MODULE 4:

Chapter-8
Memory Management Strategies

- Background
  - Basic Hardware
  - Address Binding
  - Logical vs Physical Address Space
  - Dynamic Loading
  - Dynamic Linking and Shared Libraries
- Swapping
- Contiguous Allocation
- Paging
- Segmentation

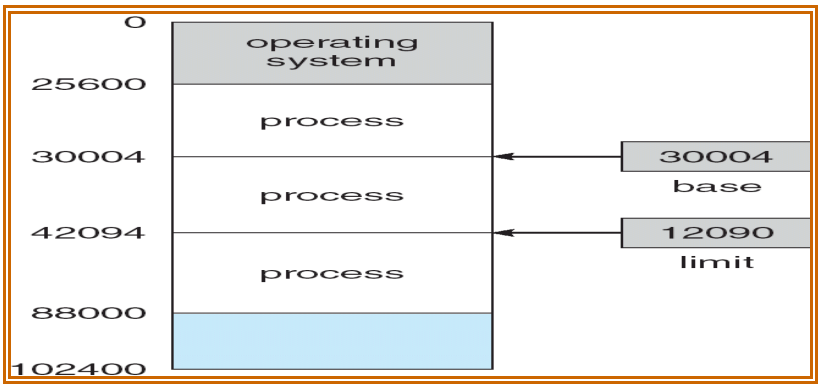
3.1 Background

- Memory consists of a large array of words or bytes, each with its own address.
- The CPU fetches instructions from memory according to the value of the program counter.
- These instructions may cause additional loading from and storing to specific memory addresses.
- CPU fetch-execute cycle
  - fetch next instruction from memory (based on program counter)
  - decode the instruction
  - possibly fetch operands from memory
  - execute the instruction
  - store the result in memory
- The memory unit sees only a stream of memory addresses

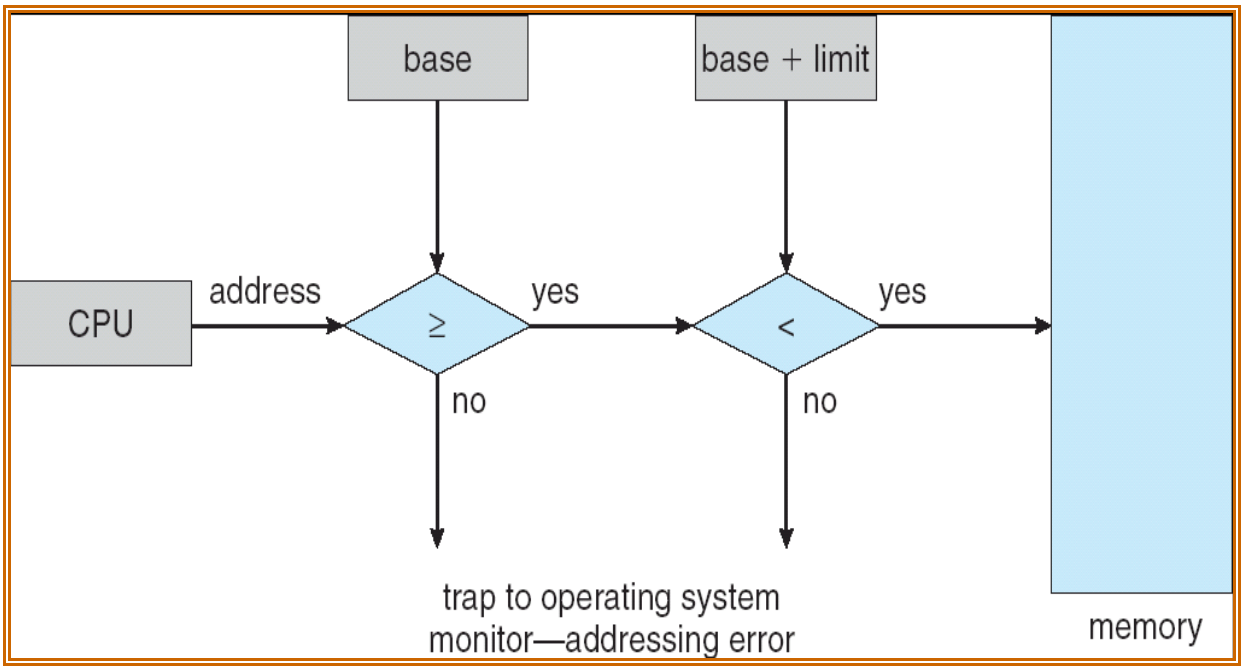
- It does not distinguish between instructions and data
- It does not care how the address was arrived

Basic Hardware

- CPU can access only storage like main memory and CPU registers.
- Thus any instruction to be executed and data being used by the instructions must be in one of these storages.
- Registers that are built into CPU are accessible within one cycle of the CPU clock.
- CPU can decode instructions and perform operations on registers contents at the rate of one or more operations per CPU clock tick.
- Memory which is accessed via a transaction on memory bus takes many CPU cycles to complete thus CPU needs to wait till data or instructions are available.
- Frequency of memory access does not match with CPU work.
- Thus remedy is to add a faster memory i.e. cache between CPU and main memory.
- Operating system should be protected from user programs and user programs from each other.
- Each process should have separate memory space.
- There should be a range of legal address that processes may access.
- This protection can be provided by using two registers i.e. base register and limit register
- The base register holds smallest legal physical address
- The limit register specifies the size of the range.
- For example:
    - Base register: 30004
    - Limit register: 12090
    - The program can access all addresses from 30004 through 42094 (inclusive)



- Protection of memory space is accomplished by having the CPU hardware compare every address generated with registers.
- Any attempt by a program executing in user mode to access operating system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error.



- The base and limit registers can be loaded only by operating system, which needs privileged instruction.
- Since privileged instruction can be executed in kernel mode and only OS executes in kernel mode.
- Thus only OS can load base and limit registers.

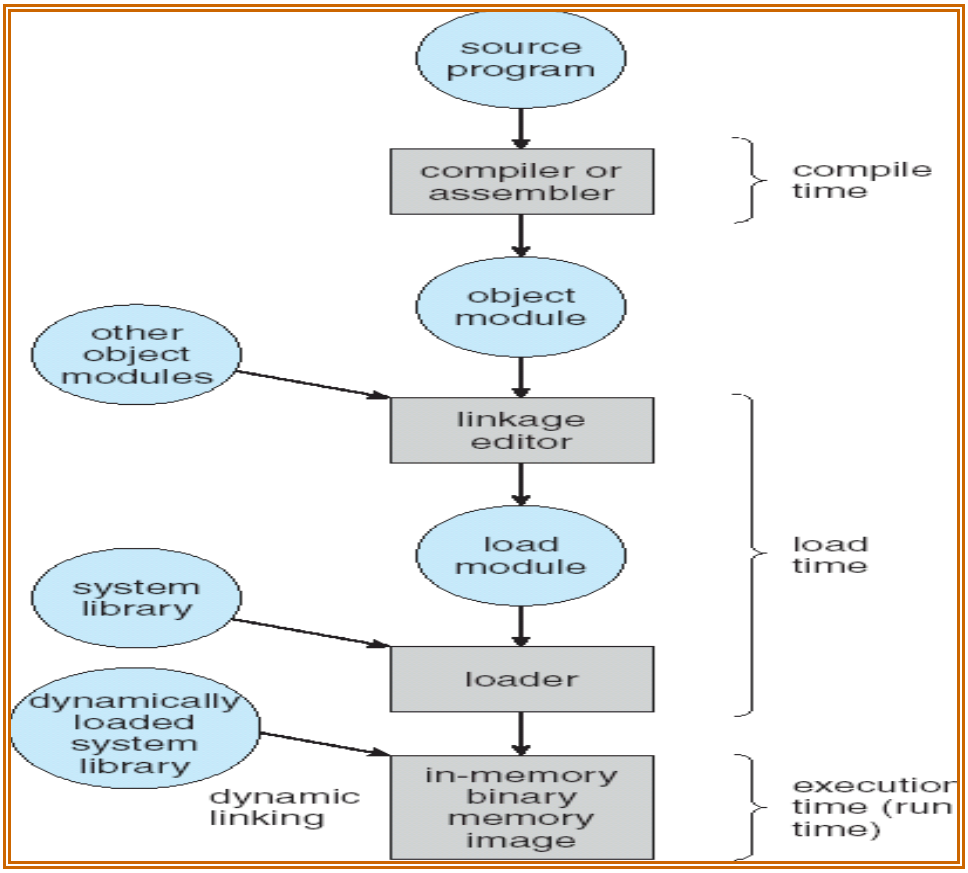
- This scheme allows OS to change the value of the registers but prevents user programs from changing registers' contents.

Address Binding

- A program resides on a disk as a binary executable file.
- Program must be brought into memory and placed within a process for it to be executed.
- The process may be moved between disk and memory during its execution depending on the memory management in use.
- Input queue – collection of processes on the disk that are waiting to be brought into memory to run the program.
- The binding of instructions and data to memory addresses can be done at any step along the way.
- Address binding of instructions and data to memory addresses can happen at three different stages:
    - Compile time
    - Load time
    - Execution time
- **Compile time**
    - If memory location is known at compile time, then absolute code can be generated
    - Compiled code will start at that location and extend up from there.
    - If, at some later time, the starting location changes, then it will be necessary to recompile this code.
- **Load time**
    - Compiler must generate relocatable code if memory location is not known at compile time
    - Final binding is delayed until load time
- **Execution time**
    - Binding is delayed until run time if the process can be moved during its execution from one memory segment to another.

