MEMORY MANAGEMENT

Basic Concepts

- ✓ Every program to be executed has to be executed must be in memory. The instruction must be fetched from memory before it is executed.
- ✓ The program and data must be bought into the memory from the disk, for the process to run. Each process has a separate memory space and must access only this range of legal addresses.
- ✓ Protection of memory is required to ensure correct operation. This prevention is provided by hardware implementation.
- ✓ Two registers are used - a **base register** and a **limit register**.
- ✓ The **base register** holds the smallest legal physical memory address
- ✓ The **limit register** specifies the size of the range.

Logical versus Physical Address Space

- ✓ The address generated by the CPU is a ***logical address***
- ✓ An address seen by the memory unit-i.e., the one loaded into the memory address register of the memory is a ***physical address***. (OR) The memory address where programs are stored is a ***physical address***.
- ✓ The compile-time and load-time address-binding methods generate identical logical and physical addresses
- ✓ But execution-time address-binding scheme results in different logical and physical addresses.
- ✓ The set of all logical addresses used by a program composes the ***logical address space***, and the set of all physical addresses corresponding to these logical addresses is the ***physical address space.***
- ✓ The run-time mapping of logical (virtual) to physical addresses is handled by the ***memory-management unit (MMU)***
- ✓ The MMU can take on many forms. One of the simplest is a modification of the base-register scheme described earlier.
- ✓ The base register is now termed a ***relocation register***, whose value is *added* to every memory request at the hardware level as shown in figure 1

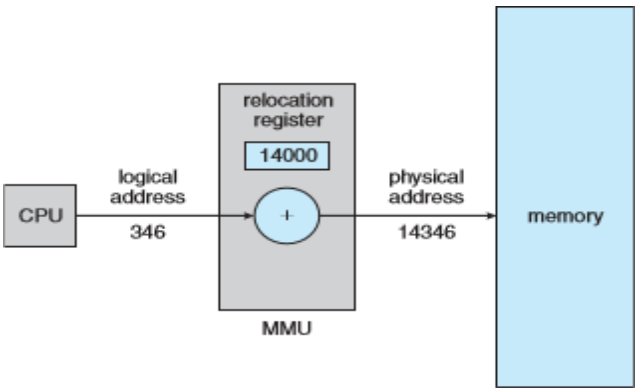

Figure 1: Dynamic relocation using a relocation register

4.2 Swapping

- ✓ A process must be loaded into memory to execute.
- ✓ If there is not enough memory available to keep all running processes in memory at the same time, then some processes that are not currently using the CPU may have their memory swapped out to a fast local disk called the ***backing store.***
- ✓ Swapping is **the process of moving a process from memory to the backing store and moving another process from the backing store to memory**. Swapping is a very slow process compared to other operations as shown in figure 2

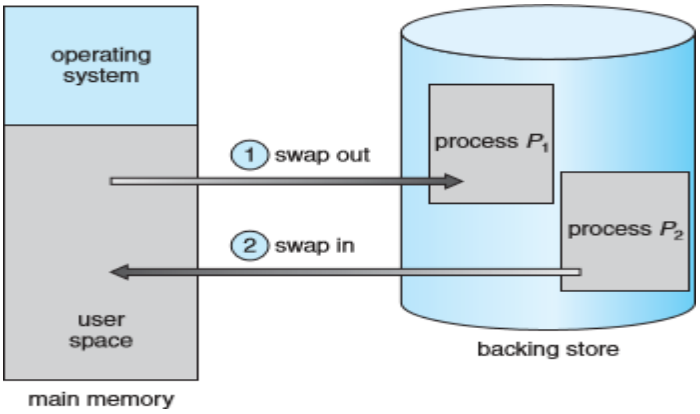

Figure 2: Swapping of two processes using a disk as a backing store

- ✓ Swapping is sometimes called ***roll out, roll in***
- ✓ A process that is swapped out will be swapped back into the same memory space it occupied previously
- ✓ If binding is done at ***assembly or load time***, then the process cannot be easily moved to a different location
- ✓ If ***execution-time binding*** is used, then a process can be swapped into a different memory space.
- ✓ Swapping requires a backing store. The backing store is commonly a fast disk

- ✓ The system maintains a ***ready queue*** consisting of all processes whose memory images are on the backing store or in memory and are ready to run
- ✓ Whenever the CPU scheduler decides to execute a process, it calls the ***dispatcher.***
- ✓ The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.
- ✓ **Constraints of Swapping**:
  - It is important to swap processes out of memory only when they are idle, or more to the point, only when there are no pending I/O operations. (Otherwise, the pending I/O operation could write into the wrong process's memory space.)
  - **The solution:**
  - Swap only idle processes
  - Execute I/O operations only into OS buffers.

