply
3	Demonstrate critics of process management and IPC.	Apply
4	Implement page replacement algorithms, memory management techniques and deadlock issues.	Apply
5	Illustrate architecture of file systems and I/O systems for mass storage structures.	Apply
6	Utilize the methods of operating system security and protection.	Apply

Unit 3: Memory Management

- **Memory Management Concepts:** Contiguous Memory Allocation, Swapping, Paging, Page Replacement algorithms, Thrashing, Memory Compression.

- **Deadlocks:** System Model, Deadlock Characterization, Methods of handling Deadlocks, Deadlock prevention, Detection and Avoidance, Recovery from deadlock.

10/15/2022   Prepared by: M. Narasimhulu, CSE, Assistant Professor   3

Unit 3 - Memory Management

Narasimhulu M$_{M.\ Tech.}$
Assistant Professor
Department of Computer Science & Engineering

Chapter 1
Memory Management Concepts

*Narasimhulu M**M. Tech.*

Assistant Professor

Department of Computer Science & Engineering

Contiguous Memory Allocation

*Narasimhulu M**M. Tech.*

Assistant Professor

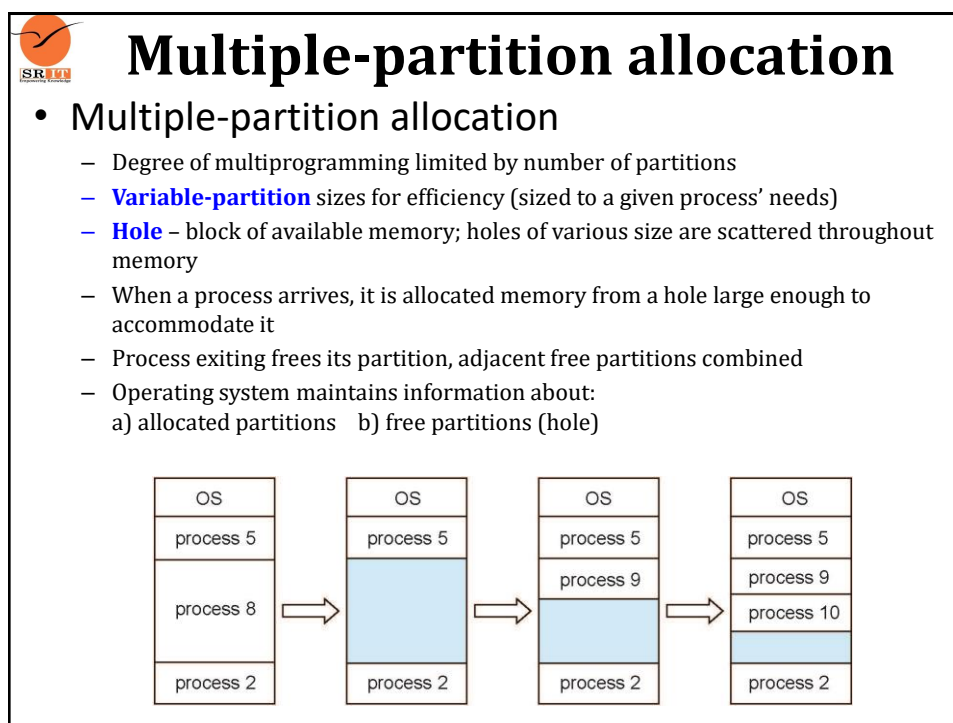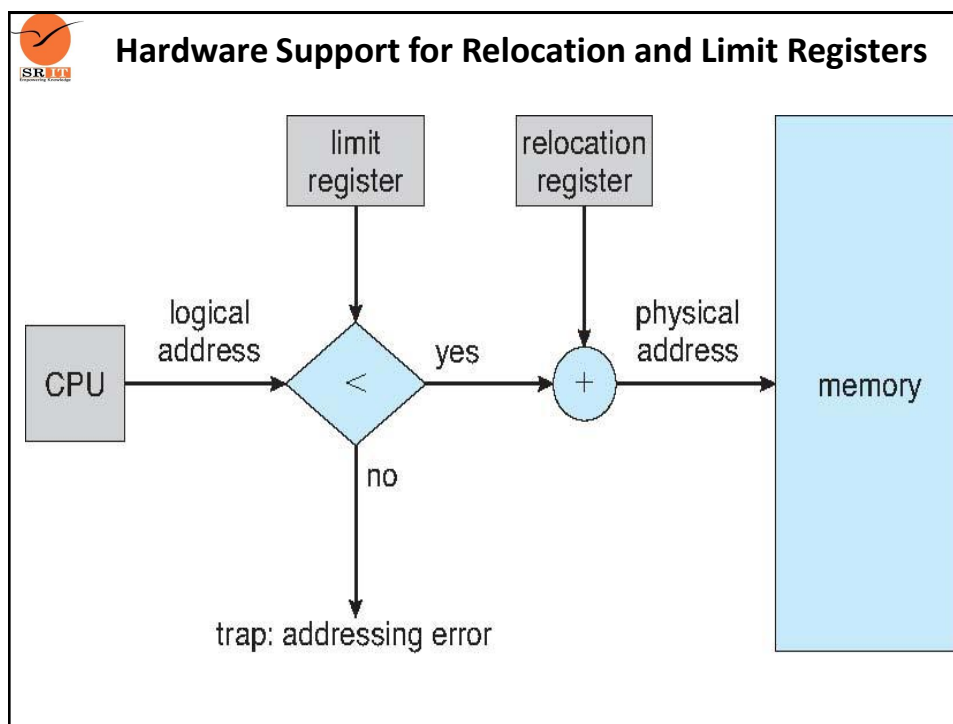*Department of Computer Science & Engineering*

Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size

Hardware Support for Relocation and Limit Registers

Multiple-partition allocation

• Multiple-partition allocation
  – Degree of multiprogramming limited by number of partitions
  – **Variable-partition** sizes for efficiency (sized to a given process' needs)
  – **Hole** – block of available memory; holes of various size are scattered throughout memory
  – When a process arrives, it is allocated memory from a hole large enough to accommodate it
  – Process exiting frees its partition, adjacent free partitions combined
  – Operating system maintains information about:
    a) allocated partitions   b) free partitions (hole)

Dynamic Storage-Allocation Problem

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

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  – 1/3 may be unusable -> **50-percent rule**

Fragmentation (Cont.)