- Special hardware must be available for address mapping (e.g., base and limit registers).
- Most general-purpose operating systems use this method.

Multistep Processing of a User Program



Logical vs. Physical Address Space

- The concept of a logical *address space* that is bound to a separate *physical address space* is central to proper memory management
  - Logical address – generated by the CPU; also referred to as *virtual address*
  - Physical address – address seen by the memory unit
- The set of all logical addresses generated by a program is a logical-address space.
- The set of all physical addresses corresponding to these logical addresses is a physical-address space.

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes.
- Logical (virtual) and physical addresses differ in execution-time address-binding scheme.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.

Memory-Management Unit (MMU)(page 20)

- The memory-mapping hardware device (MMU) converts logical (virtual) addresses into physical addresses.
- In MMU general scheme, the value of the relocation register (same as base register) is added to every logical address generated by a user process at the time it is sent to memory

Dynamic Loading:

- For a process to be executed it should be loaded in to the physical memory. The size of the process is limited to the size of the physical memory. x Dynamic loading is used to obtain better memory utilization..
- In dynamic loading the routine or procedure will not be loaded until it is called. x Whenever a routine is called, the calling routine first checks whether the called routine is already loaded or not. If it is not loaded it cause the loader to load the desired program in to the memory and updates the programs address table to indicate the change and control is passed to newly called routine.
- Advantage: Gives better memory utilization. x Unused routine is never loaded. x Do not need special operating system support.
- This method is useful when large amount of codes are needed to handle in frequently occurring cases.

Dynamic linking and Shared libraries:

- Some operating system supports only the static linking.
- In dynamic linking only the main program is loaded in to the memory. If the main program requests a procedure, the procedure is loaded and the link is established at the time of references.
- This linking is postponed until the execution time.

- With dynamic linking a “stub” is used in the image of each library referenced routine. A “stub” is a piece of code which is used to indicate how to locate the appropriate memory resident library routine or how to load library if the routine is not already present.
- When “stub” is executed it checks whether the routine is present is memory or not. If not it loads the routine in to the memory.
- This feature can be used to update libraries i.e., library is replaced by a new version and all the programs can make use of this library.
- More than one version of the library can be loaded in memory at a time and each program uses its version of the library. Only the program that are compiled with the new version are affected by the changes incorporated in it. Other programs linked before new version is installed will continue using older libraries this type of system is called “shared library”

3.2 CONTIGUOUS MEMORY ALLOCATION

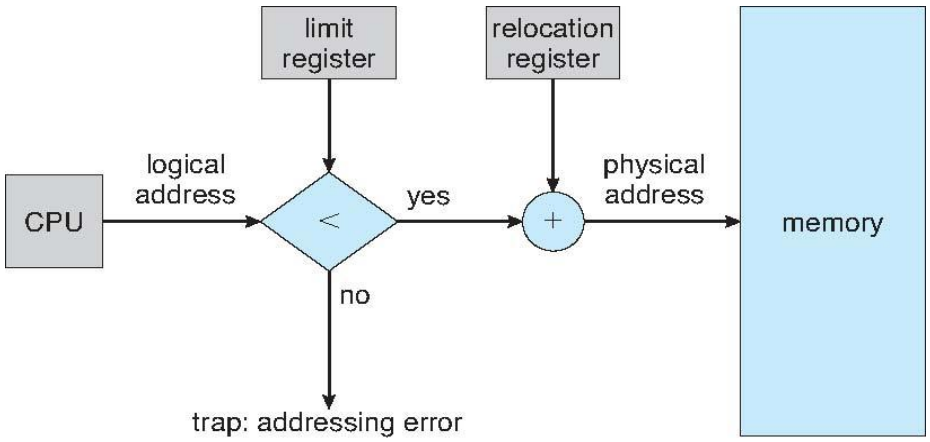
- One of the simplest methods for memory allocation is to divide memory in to several fixed partition. Each partition contains exactly one process. The degree of multi-programming depends on the number of partitions.
- In multiple partition method, when a partition is free, process is selected from the input queue and is loaded in to free partition of memory. x When process terminates, the memory partition becomes available for another process. x Batch OS uses the fixed size partition scheme.
- The OS keeps a table indicating which part of the memory is free and is occupied.
- When the process enters the system it will be loaded in to the input queue. The OS keeps track of the memory requirement of each process and the amount of memory available and determines which process to allocate the memory.
- When a process requests, the OS searches for large hole for this process, hole is a large block of free memory available.
- If the hole is too large it is split in to two. One part is allocated to the requesting process and other is returned to the set of holes.
- The set of holes are searched to determine which hole is best to allocate. There are three strategies to select a free hole:

- **First bit:-**Allocates first hole that is big enough. This algorithm scans memory from the beginning and selects the first available block that is large enough to hold the process.
- **Best bit:-**It chooses the hole i.e., closest in size to the request. It allocates the smallest hole i.e., big enough to hold the process.
- **Worst fit:-**It allocates the largest hole to the process request. It searches for the largest hole in the entire list.

- First fit and best fit are the most popular algorithms for dynamic memory allocation. First fit is generally faster.
- Best fit searches for the entire list to find the smallest hole i.e., large enough. Worst fit reduces the rate of production of smallest holes.
- All these algorithms suffer from fragmentation.

Memory Protection:

- Memory protection means protecting the OS from user process and protecting process from one another.
- Memory protection is provided by using a re-location register, with a limit register. x Relocation register contains the values of smallest physical address and limit register contains range of logical addresses. (Re-location = 100040 and limit = 74600).
- The logical address must be less than the limit register, the MMU maps the logical address dynamically by adding the value in re-location register.
- When the CPU scheduler selects a process for execution, the dispatcher loads the re-location and limit register with correct values as a part of context switch.
- Since every address generated by the CPU is checked against these register we can protect the OS and other users programs and data from being modified.



Fragmentation:

- In Internal Fragmentation there is wasted space internal to a portion due to the fact that block of data loaded is smaller than the partition. Eg:-If there is a block of 50kb and if the process requests 40kb and if the block is allocated to the process then there will be 10kb of memory left.
- External Fragmentation exists when there is enough memory space exists to satisfy the request, but it not contiguous i.e., storage is fragmented in to large number of small holes.
- External Fragmentation may be either minor or a major problem.
- One solution for over-coming external fragmentation is compaction. The goal is to move all the free memory together to form a large block. Compaction is not possible always. If the relocation is static and is done at load time then compaction is not possible. Compaction is possible if the re-location is dynamic and done at execution time.
- Another possible solution to the external fragmentation problem is to permit the logical address space of a process to be non-contiguous, thus allowing the process to be allocated physical memory whenever the latter is available.

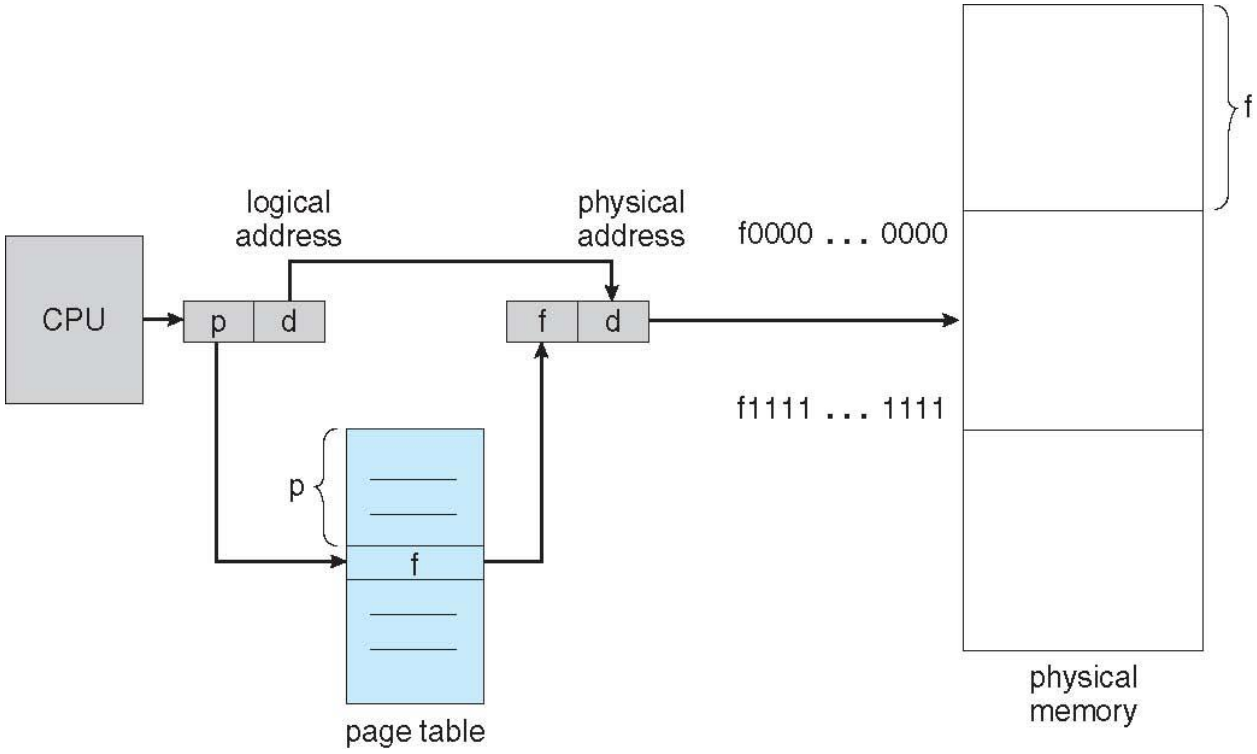
3.3 PAGING AND STRUCTURE OF PAGE TABLE

- Paging is a memory management scheme that permits the physical address space of a process to be non-contiguous. Support for paging is handled by hardware. It is used to avoid external fragmentation.
- Paging avoids the considerable problem of fitting the varying sized memory chunks on

to the backing store. When some code or date residing in main memory need to be swapped out, space must be found on backing store.