4.3 Contiguous Memory Allocation

- ✓ The main memory must accommodate both the OS and the various user processes
- ✓ The memory is divided into two partitions: OS and the user processes
- ✓ The operating system is allocated first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed (The OS is usually loaded low because that is where the interrupt vectors are located)
- ✓ In ***contiguous memory allocation,*** each process is contained in a single contiguous section of memory

4.3.1 Memory Mapping and Protection

- ✓ Protection can be provided to user processes with relocation (base register) and limit register
- ✓ The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses
- ✓ With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address *dynamically* by adding the value in the relocation register. This mapped address is shown in figure 3

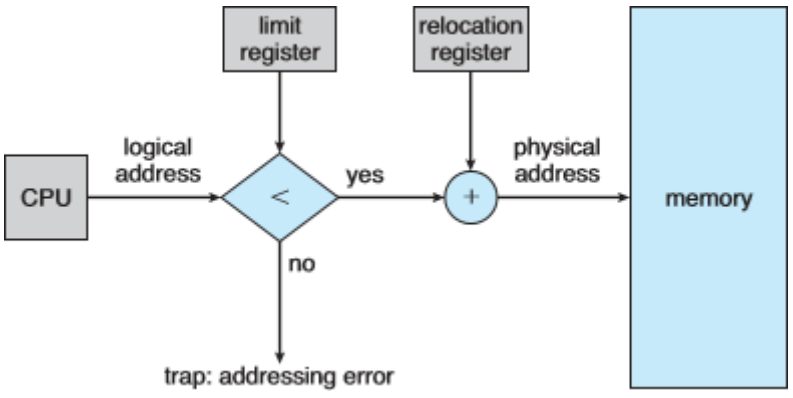

Figure 3: Hardware support for relocation and limit registers

- ✓ When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch

4.3.2 Memory Allocation

- ✓ One of the simplest methods for allocating memory is to divide memory into severalfixed-sized ***partitions***
- ✓ Each partition may contain exactly one process
- ✓ Partitioning can be either fixed or variable
- ✓ In **Multiprogramming with the fixed task (MFT**): The number of partitions are fixed, butthe size is different.
  - In this method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process
- ✓ In **Multiprogramming with Variable task (MVT):** The OS keeps a table indicating which parts of memory are available and which are occupied.
  - Initially, all memory is available for user processes and is considered one large block of available memory called a ***hole.*** (free space in memory)
- ✓ As processes enter the system, they are put into an input queue. The OS takes into account the *memory requirements* of each process and the *amount of available memory space* in determining which processes are allocated memory
- ✓ When a process is allocated apace, it is loaded into memory, and it can then compete for CPU.
- ✓ When a process terminates, it releases its memory, which OS may then fill with another process from the input queue
- ✓ In general, the memory blocks available comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.
- ✓ If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes
- ✓ When a process terminates, it releases its block of memory, which is then placed back in the set of holes.
- ✓ If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole
- ✓ There are many different strategies for finding the "best" allocation of memory to processes, including the three most commonly discussed:
  - **First fit** - Search the list of holes until one is found that is **big enough** to satisfy the request, and assign a portion of that hole to that process. Whatever fraction of the hole is not needed by the request is left on the free list as a smaller hole. Subsequent requests may start looking either from the beginning of the list or from the point at which this search ended.
  - **Best fit** - Allocate the ***smallest*** **hole that is big enough** to satisfy the request. This saves large holes for other process requests that may need them later, but the resulting unused portions of holes may be too small to be of any use and will

therefore be wasted. Keeping the free list sorted can speed up the process of finding the right hole.

- **Worst fit** - Allocate the **largest hole** available, thereby increasing the likelihood that the remaining portion will be used for satisfying future requests.

✓ Simulations show that both first fit and best fit are better than worst fit in terms of both time and storage utilization

4.3.3 Fragmentation

✓ Memory fragmentation can be either **Internal** or **External**
✓ Both the first-fit and best-fit strategies suffer from ***External Fragmentation***
✓ **External fragmentation** exists when there is enough total memory space to satisfy a request but the available spaces are ***not contiguous***; storage is fragmented into a large number of small holes
✓ Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem
✓ With **Internal Fragmentation**, the memory allocated to a process may be slightly ***larger*** then the requested memory.
✓ Solution to External Fragmentation:

- If the programs in memory are relocatable, (using execution-time address binding) then the **external fragmentation** problem can be reduced via ***compaction***, i.e. moving all processes down to one end of physical memory so as to place all free memory together to get a large free block. This only involves updating the relocation register for each process, as all internal work is done using logical addresses
- Another solution to external fragmentation is to allow processes to use non-contiguous blocks of physical memory- **Paging** and **Segmentation**.

4.4 Paging

✓ **Paging** is a memory-management scheme that allows the physical address space of a process to be ***non-contiguous***; thus, allowing a process to be allocated physical memory wherever memory is available
✓ Paging avoids external fragmentation and the need for compaction

4.4.1 Basic Method

✓ The basic idea behind paging is to divide physical memory into a number of equal sized blocks called ***frames***, and to divide a program's logical memory space into blocks of the same size called ***pages***
✓ When a process is to be executed, its pages are loaded into any available memory frames from their source
✓ The paging hardware is illustrated in shown in figure 4