- 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
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

Swapping

Narasimhulu M$_{M.\ Tech.}$
Assistant Professor
Department of Computer Science & Engineering

Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Schematic View of Swapping
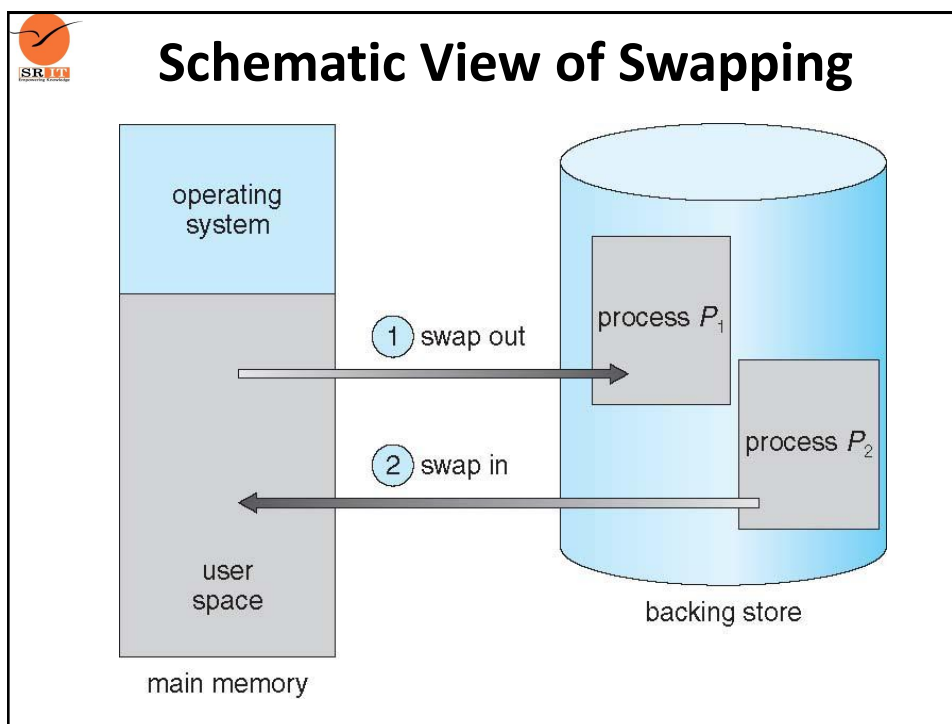
Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low

Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS *asks* apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below

Paging

*Narasimhulu M*_{M. Tech.}
Assistant Professor
Department of Computer Science & Engineering

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size N pages, need to find N free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
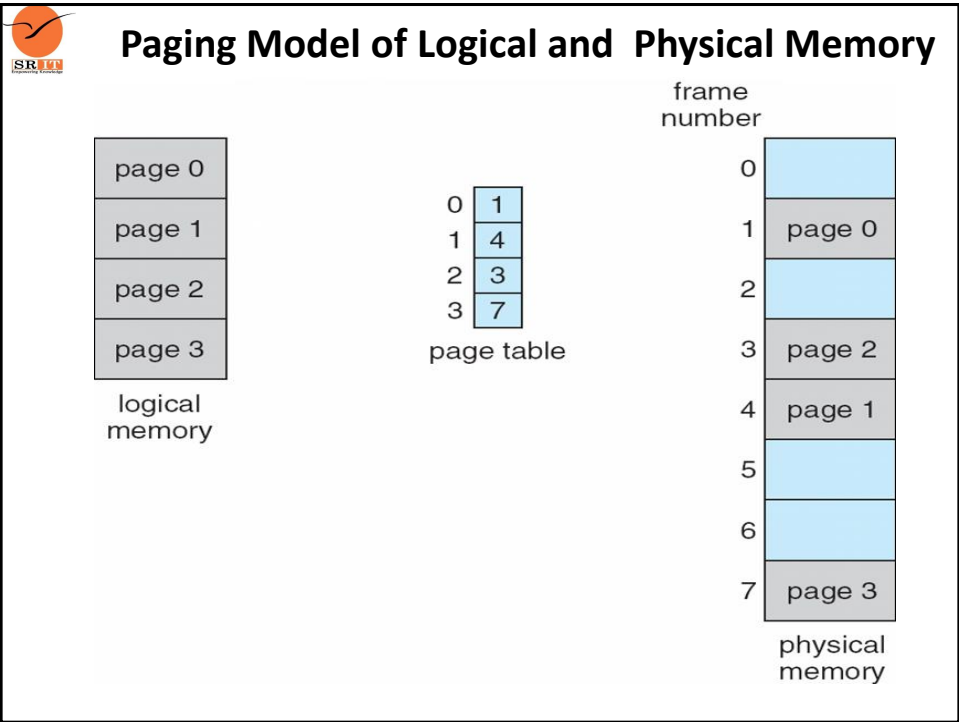- Still have Internal fragmentation

Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number** (**p**) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (**d**) – combined with base address to define the physical memory address that is sent to the memory unit