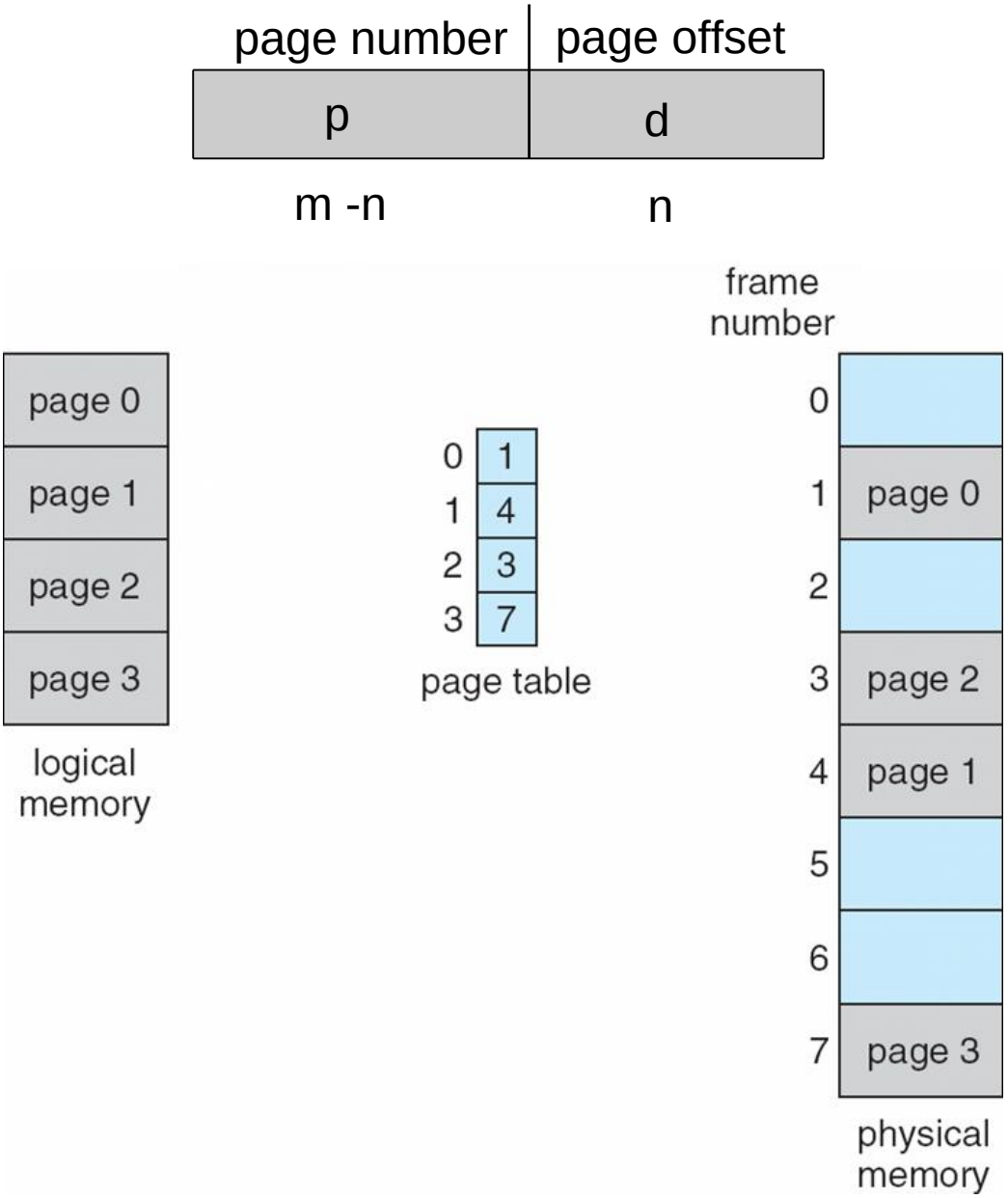
Basic Method:

- Physical memory is broken in to fixed sized blocks called frames (f). Logical memory is broken in to blocks of same size called pages (p).
- When a process is to be executed its pages are loaded in to available frames from backing store.
- The blocking store is also divided in to fixed-sized blocks of same size as memory frames.
- The following figure shows paging hardware:

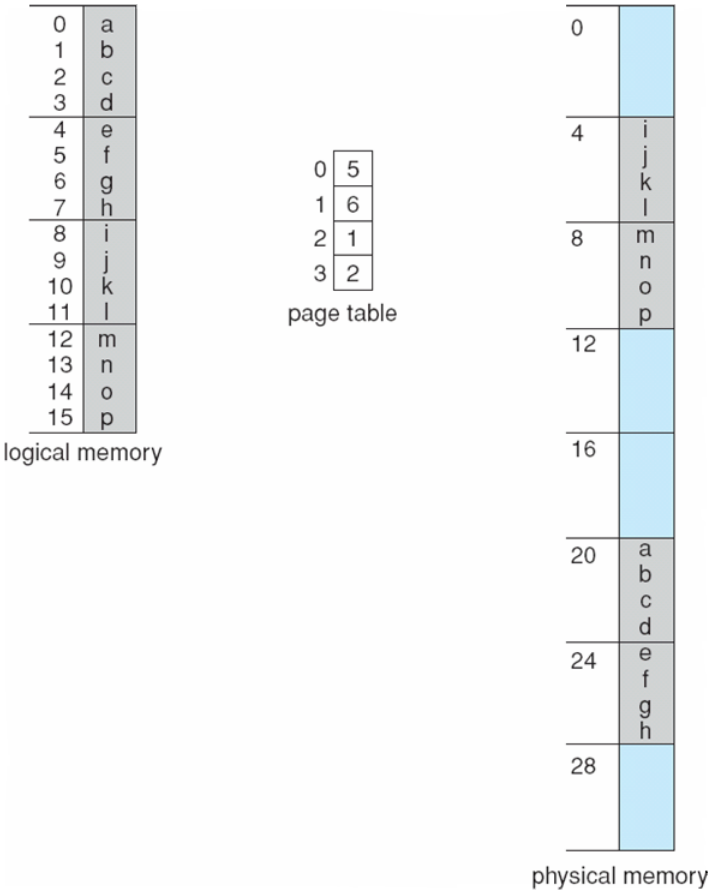


- Logical address generated by the CPU is divided in to two parts: page number (p) and page offset (d).
- The page number (p) is used as index to the page table. The page table contains base address of each page in physical memory. This base address is combined with the page offset to define the physical memory i.e., sent to the memory unit.
- x

- The page size is defined by the hardware. The size of a power of 2, varying between 512 bytes and 10Mb per page.
- If the size of logical address space is 2^m address unit and page size is 2^n, then high order m-n designates the page number and n low order bits represents page offset.



- **Eg:-To show how to map logical memory in to physical memory consider a page size of 4 bytes and physical memory of 32 bytes (8 pages).**



a) Logical address 0 is page 0 and offset 0. Page 0 is in frame 5. The logical address 0 maps to physical address 20. [(5*4) + 0].

b) Logical address 3 is page 0 and offset 3 maps to physical address 23 [(5*4) + 3].