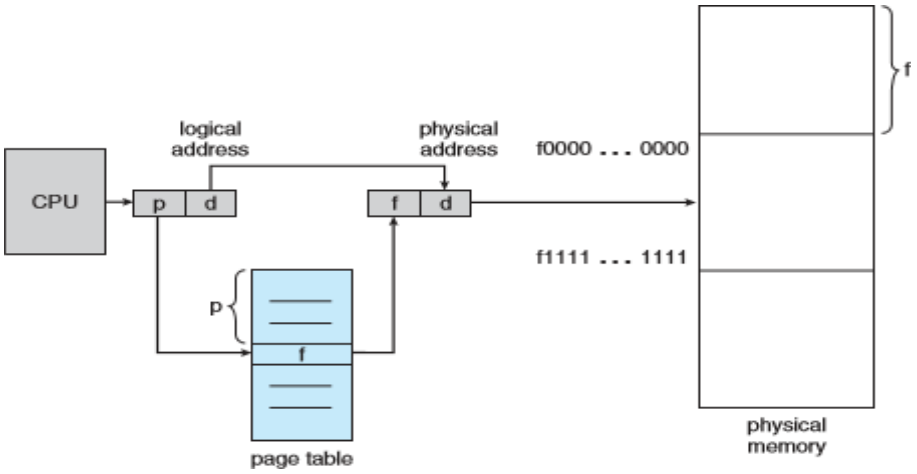

Figure 4: Paging Hardware

- ✓ Every address generated by the CPU is divided into two parts: **a page number(p)** and a **page offset(d)**
- ✓ The page number is used as an index into a **page table.** The page table contains the base address of each page in physical memory address.
- ✓ The paging model of memory is shown in figure 5

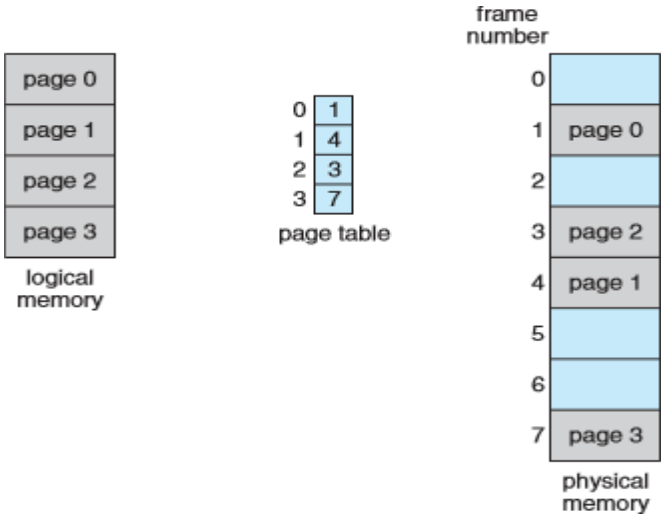

Figure 5: Paging model of logical and physical memory

- ✓ The page size is defined by the hardware.
- ✓ The size of a page is typically a power of 2. If the size of the logical address space is 2^m and the page size is 2^n addressing units, then the high order ***m-n*** bits of a logical address indicate the page number and the ***n*** low-order bits indicate the page offset
- ✓ The logical address is shown in figure 6, Where ***p*** is an index into the page table and ***d*** *is the* offset within the page

Figure 6: Logical address

✓ Consider the memory in figure 7, here logical address *n=2* and *m=4.* Using a page size of 4 bytes and a physical memory of 32 bytes.

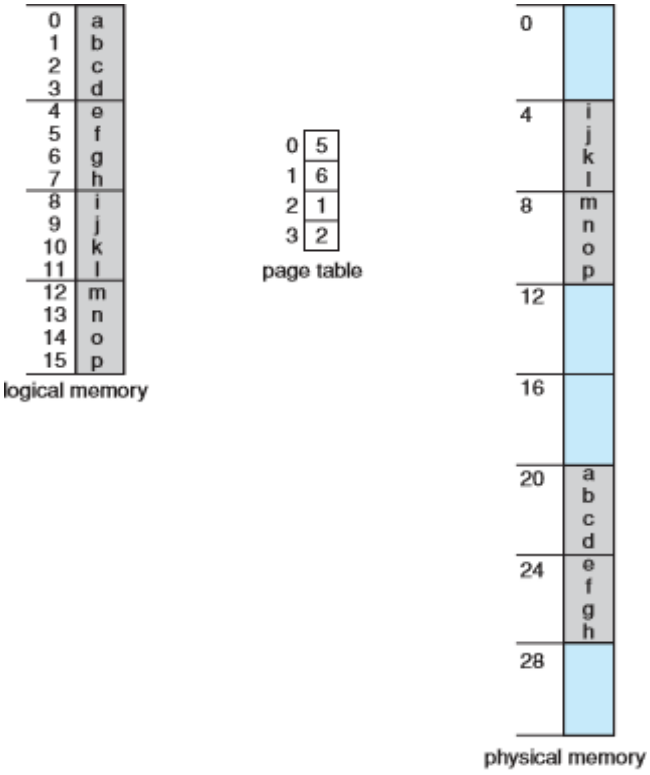

Figure 7: Paging example for a 32-byte memory with 4-byte pages

- Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [=(5*4)+0]
- Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5*4)+3].
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6.