page number	page offset
p	d
m -n	n

  - For given logical address space 2^m and page size 2^n

Paging Hardware

Paging Model of Logical and Physical Memory

Paging Example

n=2 and m=4   32-byte memory and 4-byte pages
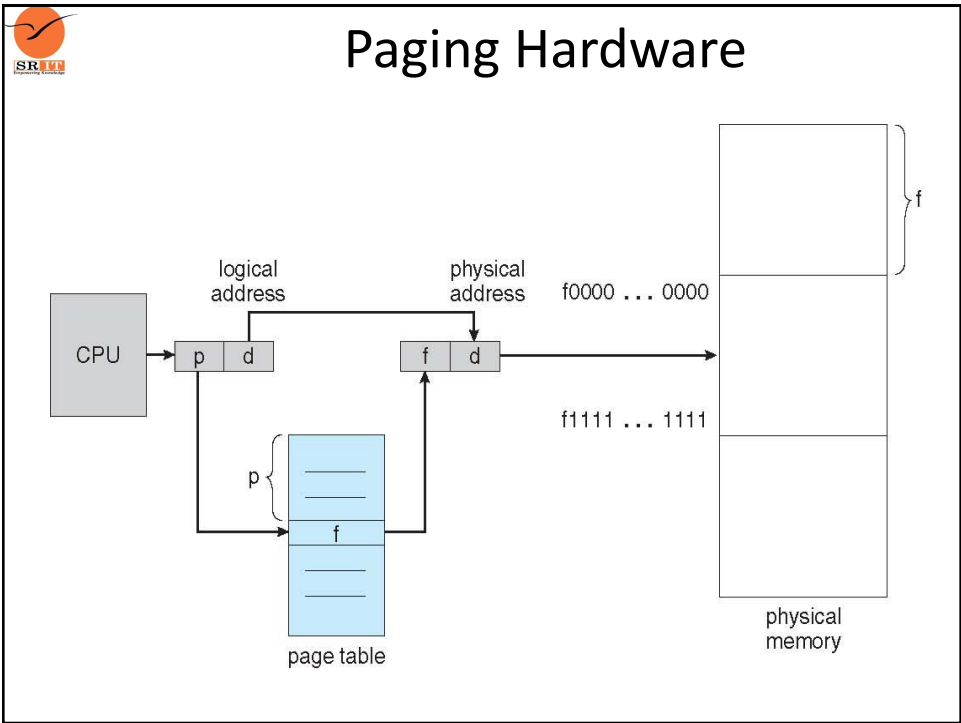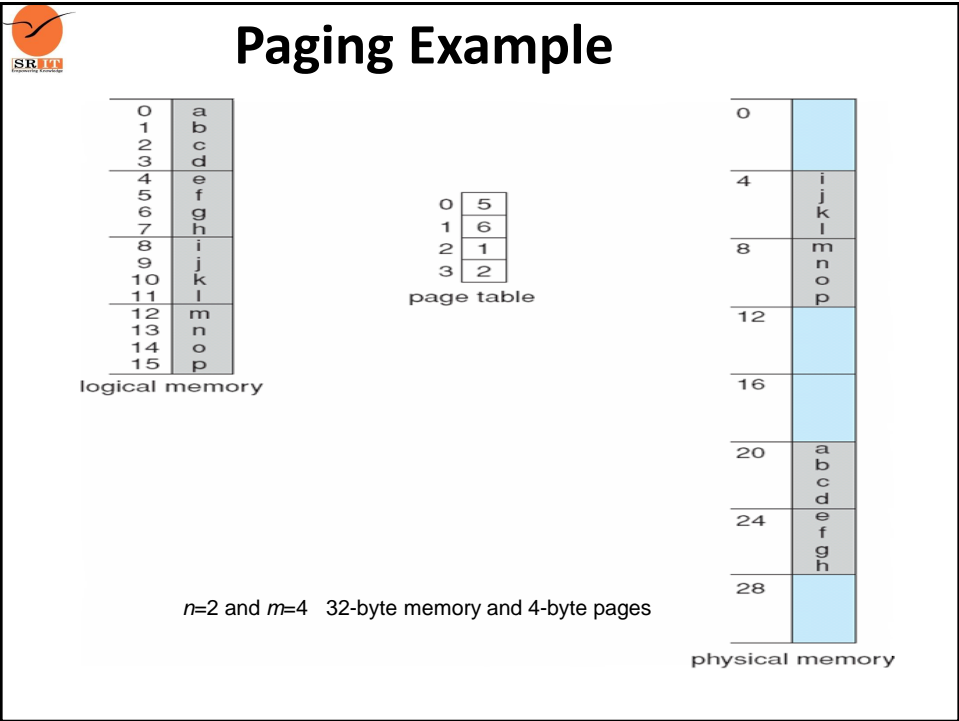
Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory.

Free Frames
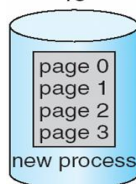
(a) Before allocation
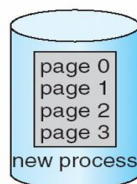
(b) After allocation
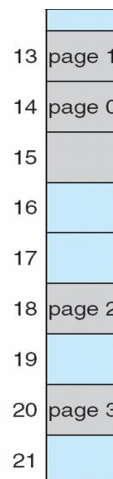
Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)

Page Replacement algorithms

*Narasimhulu M*_{M. Tech.}
Assistant Professor
Department of Computer Science & Engineering

Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement

- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk

- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Need For Page Replacement
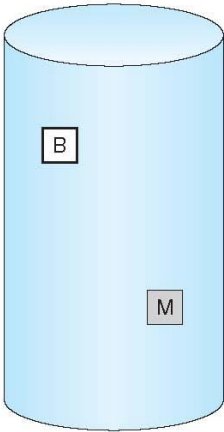
Basic Page Replacement

1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
       - Write victim frame to disk if dirty

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT
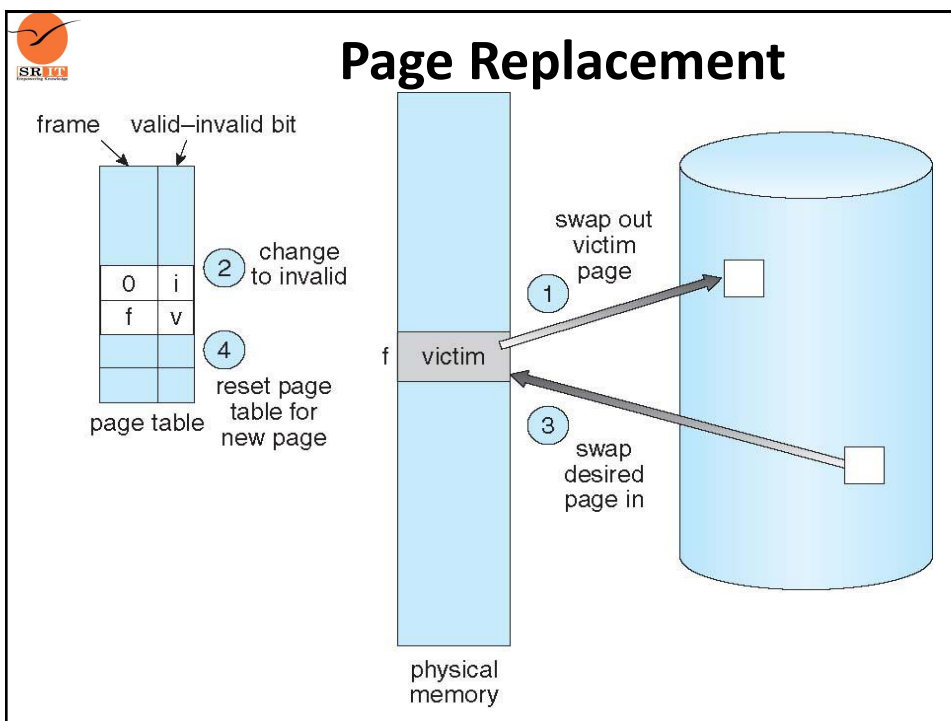
Page Replacement

Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

  7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus The Number of Frames
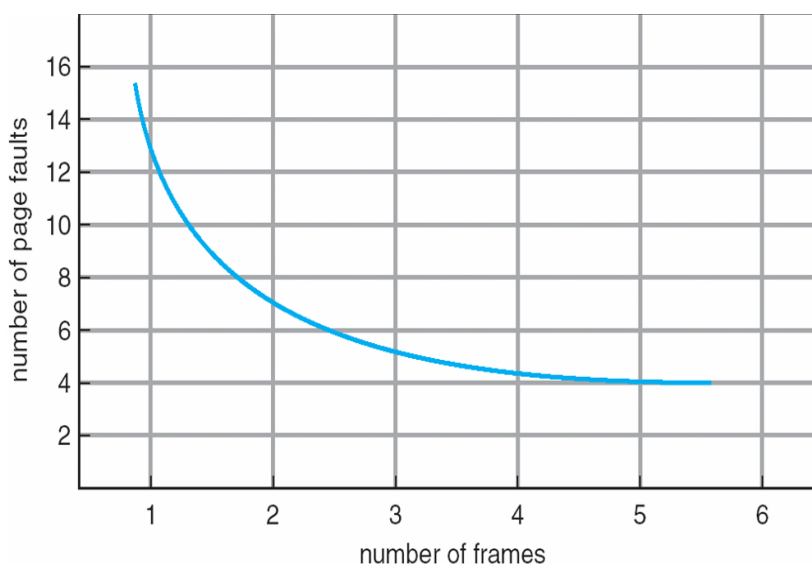
First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)
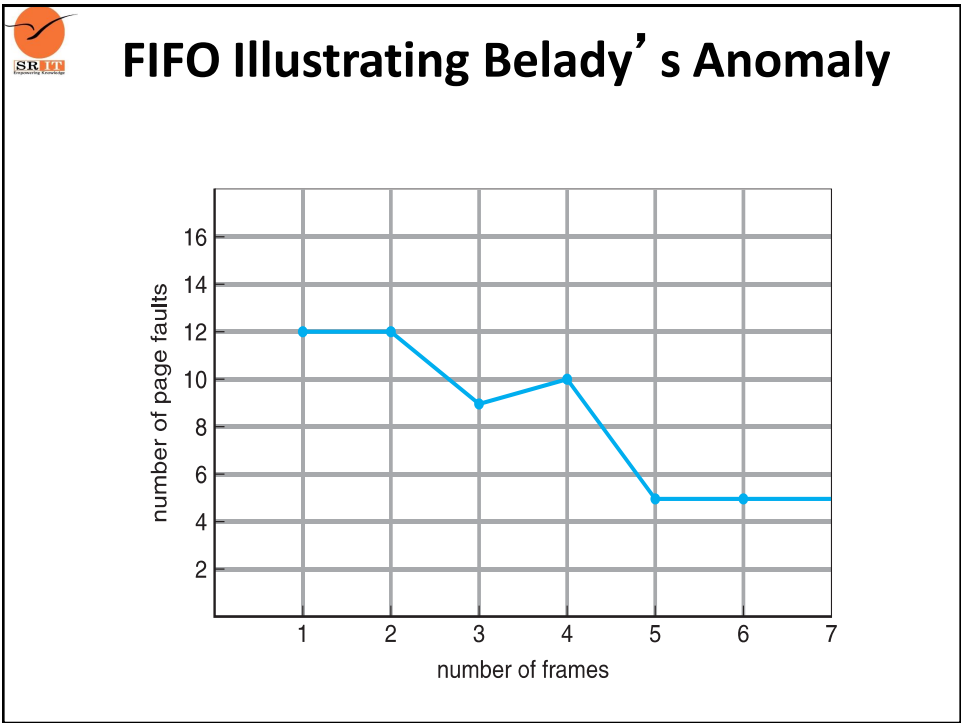
reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
0	0	0		3	3	3	2	2	2			1	1			1	0	0
1	1		1	0	0	0	3	3			3	2			2	2	1

page frames

                        15 page faults

- Can vary by reference string: consider 2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

FIFO Illustrating Belady's Anomaly

Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames

- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

Use Of A Stack to Record Most Recent Page References

reference string

4 7 0 7 1 0 1 2 1 2 7 1 2

                                    a   b

stack before a
2
1
0
7
4

| stack after b |
|:---:|
| 7 |
| 2 |
| 1 |
| 0 |
| 4 |

LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

Second-Chance (clock) Page-Replacement Algorithm
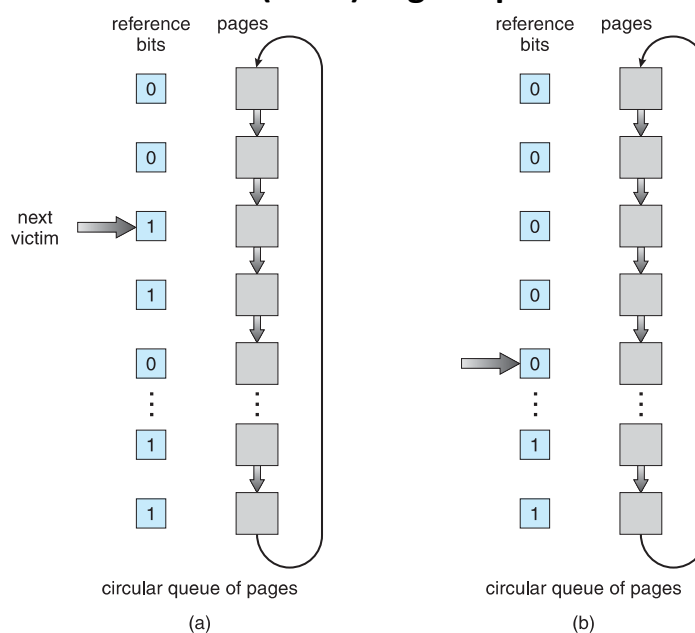
Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)
1. (0, 0) neither recently used not modified – best page to replace
2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
3. (1, 0) recently used but clean – probably will be used again soon
4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
    - Might need to search circular queue several times