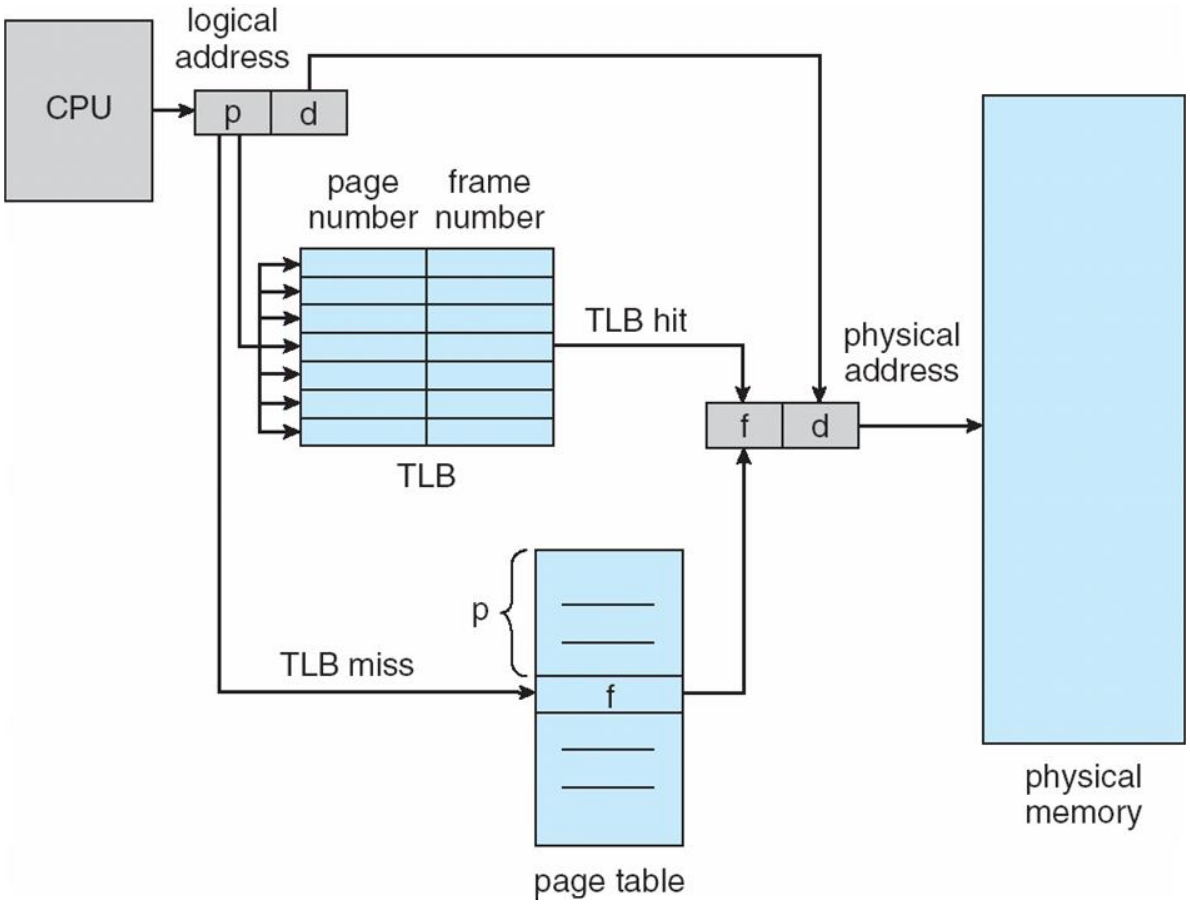
c) Logical address 4 is page 1 and offset 0 and page 1 is mapped to frame 6. So logical address 4 maps to physical address 24 [(6*4) + 0].

d) Logical address 13 is page 3 and offset 1 and page 3 is mapped to frame 2. So logical address 13 maps to physical address 9 [(2*4) + 1].

Hardware Support for Paging:

- The hardware implementation of the page table can be done in several ways:
- The simplest method is that the page table is implemented as a set of dedicated registers. These registers must be built with very high speed logic for making paging address translation.

- Every accessed memory must go through paging map. The use of registers for page table is satisfactory if the page table is small.
- If the page table is large then the use of registers is not visible. So the page table is kept in the main memory and a page table base register [PTBR] points to the page table. Changing the page table requires only one register which reduces the context switching type.
- The problem with this approach is the time required to access memory location. To access a location [i] first we have to index the page table using PTBR offset. It gives the frame number which is combined with the page offset to produce the actual address. Thus we need two memory accesses for a byte.
- The only solution is to use special, fast, lookup hardware cache called **translation look aside buffer [TLB] or associative register.**
- TLB is built with associative register with high speed memory. Each register contains two paths a key and a value.



- When an associative register is presented with an item, it is compared with all the key values, if found the corresponding value field is return and searching is fast.

TLB is used with the page table as follows: TLB contains only few page table entries.

- When a logical address is generated by the CPU, its page number along with the frame number is added to TLB.

- If the page number is found its frame memory is used to access the actual memory.

- If the page number is not in the TLB (TLB miss) the memory reference to the page table is made. When the frame number is obtained use can use it to access the memory.

- If the TLB is full of entries the OS must select anyone for replacement.

- Each time a new page table is selected the TLB must be flushed [erased] to ensure that next executing process do not use wrong information.

- The percentage of time that a page number is found in the TLB is called HIT ratio.

Protection:

- Memory protection in paged environment is done by protection bits that are associated with each frame these bits are kept in page table.

- One bit can define a page to be read-write or read-only.

- To find the correct frame number every reference to the memory should go through page table. At the same time physical address is computed.

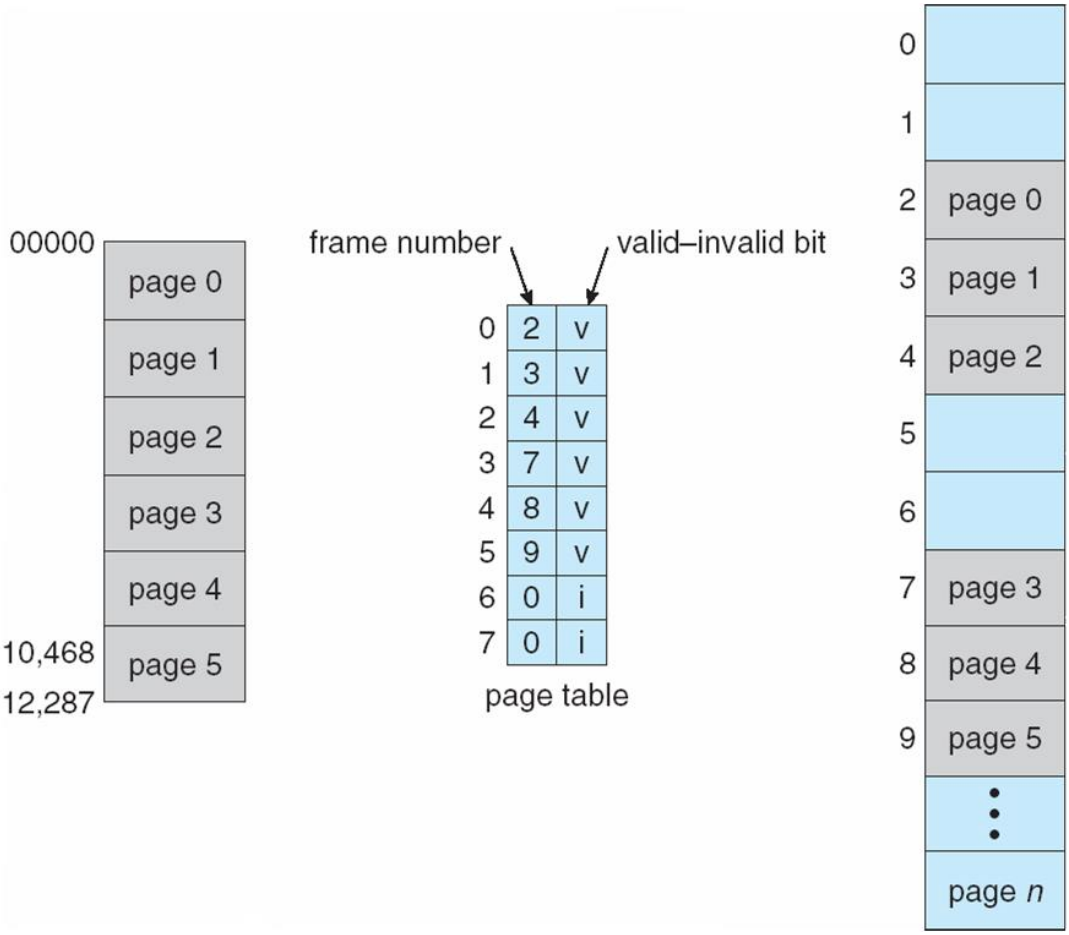
- The protection bits can be checked to verify that no writers are made to read-only page.

- Any attempt to write in to read-only page causes a hardware trap to the OS.

- This approach can be used to provide protection to read-only, read-write or execute-only pages.

    - One more bit is generally added to each entry in the page table: a valid-invalid bit.

- A valid bit indicates that associated page is in the processes logical address space and thus it is a legal or valid page.
- If the bit is invalid, it indicates the page is not in the processes logical addressed space and illegal. Illegal addresses are trapped by using the valid-invalid bit.
- The OS sets this bit for each page to allow or disallow accesses to that page.