✓ Every logical address is bound by the paging hardware to some physical address
✓ When we use a paging scheme, we have ***no external fragmentation***; any free frame can be allocated to a process that needs it
✓ However, paging suffers from **internal fragmentation**
✓ When a process arrives in the system to be executed, its size, expressed in pages, is examined
✓ Each page of the process needs one frame. Thus, if the process requires *n* pages, at least *n* frames must be available in memory.
✓ If *n* frames are available, they are allocated to this arriving process.
✓ The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process
✓ The next page is loaded into another frame, its frame number is put into the page table, and so on as in figure 8

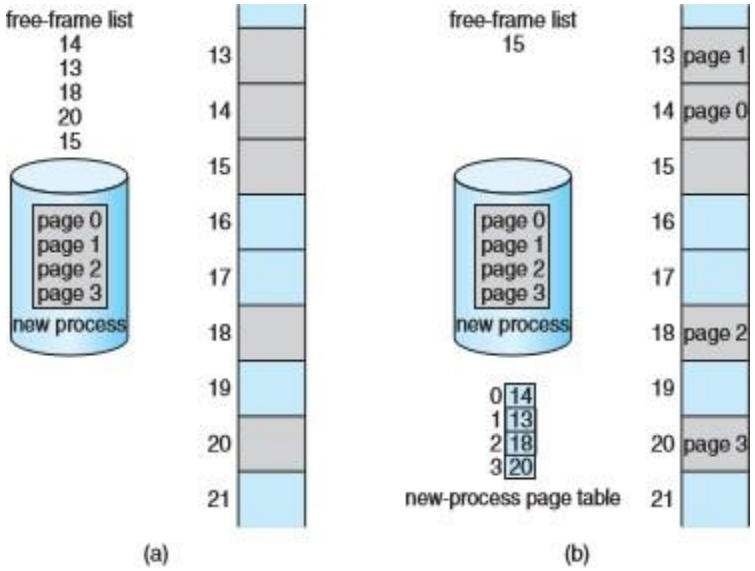

Figure 8: Free frames (a) Before allocation (b) After allocation

- ✓ Paging separates user's view of memory and the actual physical memory
- ✓ The difference between the user's view of memory and the actual physical memory is reconciled by the address-translation hardware.
- ✓ The logical addresses are translated into physical addresses. This mapping is hidden from the user and is controlled by the OS
- ✓ The OS must be aware of the allocation details of physical memory-which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a ***frame table***
- ✓ Paging increases the context switching

4.4.2 Hardware Support

- ✓ The hardware implementation of the page table can be done in several ways
- ✓ The page table is implemented as a set of dedicated registers
- ✓ For most machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory and a ***page-table base register (PTBR)*** points to the page table
- ✓ **Problem:** Time required to access a user memory location. If we want to access location *i*, we must first index into the page table, using the value in the PTBR offset by the page number for *i.* It requires a memory access.
  In this scheme, two memory access are need to access a byte
- ✓ **Solution:** To use a special, small, fast-lookup hardware cache, called a ***Translationlook-aside buffer (TLB)***
- ✓ Each entry in the TLB consists of two parts: a Key(or tag) and a value as shown in figure 9
- ✓ When the associative memory is presented with an item, the item is compared with all keys simultaneously.
  - If the item is found, the corresponding value field is returned

Usage of TLB with page tables

- The TLB contains only a few of the page-table entries
- When a logical address is generated by CPU, its page number is presented to the TLB
- If the page number is found, its frame number is immediately available and is used to access memory
- If the page number is not in the TLB (TLB miss), a memory reference to the page table must be made.
- When the frame number is obtained, we can use it to access memory
- Later, Page number and frame number to the TLB so that they will be found quickly on the next reference