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
    - Not common

- **Lease Frequently Used** (**LFU**) **Algorithm**: replaces page with smallest count

- **Most Frequently Used** (**MFU**) **Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc

Thrashing

Narasimhulu M$_{M.\ Tech.}$
Assistant Professor
Department of Computer Science & Engineering

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

- **Thrashing** $\equiv$ a process is busy swapping pages in and out
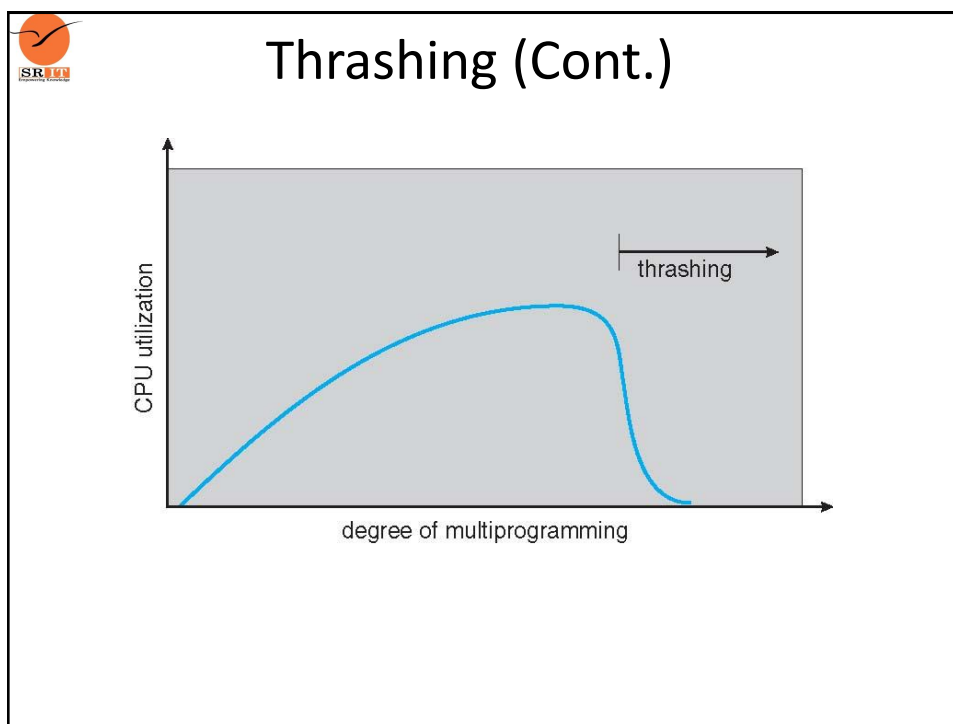
Thrashing (Cont.)

Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?
  Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern
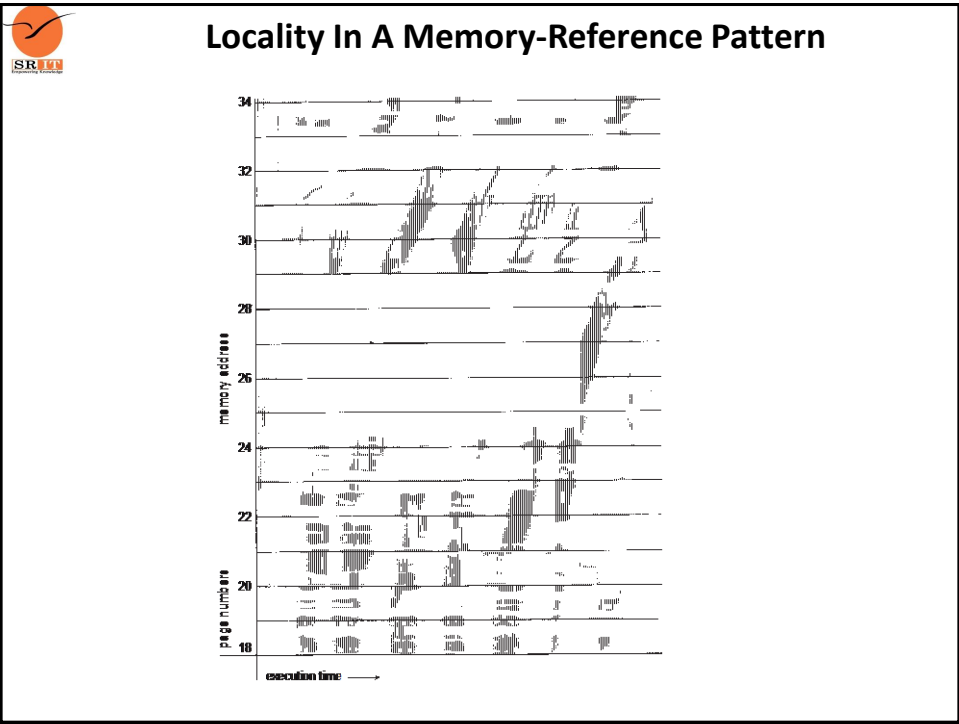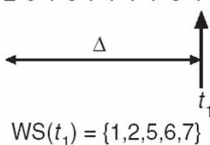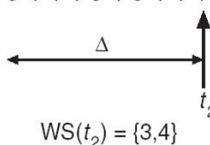

Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

page reference table

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .