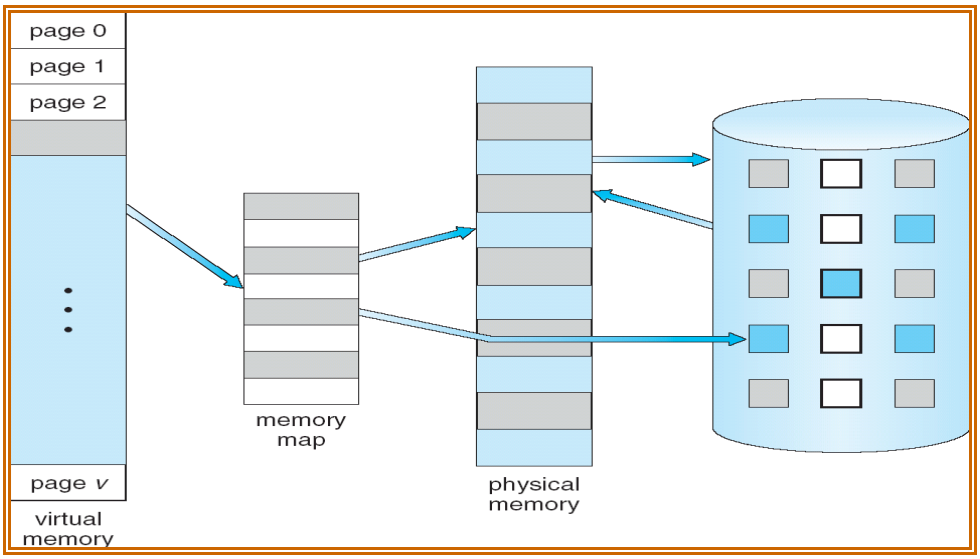


Chapter 9: Virtual memory

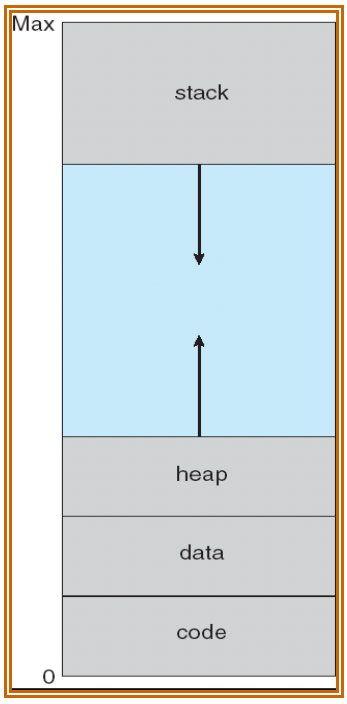
- Background
- Demand Paging
- Copy-on-write
- Page Replacement
- Allocation of Frames
- Thrashing

9.1 Background

- Virtual memory is a technique that allows the execution of processes that may not be completely in memory.
- Only part of the program needs to be in memory for execution.
- Logical address space can therefore be much larger than physical address space.
- Virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.
- Benefits of Virtual memory are:
    - It simplifies the programming task without worrying about the size of physical memory.
    - Program can be of any large size.
    - It provides high Degree of multiprogramming
    - It allows processes to easily share files and address spaces
    - It provides an efficient mechanism for process creation.
    - Due to less I/O, execution of program will be faster.

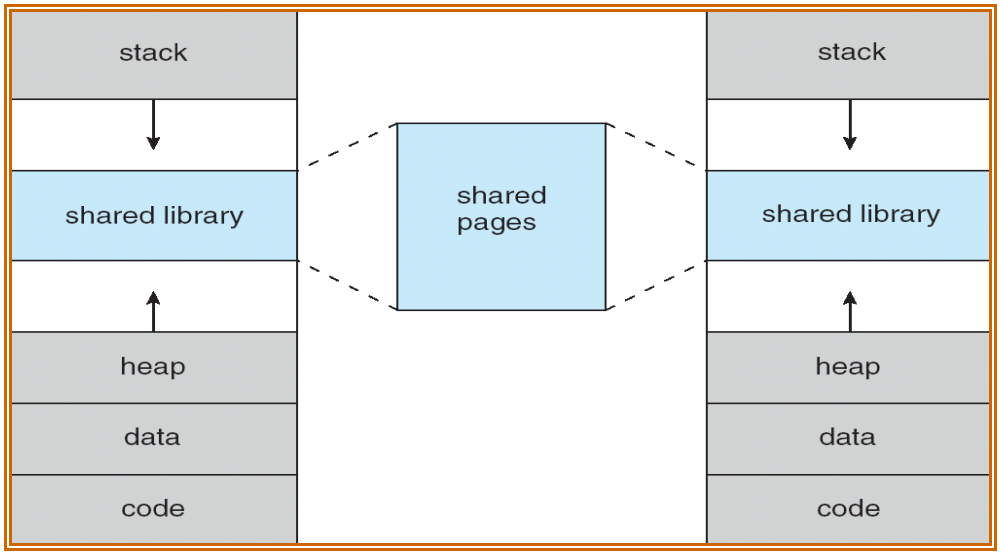


- Virtual address space of a process refers to the logical view of how a process is stored in the memory.
- Physical memory may be organized in page frames and the physical page frames assigned to a process may not be contiguous.
- MMU maps logical address to physical address.



- Heap and stack are allowed to grow thus a large blank space (sparse address space) is also allocated.

- Using sparse address space is beneficial because
  - The holes can be filled as the stack or heap segments grow
  - Dynamic linking of the libraries is possible
  - Shared objects can be linked
  - Processes can communicate through shared memory



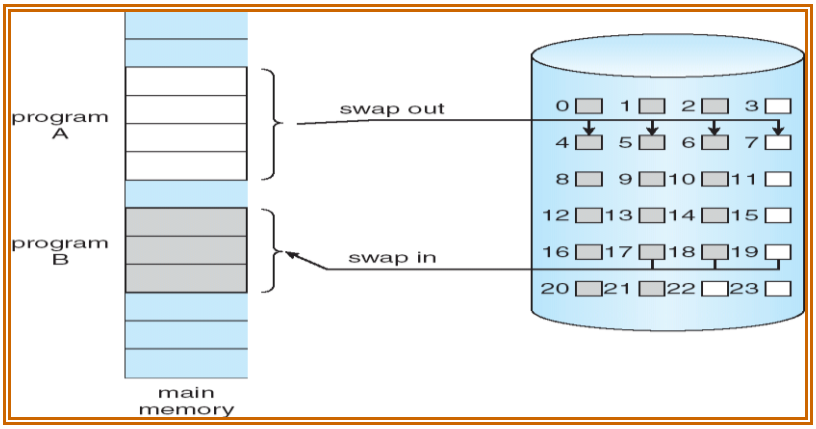
- Virtual memory can be used through the technique
  - Demand paging

9.2 Demand Paging

- Demand paging technique avoids reading pages into memory that will not be used currently.
- Demand paging is commonly used in virtual memory systems.
- Initially only those pages are loaded which are needed.
- Thus it brings a page into memory only when it is needed.
- Advantages:
  - Less memory needed
  - More users

- Demand paging is similar to paging system with swapping
- Processes reside on secondary memory.
- Whenever a process is submitted for execution, the page needed will be swapped in physical memory (lazy swapper)
- A lazy swapper never swaps a page into memory unless that page will be needed
- Pager is concerned with the individual pages for swapping thus reducing swap time .
- **"Page-in" operation**
  - **When a page is required, pager finds a free frame in memory and load the page in memory.**
  - **"Page-out" operation**
  - **If no free frame is available, one of the page in memory is written out onto secondary memory to make a frame free.**

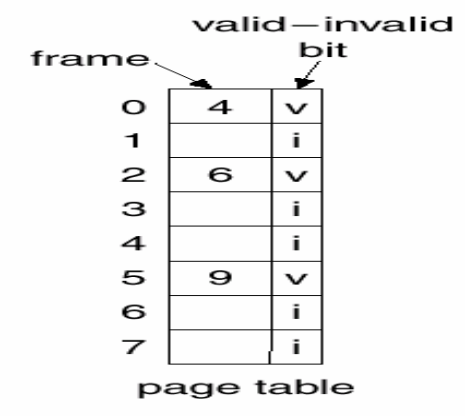
Transfer of a Paged Memory to Contiguous Disk Space (Swapping)

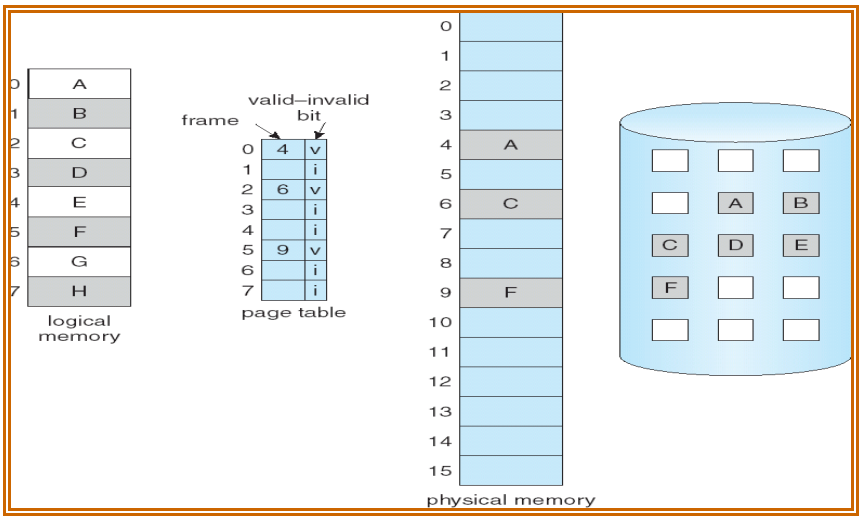


Hardware Support (Valid-Invalid Bit)