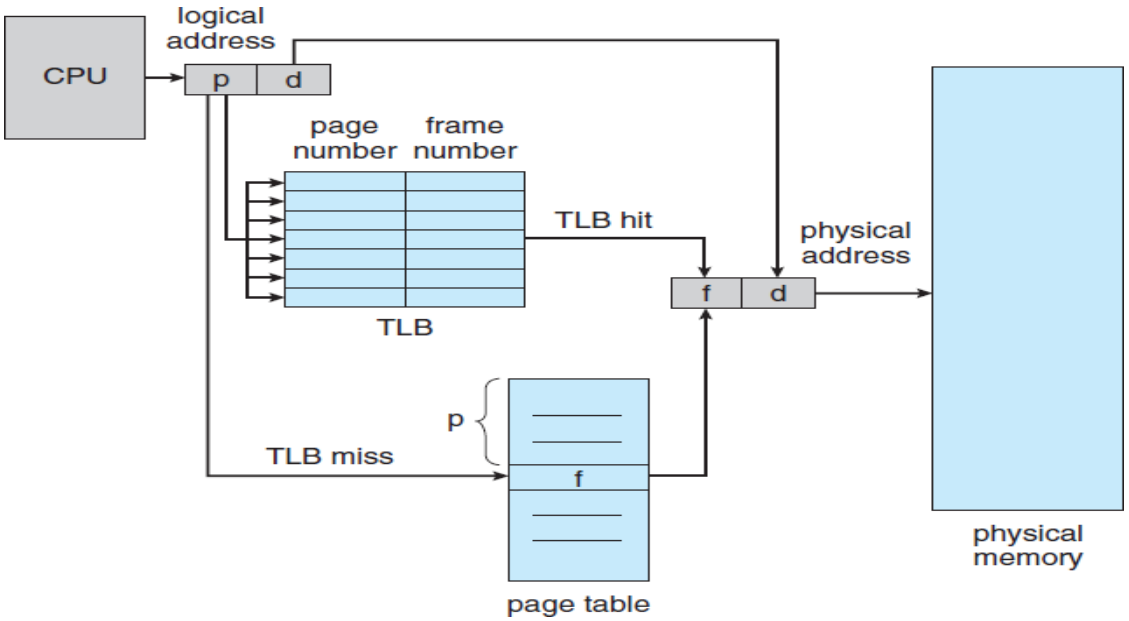

Figure 9: Paging Hardware with TLB

✓ The percentage of times that a particular page number is found in the TLB is called the ***hit ratio***

4.4.3 Protection

✓ Memory protection in a paged environment is accomplished by protection bits associated with each frame. These bits are kept in the page table
✓ The page table can also help to protect processes from accessing memory
✓ A bit is added to the page table: ***Valid-invalid*** bit
✓ When this bit is set to "***valid (V)",*** the associated page is in the process's logical address space and is thus a legal (valid) page
✓ When the bit is set to "***invalid(I)",*** the page is not in the process's logical address space
✓ For example, as shown in figure 10.Addresses in pages 0,1,2,3 and 5 are mapped normally through the page table
✓ Addresses of the pages 6 and 7 are invalid and cannot be mapped. Any attempt to access those pages will send a trap to the OS.

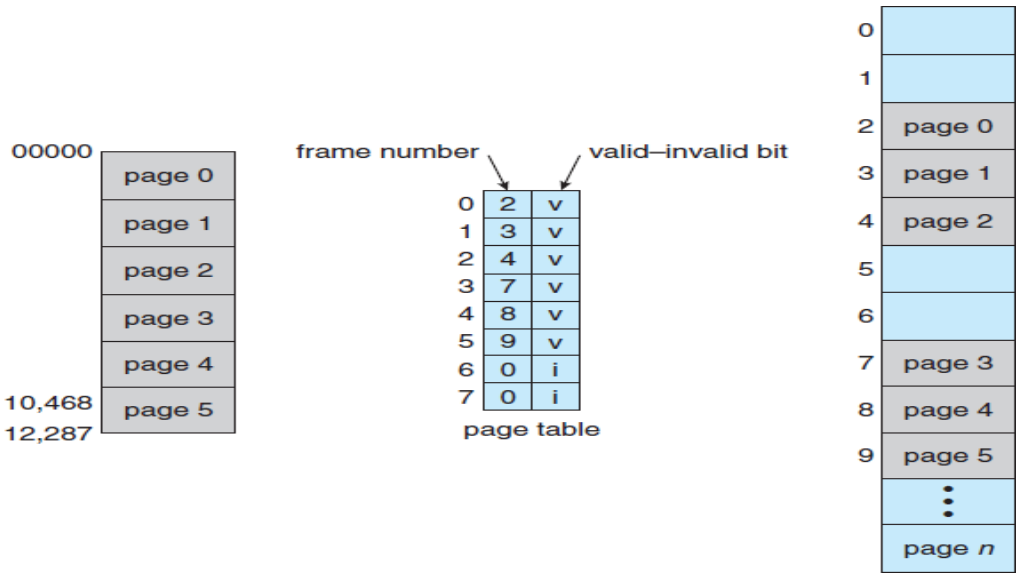

Figure 10: Valid (v) or invalid(i) bit in a page table

- ✓ **Problem:** Program extends only to address 10468, any reference beyond that address is illegal. But, references to page 5 are classified as valid, so access to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid
- ✓ **Internal fragmentation**: Many processes use only a small fraction of the address space available to them. It would be wasteful in these cases to create a page table with entries for every page in the address range
- ✓ **Solution:** Systems provide hardware in the form of a ***Page-table length register (PTLR),*** to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range of the process

4.4.4 Shared Pages

- ✓ **Advantage of Paging:** Sharing common page, which is important in time-sharing environment
- ✓ Reentrant code is non-self modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time
- ✓ If code is reentrant code (pure code), it can be shared as shown in figure 11
- ✓ Each process has its own data page
- ✓ Each process has its own copy of registers and data storage to hold the data for process's execution. Hence data for two different processes will be different
- ✓ Each user's page table maps onto the same physical copy of the process, but data pages are mapped onto different frames
- ✓ The sharing of memory among processes on a system is similar to the sharing of the address space of a task by threads