Δ t_1 Δ t_2

WS(t_1) = {1,2,5,6,7}   WS(t_2) = {3,4}

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
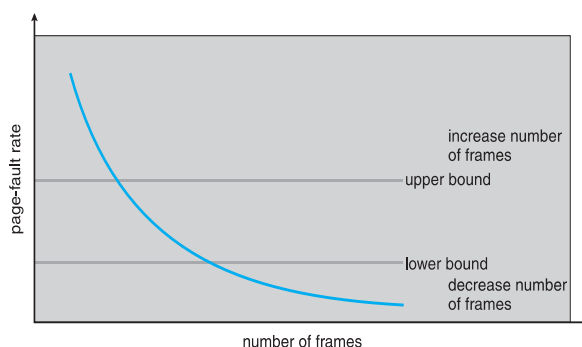
Working Sets and Page Fault Rates

☐ Direct relationship between working set of a process and its page-fault rate

☐ Working set changes over time

☐ Peaks and valleys over time
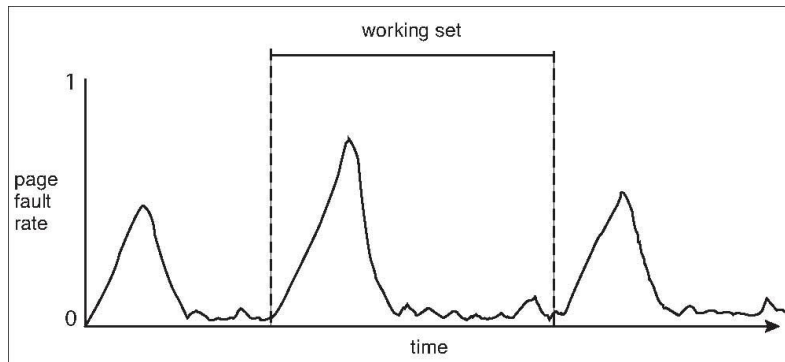
Memory Compression

Narasimhulu M$_{M. Tech.}$
Assistant Professor
Department of Computer Science & Engineering

END of Chapter - 1

Chapter 2
Deadlocks

Narasimhulu M$_{M. Tech.}$
Assistant Professor
Department of Computer Science & Engineering

System Model

Narasimhulu M$_{M. Tech.}$
Assistant Professor
Department of Computer Science & Engineering

System Model

- System consists of resources
- Resource types R_1, R_2, . . ., 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**

Deadlock Characterization

Narasimhulu M$_{\text{M. Tech.}}$
Assistant Professor
Department of Computer Science & Engineering

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 $\{P_0, P_1, …, P_n\}$ 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, …, 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.

Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc.
- See example box in text page 318 for mutex deadlock
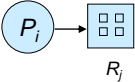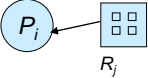
Resource-Allocation Graph

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

- V is partitioned into two types:
  - $P = \{P_1, P_2, …, P_n\}$, the set consisting of all the processes in the system

  - $R = \{R_1, R_2, …, R_m\}$, 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 (Cont.)