- **With each page table entry a valid–invalid bit is associated (1 ⇒ valid, 0 ⇒ invalid)**
- **Initially valid–invalid bit is set to 0 on all entries.**
- **When this bit is set to "valid," this value indicates that the associated page is both legal and in memory.**

- **If the bit is set to "invalid", this value indicates that the page either is not valid (that is, not in the logical address space of the process), or is valid but is currently on the disk.**





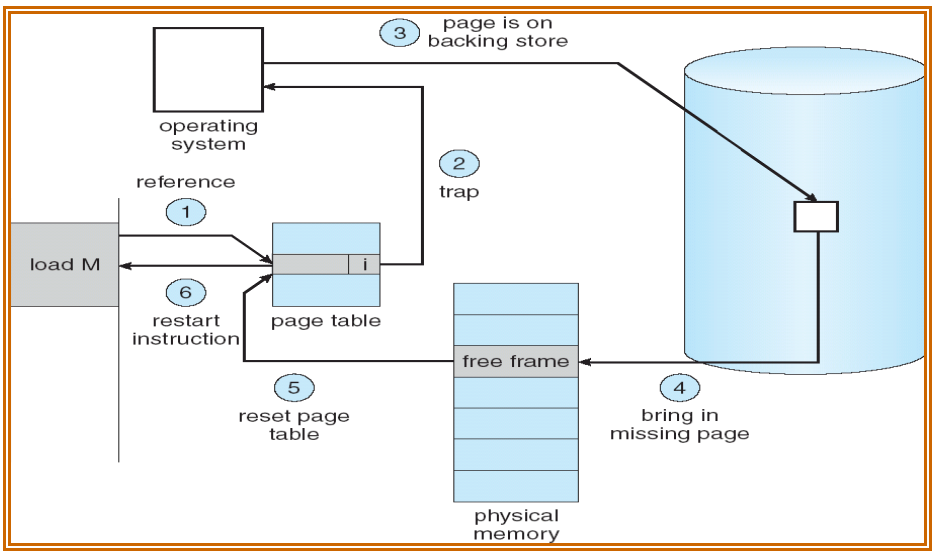
Page Fault

If there is a reference to a page:

- **Reference will be checked in page table for the bit being valid or invalid**
- **If bit is valid, page exists in memory.**
- **While the process executes and accesses pages that are memory resident, execution proceeds normally**
- **If page is not in memory:**
  - **invalid reference ⇒ abort**
  - **not-in-memory ⇒ bring to memory**

- **Reference can be checked whether valid or invalid:**
  - **An internal table (usually kept with the process control block) for this process is checked to determine whether the reference was a valid or invalid page in logical address space.**
- **If page is not in memory but valid logical address:**
  - **Get empty frame**
  - **Swap desired page into frame**
  - **Reset tables, validation bit = 1**
  - **If empty frame is not available, a page which is not in use, can be swapped out to secondary memory i.e. Page Replacement**

Steps in Handling a Page Fault



Pure Demand Paging

- **In the extreme case, we could start executing a process with no pages in memory.**
- **When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page.**
- **After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory.**
- **At some point, it can execute with no more faults.**

- **This scheme is pure demand paging: Never bring a page into memory until it is required.**

Demand Paging

- **Some programs may access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction.**
- **This situation would result in unacceptable system performance.**
- **Analysis of running processes shows that this behavior is exceedingly unlikely.**
- **Programs tend to have locality of reference which results in reasonable performance from demand paging.**
- **The hardware to support demand paging is the same as the hardware for paging and swapping.**
- **Page table**
    - **This table has the ability to mark an entry invalid through a valid-invalid bit or special value of protection bits.**
- **Secondary memory**
    - **This memory holds those pages that are not present in main memory.**
    - **The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as swap space.**

Performance of Demand Paging

- **Demand paging can have a significant effect on the performance of a computer system**
- **Memory access (ma): 10 – 200 nanoseconds**
- **If no page faults occurs, the effective access time is equal to the memory access time.**
- **If a page fault occurs, we must first read the relevant page from disk, and then access the desired word.**
- **Probability of Page Fault $0 \leq p \leq 1.0$**

- **if p = 0 no page faults**
- **if p = 1, every reference is a fault**

- **Effective Access Time (EAT)**

EAT = (1 – p)*memory access + p*page fault time

- **To compute the effective access time, we must know how much time is needed to service a page fault.**
- **If an average page-fault service time is 8 milliseconds and a memory-access time is 200 nanoseconds, then the effective access time in nanoseconds is**

effective access time = (1 - p) x (200) + p (8 milliseconds)

= (1 - p) x 200 + p x 8,000,000

= 200 + 7,999,800 x p nanoseconds

- **Effective access time is directly proportional to the page-fault rate.**
- **If one access out of 1,000 causes a page fault, the effective access time is 8.2 microseconds.**
- **The computer would be slowed down by a factor of 40 because of demand paging!**
- **If we want less than 10-percent degradation, we need**

$$220 > 200 + 7{,}999{,}800 * p$$

$$20 > 7{,}999{,}800 * p$$

$$p < 0.0000025$$

i.e. 25 faults in 10 millions

- **That is, to keep the slowdown due to paging to a reasonable level, we can allow only less than one memory access out of 399,990 to page fault.**
- **A page fault causes the following sequence to occur:**

1. Trap to the operating system.

2. Save the user registers and process state.

3. Determine that the interrupt was a page fault.

4. Check that the page reference was legal and determine the location of the page on the disk.

5. Issue a read from the disk to a free frame:

a. Wait in a queue for this device until the read request is serviced.

b. Wait for the device seek and/or latency time.

c. Begin the transfer of the page to a free frame.

6. While waiting, allocate the CPU to some other user (CPU scheduling; optional).

7. Interrupt from the disk (I/O completed).

8. Save the registers and process state for the other user (if step 6 is executed).

9. Determine that the interrupt was from the disk.

10. Correct the page table and other tables to show that the desired page is now in memory.

11. Wait for the CPU to be allocated to this process again.

12. Restore the user registers, process state, and new page table, then resume the interrupted instruction.

- **Not all of these steps are necessary in every case**
- **In any case, we are faced with three major components of the page-fault service time:**

1. Service the page-fault interrupt.

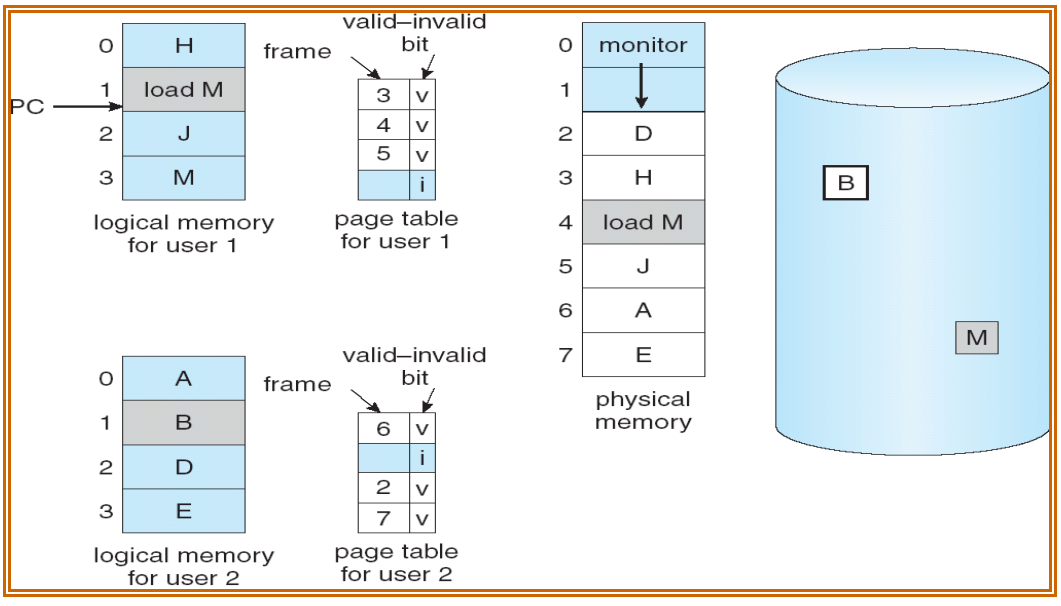
2. Read in the page.