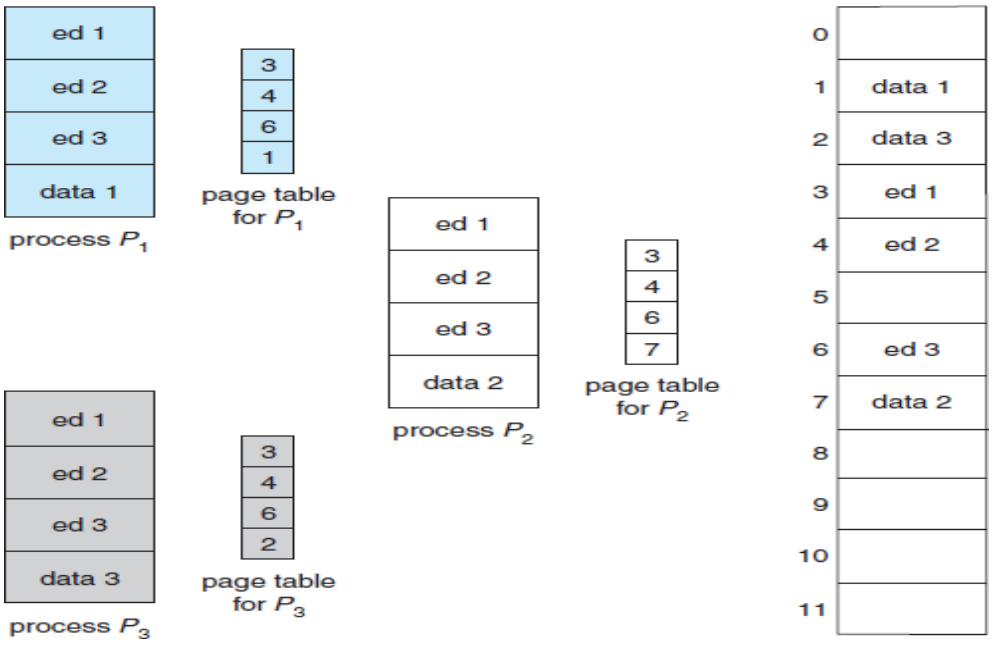

Figure 11: Sharing of code in a paging environment

4.5 Segmentation

- ✓ An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory from the actual physical memory
- ✓ The user's view is mapped onto physical memory. This mapping allows differentiation between logical and physical memory

4.5.1 Basic Method

- ✓ The main contains a set of methods, procedures, or functions. It also include various data structures: objects, arrays, stacks, variables and so on
- ✓ Each of these modules or data elements is referred by name called ***segments***
- ✓ Each of these segments is of variable length; the length is defined by the purpose of the segment in the program
- ✓ Elements within a segment are identified by their offset from the beginning of the segment
- ✓ ***Segmentation*** is a memory-management scheme that supports user view of memory
- ✓ A logical address space is a collection of segments
- ✓ Each segment has a ***name*** and a ***length***
- ✓ The addresses specify both the segment name and the offset within the segment
- ✓ The user specifies each address by two quantities: ***segment name*** and an ***offset***
- ✓ For implementation, segments are numbered and are referred by a ***segment number***, rather than by a segment name.
- ✓ Thus, a logical address consists of a two tuple: ***<segment-number, offset>***

4.5.2 Hardware

- ✓ The mapping of two-dimensional user-defined addresses into one-dimensional physical address is effected by ***segment table***

- ✓ Each entry in the segment table has a ***segment base*** and a ***segment limit***
- ✓ **Segment base** contains the starting physical address where the segment resides in memory
- ✓ **Segment limit** specifies the length of the segment
- ✓ The segment table is illustrated in figure 12.

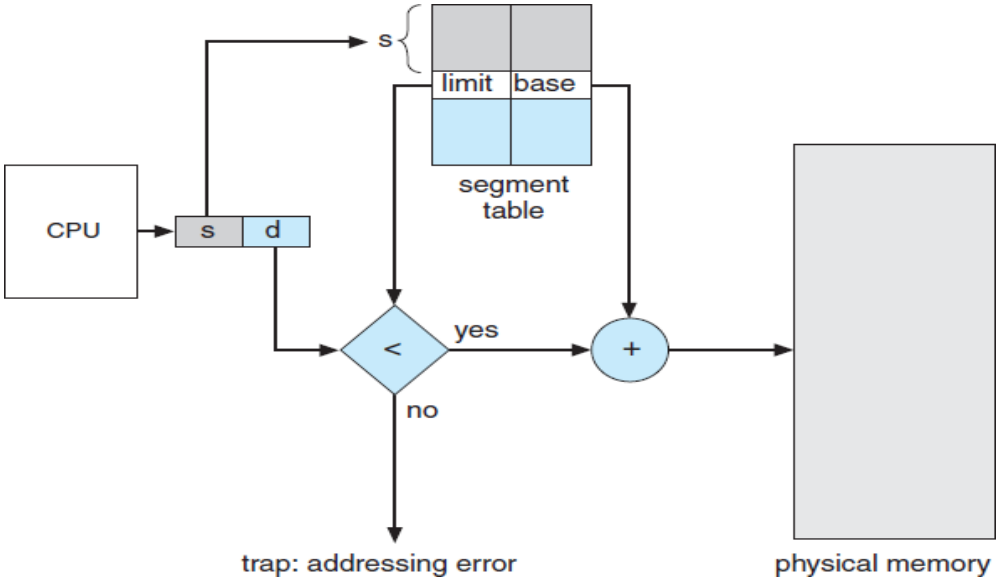

Figure 12: Segmentation hardware

- ✓ A logical address consists of two parts: a ***segment number, s***, and an ***offset*** into that segment, ***d***
- ✓ The segment number is used as an index to the segment table
- ✓ The offset *d* of the logical address must be between 0 and the segment limit
- ✓ If it is not (invalid), trap to the Os
- ✓ When an offset is legal (valid), it is added to the segment base to produce the address in the physical memory of the desired byte
- ✓ The segment table is an *array of base-limit register pairs*