- Process

- Resource Type with 4 instances

- P_i requests instance of R_j

$P_i \rightarrow R_j$

- P_i is holding an instance of R_j

$P_i \leftarrow R_j$

Example of a Resource Allocation Graph

Resource Allocation Graph With a Deadlock

Graph With A Cycle But No Deadlock
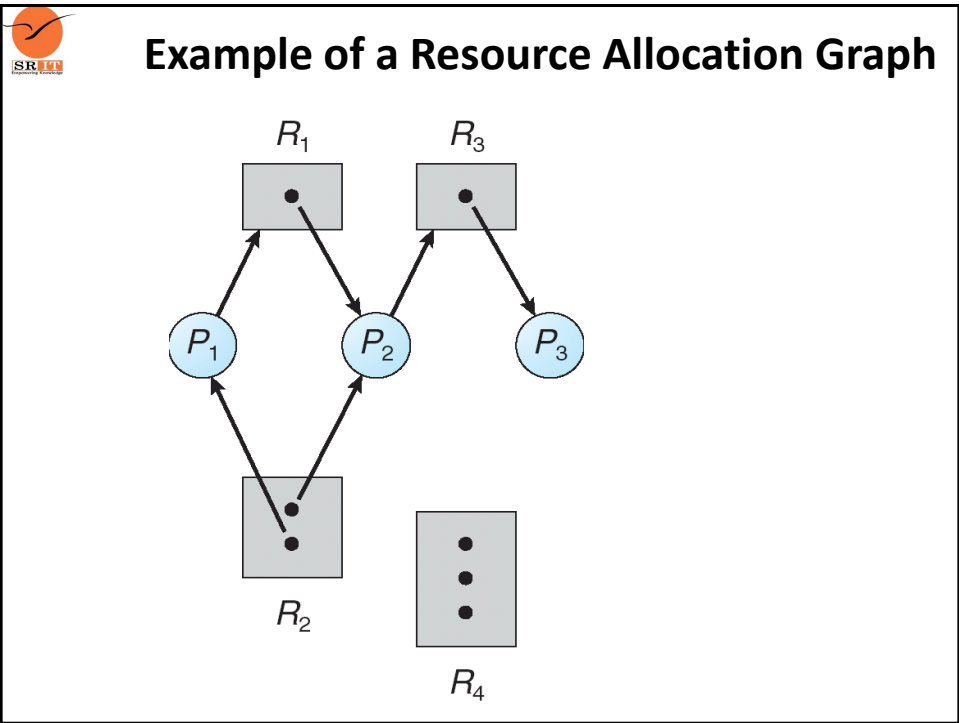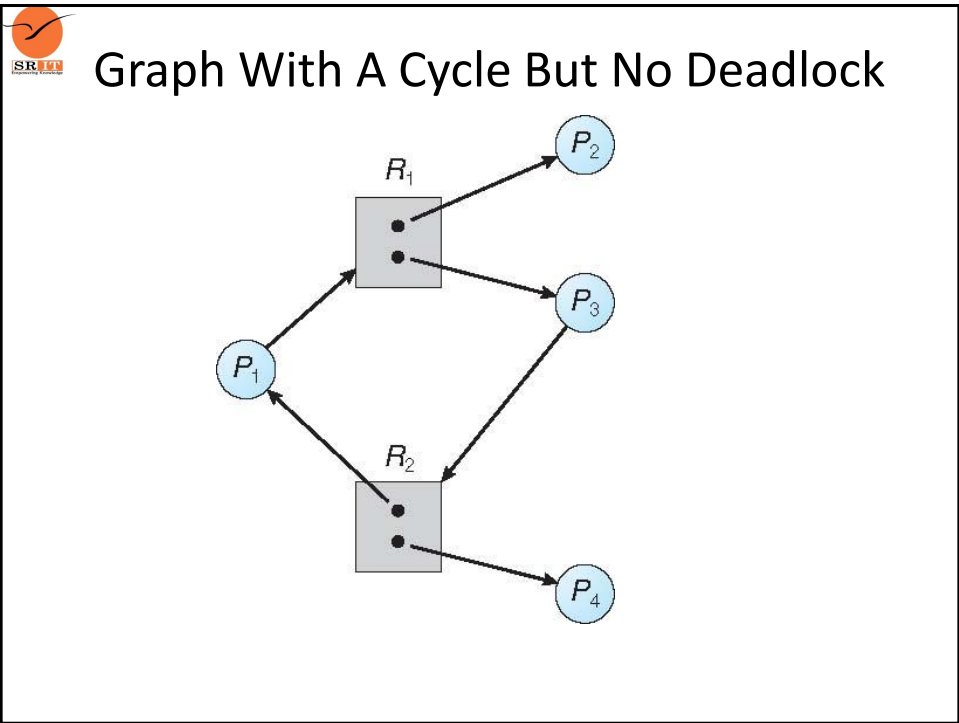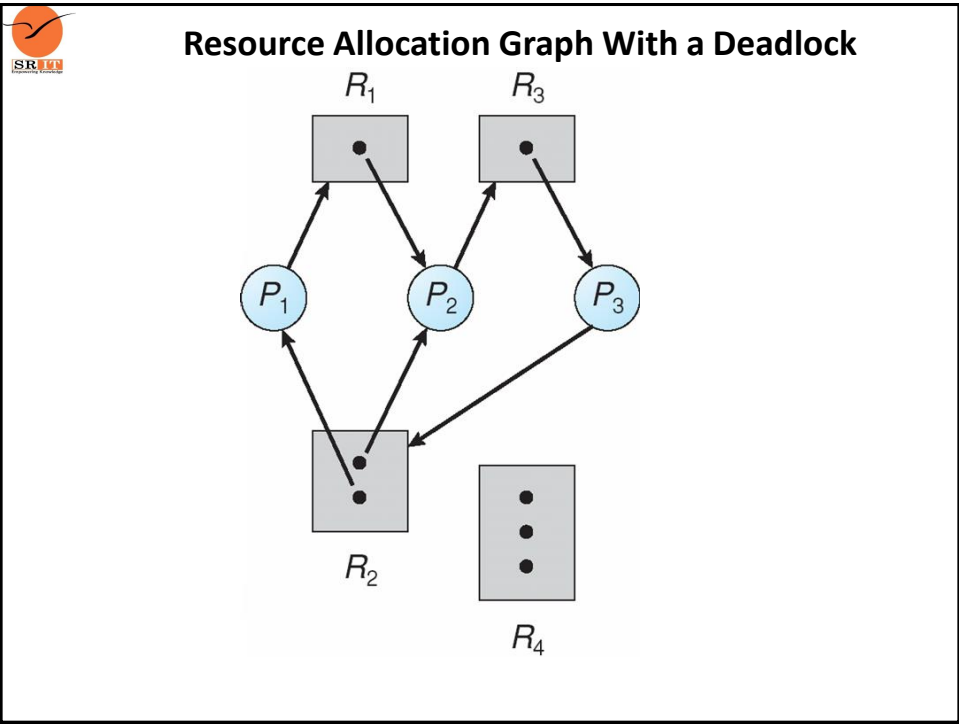
Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods of handling Deadlocks

Narasimhulu M$_{M.\ Tech.}$
Assistant Professor
Department of Computer Science & Engineering

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- 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

Deadlock prevention

Narasimhulu M$_{M. Tech.}$
Assistant Professor
Department of Computer Science & Engineering

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); 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 allocated to it.
  - Low resource utilization; starvation possible