3. Restart the process.

- **One additional aspect of demand paging is the handling and overall use of swap space.**
- **Disk I/O to swap space is generally faster than that to the file system.**
- **It is faster because swap space is allocated in much larger blocks.**

- **It is therefore possible for the system to gain better paging throughput, by copying an entire file image into the swap space at process startup, and then performing demand paging from the swap space.**
- **Another option is to demand pages from the file system initially, but to write the pages to swap space as they are replaced.**
- **This approach will ensure that only needed pages are ever read from the file system, but all subsequent paging is done from swap space.**
- **Some systems limits the amount of swap space.**
- **Demand pages are brought directly from the file system.**
- **However, when page replacement is called for, these pages can simply be overwritten and read in from the file system again if needed.**
- **The file system itself serves as the backing store.**
- **However, swap space must still be used for pages not associated with a file; these pages include the stack and heap for a process.**
- **Example: Solaris 2.**

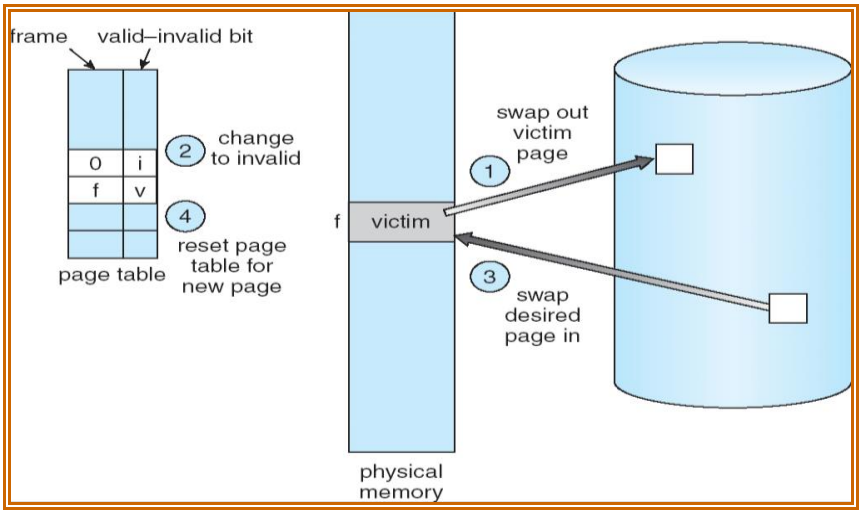
9.3 Need For Page Replacement



Basic Page Replacement

- **Find the location of the desired page on disk.**
- **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* page to make frame free.**
- **Read the desired page into the (existing/newly) free frame. Update the page and frame tables.**
- **Restart the process.**

Page Replacement



What happens if there is no free frame?

- **Page replacement is to find some page (not really in use) in memory to page out.**
  - **Various algorithm are possible to handle page replacement.**
  - **Performance of an algorithm should result in minimum number of page faults.**
- **Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk**

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**

Page Replacement Algorithms

- **Page replacement algorithm should have lowest page-fault rate.**
- **Algorithm can be evaluated by running it on a particular string of memory references (reference string) and computed the number of page faults on that string.**

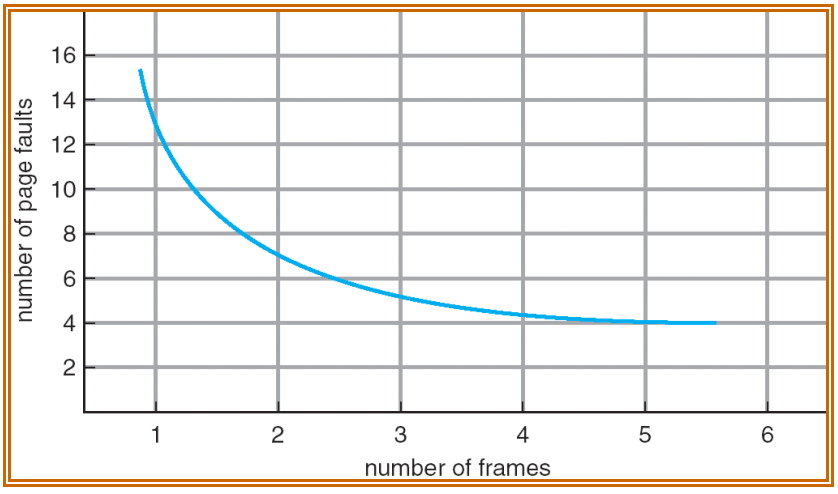
0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105.

- **If page size is 100 bytes**
- **Find total page faults for the reference string**

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

i) If one free frame is available

ii) If two, three…. free frames are available

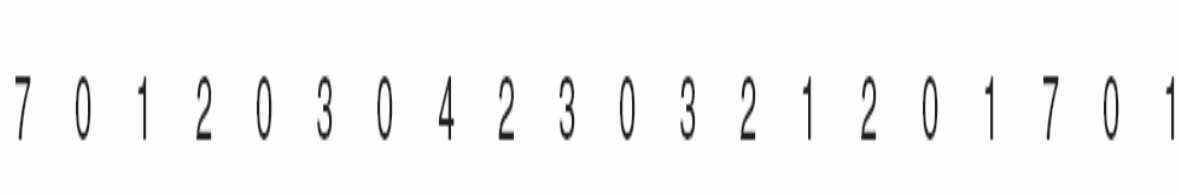


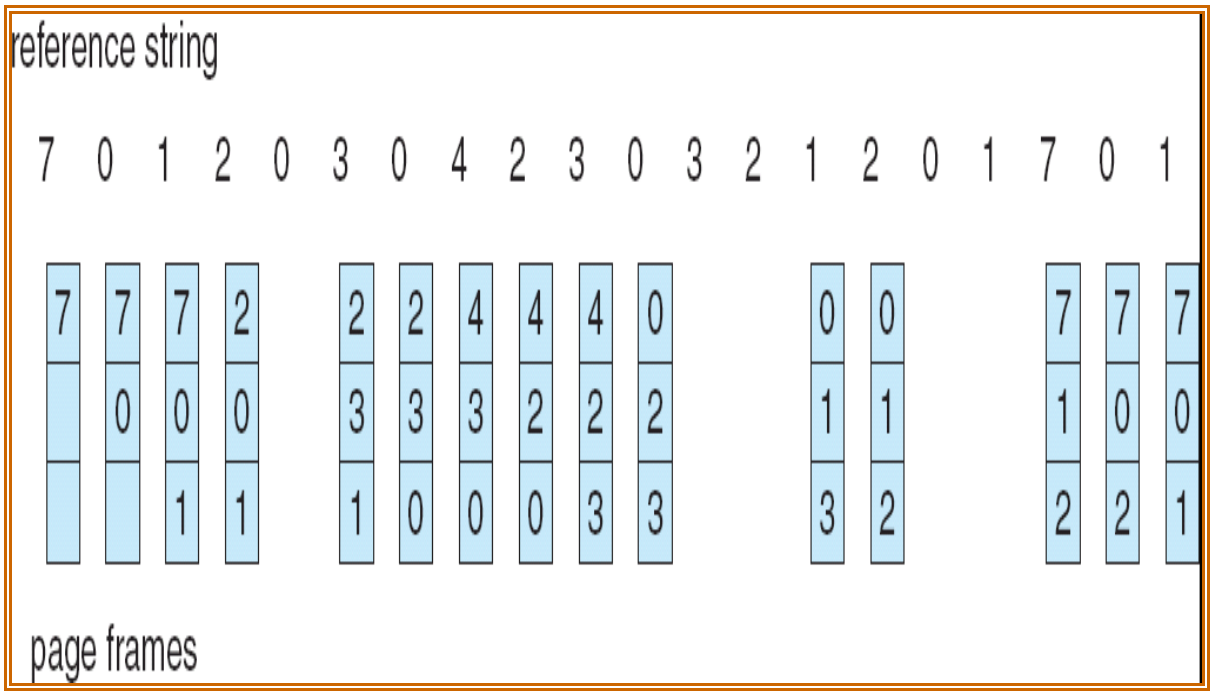
- **FIFO Page Replacement**
- **Optimal Page Replacement**
- **LRU Page Replacement**
- **LRU Approximation Page Replacement**
    - **Additional-Reference-Bits Algorithm**
    - **Second-Chance Algorithm**
    - **Enhanced Second-Chance Algorithm**
- **Counting-Based Page Replacement**
- **Page-Buffering Algorithm**

First-In-First-Out (FIFO) Replacement Algorithm

- **When a page must be replaced, the oldest page is chosen (which came first in memory).**
- **A FIFO queue of the pages present in memory is maintained.**
- **A page is replaced from the head of the queue.**
- **When a page is brought into the memory, it is inserted at the tail of the queue.**

Find the page faults for following string:





Page faults for next example = 15

- **Reference string:**

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- **3 frames (3 pages can be in memory at a time per process)**
- **4 frames**
- **FIFO Replacement suffers from Belady's Anomaly**
  - **more frames ⇒ more page faults (sometimes)**