Example: Consider the situation shown in Figure 13. We have five segments numbered from 0 through 4. The segments are stored in physical memory

- The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit).
- For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353.
- A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052.
- A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

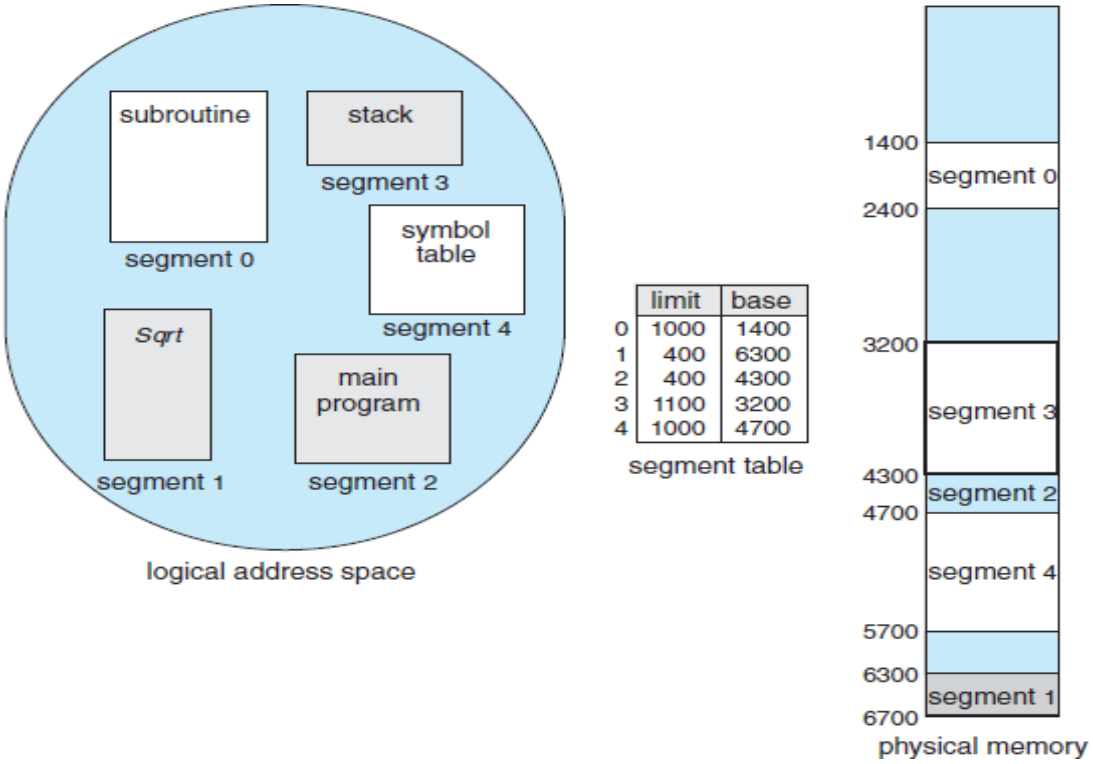

Figure 13: Example of Segmentation

VIRTUAL MEMORY

Basic Concepts

- ✓ Virtual memory is a technique that allows the execution of processes that are not completely in main memory (physical memory)
- ✓ Advantages:
  - Programs can be larger than the physical memory
  - Virtual memory allows processes to share files easily and to implement shared memory
- ✓ Virtual memory involves the separation of logical memory as viewed by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.
- ✓ Virtual memory makes the task of programming much easier because the programmer no longer needs to worry about the amount of physical memory available
- ✓ The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory
- ✓ Virtual address spaces that include free space are known as **sparse** address spaces

4.1 Demand Paging

- ✓ Load the pages only when the process needs, instead of loading entire program in physical memory at execution time. This technique is known as ***Demand Paging*** and is commonly used in virtual memory systems