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

Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```

Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);
    acquire(lock1);
        acquire(lock2);
            withdraw(from, amount);
            deposit(to, amount);
        release(lock2);
    release(lock1);
}
```

Transactions 1 and 2 execute concurrently. Transaction 1 transfers $25 from account A to account B, and Transaction 2 transfers $50 from account B to account A

Detection and Avoidance

Narasimhulu M$_{M. Tech.}$
Assistant Professor
Department of Computer Science & Engineering

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

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 sequence $<P_1, P_2, ..., P_n>$ of ALL the processes in the systems such that 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 $j < I$

- That is:
  - 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

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks

- If a system is in unsafe state $\Rightarrow$ possibility of deadlock

- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State
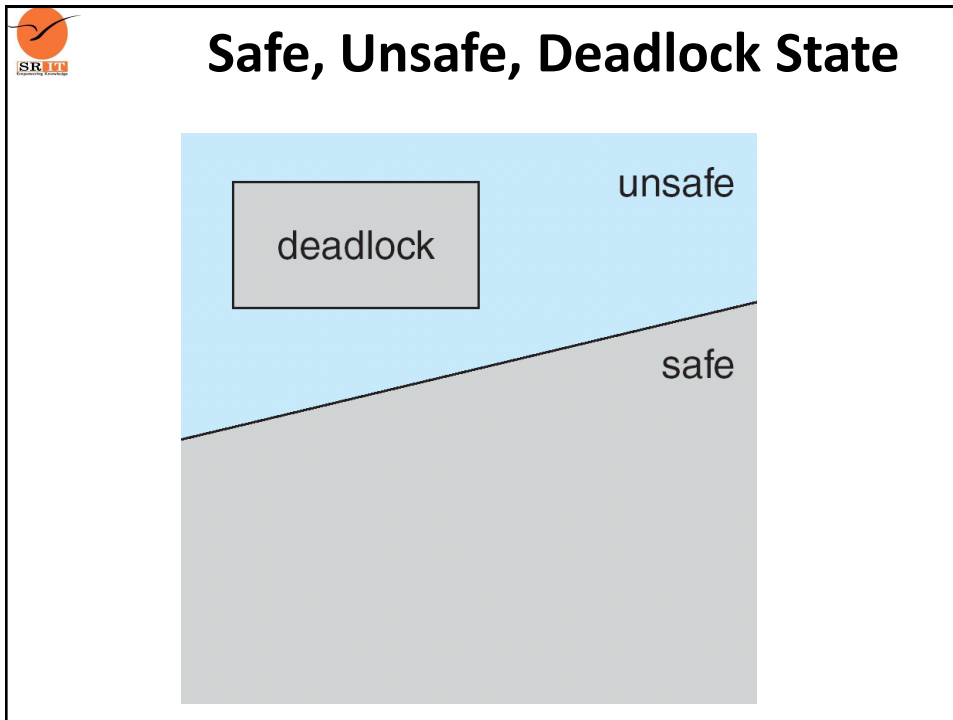
Avoidance Algorithms

- 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

Resource-Allocation Graph

Unsafe State In Resource-Allocation Graph
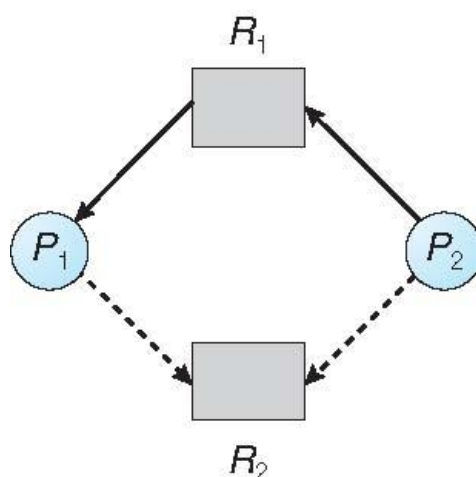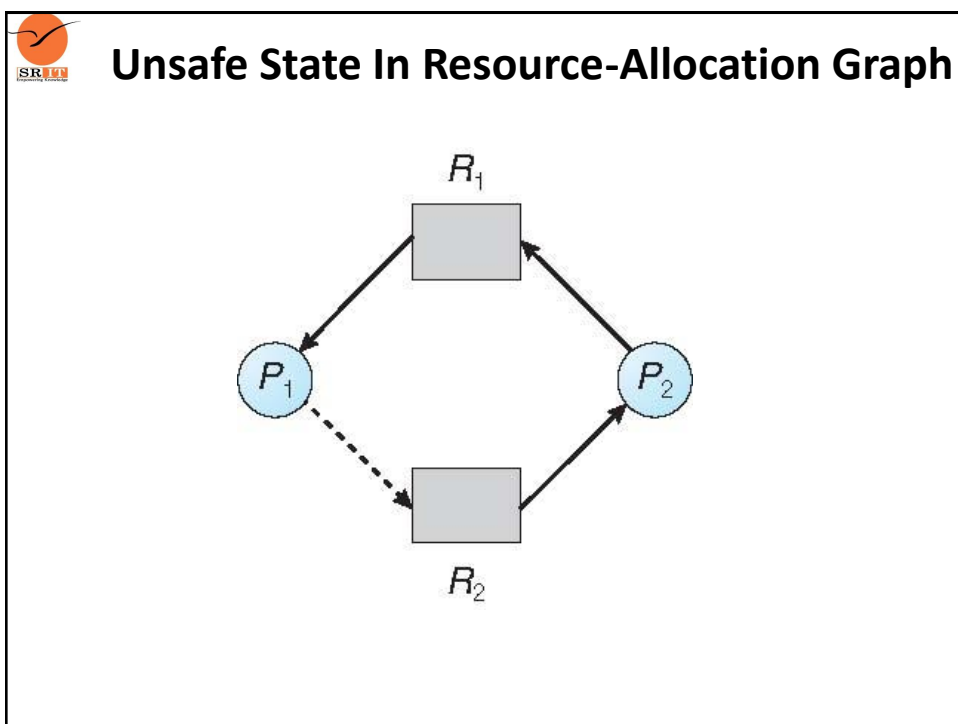
Resource-Allocation Graph Algorithm

- 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

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

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 x 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, …, *n*- 1

2. Find an *i* such that both:
   - (a) **Finish [*i*] = false**
   - (b) **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-Request Algorithm for Process P_i

Request$_i$ = 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$_i$ ≤ Need$_i$** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If **Request$_i$ ≤ 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 ⟹ the resources are allocated to **P_i**
   - If unsafe ⟹ **P_i** 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**

	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 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 $\leq$ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ 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	

- 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)
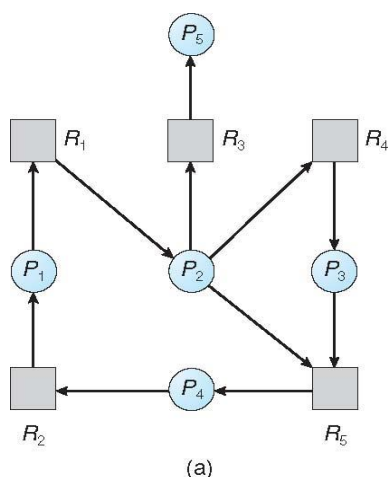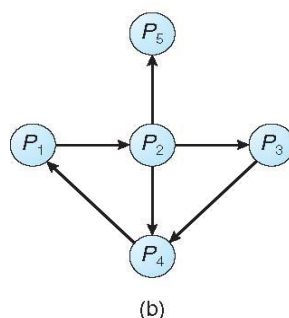
Corresponding wait-for graph

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, …, 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**

   (b) **Request$_i$ ≤ 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] == 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	

- Sequence <*P$_0$, P$_2$, P$_3$, P$_1$, P$_4$*> will result in *Finish[i] = true* for all *i*

Example (Cont.)

- P_2 requests an additional instance of type C

	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

Request

- 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

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.

Recovery from deadlock

*Narasimhulu M*_{M. Tech.}
Assistant Professor
Department of Computer Science & Engineering

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?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- **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

END of Chapter - 2

10/15/2022   Prepared by: M. Narasimhulu, CSE,   108
                            Assistant Professor

END of Unit-3

10/15/2022   Prepared by: M. Narasimhulu, CSE, Assistant Professor   109