- **Reference string:**

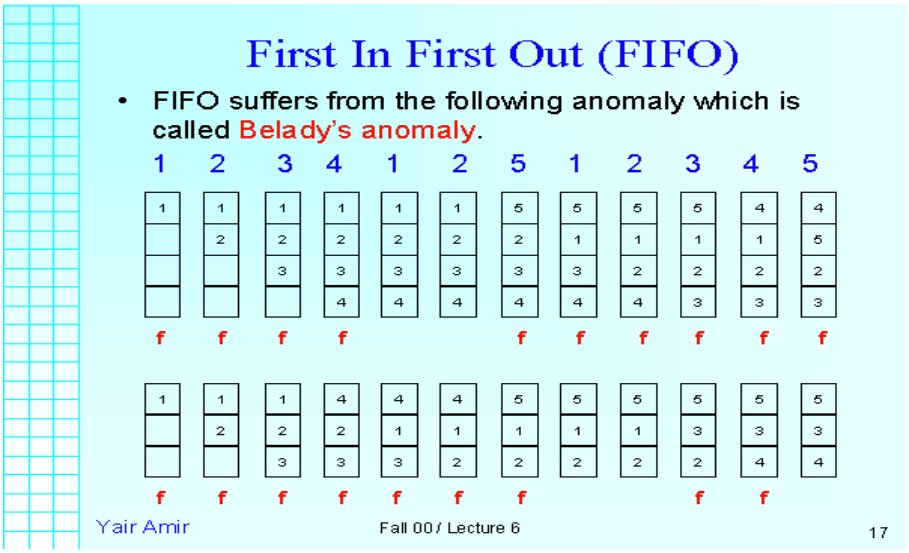
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- **3 frames (3 pages can be in memory at a time per process)**
- **4 frames**

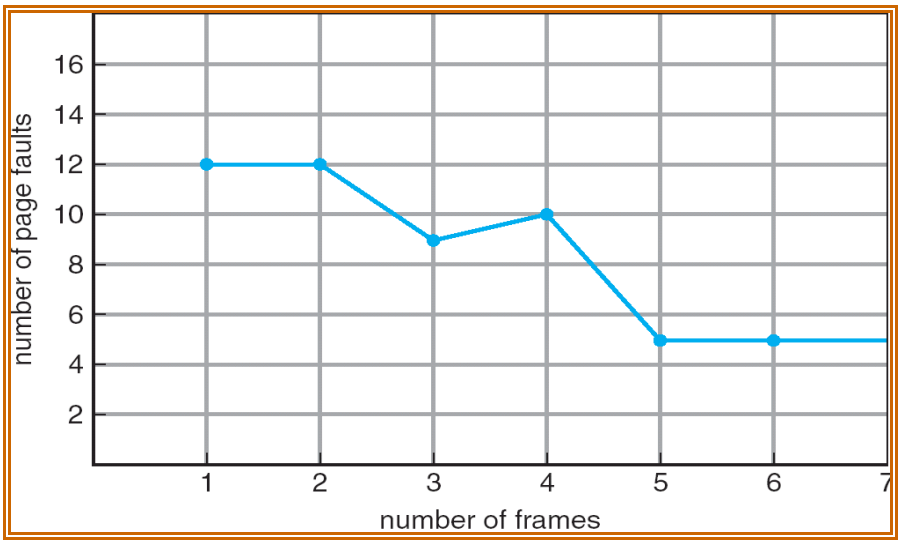
- **FIFO Replacement suffers from Belady's Anomaly**
  - **more frames ⇒ more page faults (sometimes)**

1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		



FIFO Illustrating Belady's Anomaly

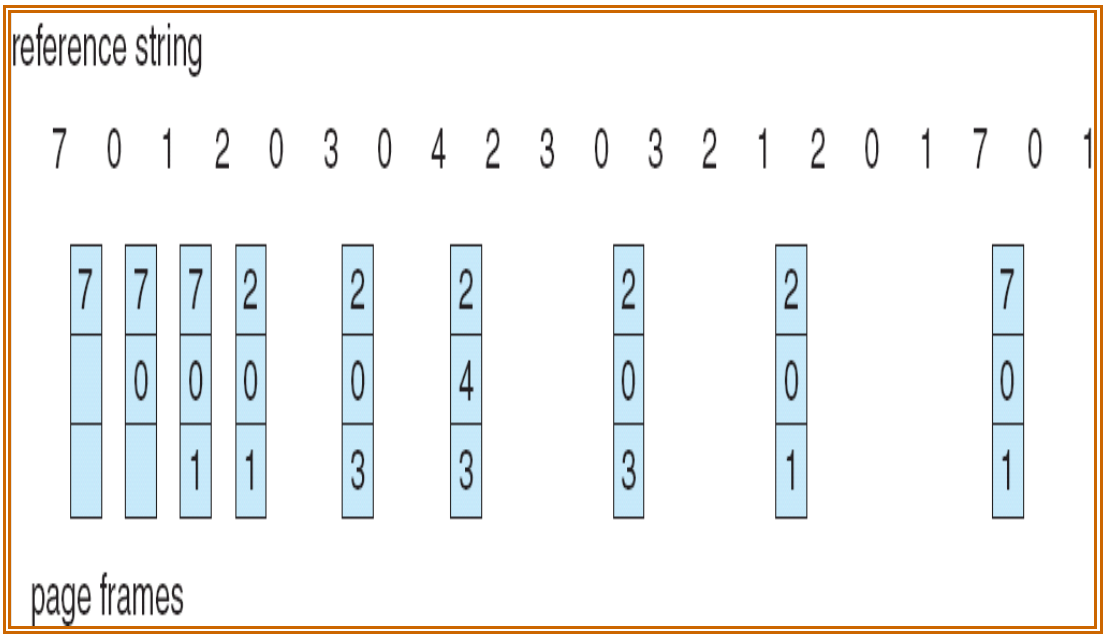


Optimal Page Replacement Algorithm

- **Page that will not be used for longest period of time is replaced.**
- **This algorithm guarantees the lowest possible page-fault rate.**

- **Find the page faults for following string:**

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



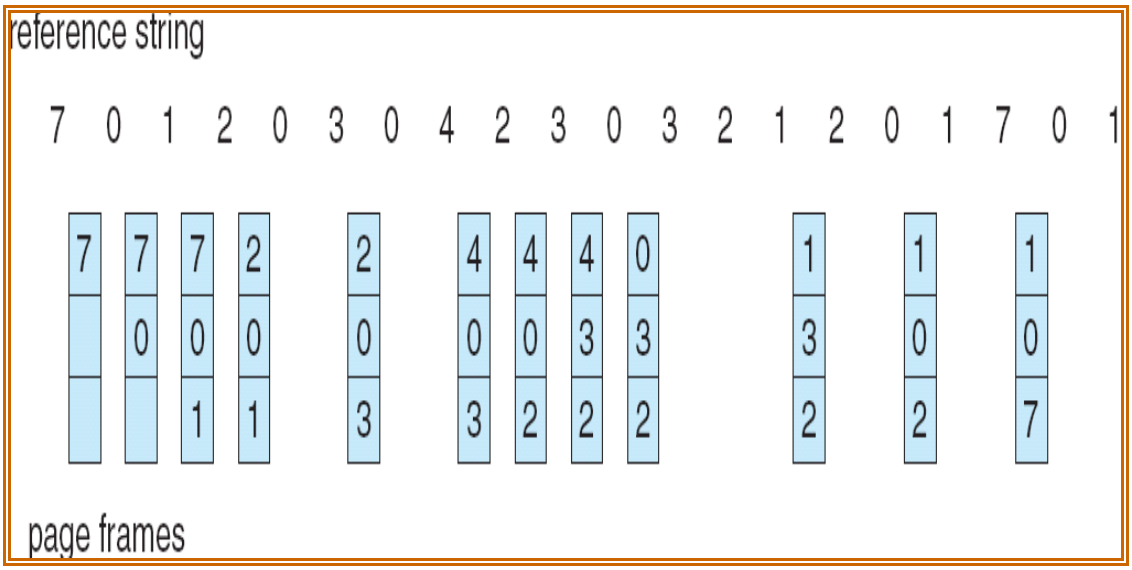
Page faults for next example = 9

Least Recently Used (LRU) Algorithm

- **When a page must be replaced, LRU chooses the page that has not been used for a longest period of time.**
- **The LRU policy is often used as a page-replacement algorithm and is considered to be good.**
-

Find the page faults for following string:

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



Page faults for next example = 12

LRU Algorithm: Implementation

- **An LRU page-replacement algorithm may require substantial hardware assistance.**
- **The problem is to determine an order for the frames defined by the time of last use.**
- **Two implementations are**
  - **Counter**
  - **Stack implementation**

LRU Algorithm: Counter Implementation

- **Each page-table entry has a 'time-of-use' field**
- **A logical clock (counter) is added to the CPU.**
- **The clock is incremented for every memory reference.**
- **Whenever a reference to a page is made, the contents of the clock register are copied to the 'time-of-use' field in the page-table entry for that page.**
- **In this way, the "time" of the last reference to each page can be maintained.**
- **The page with the smallest time value is replaced.**

- **This scheme requires a search of the page table to find the LRU page, and a write to memory (to the 'time-of-use' field in the page table) for each memory access.**
- **The times must also be maintained when page tables are changed (due to CPU scheduling).**
- **Overflow of the clock must be considered.**

LRU Algorithm: Stack Implementation

- **A stack of page numbers can be maintained**
- **Whenever a page is referenced, it is removed from the stack and put on the top.**
- **In this way, the top of the stack is always the most recently used page and the bottom is the LRU page.**
- **Because entries must be removed from the middle of the stack, it is best implemented by a doubly linked list, with a head and tail pointer.**
- **Removing a page and putting it on the top of the stack requires changing six pointers at worst.**
- **Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page.**
- **Neither optimal replacement nor LRU replacement suffers from Belady's anomaly.**