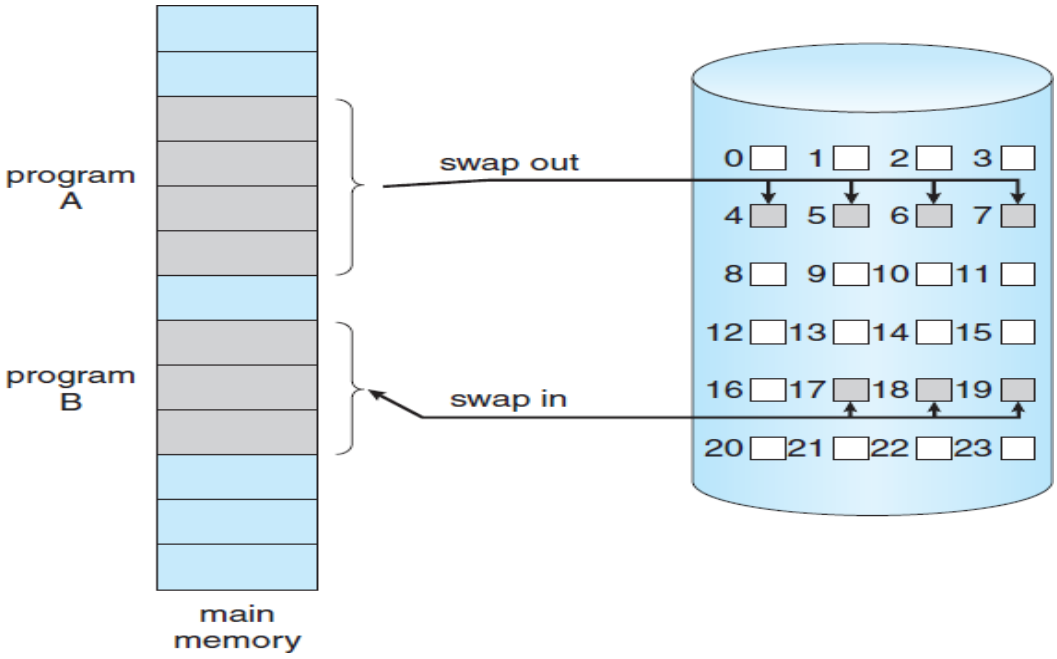

Figure 1: Transfer of paged memory to contiguous disk space

- ✓ With demand-paged virtual memory, pages are loaded only when they are demanded during program execution. Pages that are never accessed are thus never loaded into physical memory
- ✓ A demand-paging system is similar to a paging system with swapping as shown in figure 1 where processes reside in secondary memory (usually a disk).
- ✓ When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, though, we use a ***lazy swapper*** or ***pager***
- ✓ A lazy swapper never swaps a page into memory unless that page will be needed.

4.1.1 Basic Concepts

- ✓ The basic idea behind demand paging is that when a process is swapped in, the pager only loads into memory those pages that is needed presently.
- ✓ Hardware support is needed to distinguish between pages that are in memory and the pages that are on the disk by using *valid* and *invalid bit*
- ✓ If the bit is set to ***"valid,"*** the associated page is both legal and in memory.
- ✓ If the bit is set to ***"invalid,"*** 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.

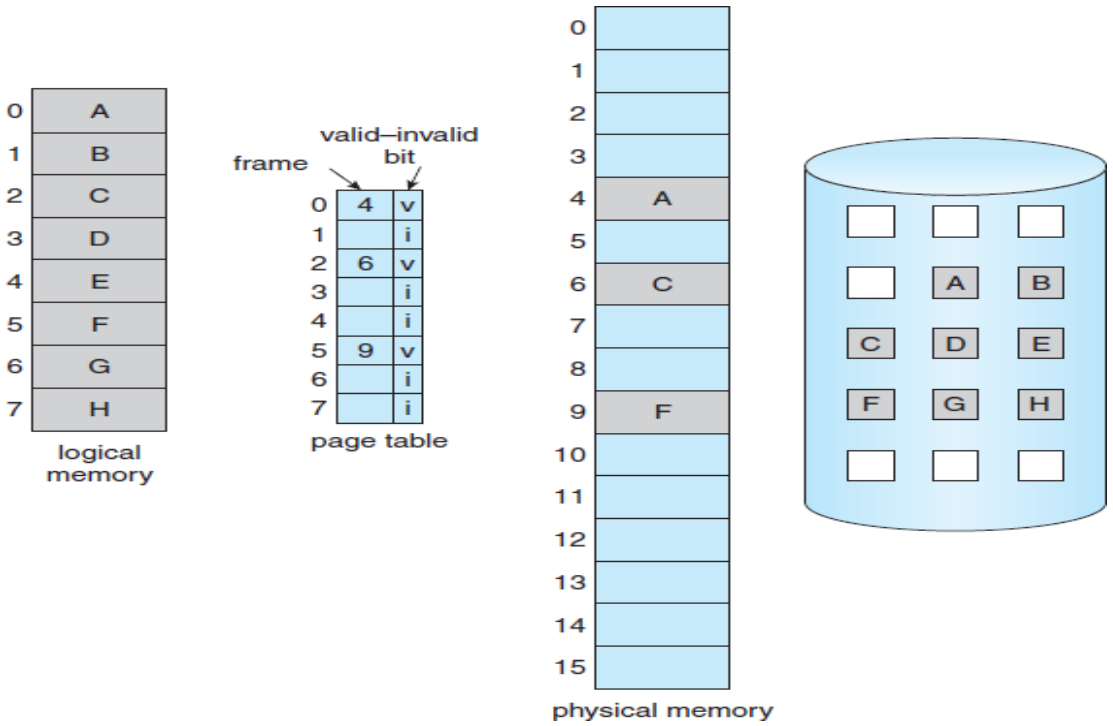

Figure 2: Page table when some pages are not in main memory

- ✓ The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk as shown in figure 2
- ✓ If a process tries to access a page that is not in the main memory, then access to that page is marked invalid and causes a ***page fault.*** (Page not in main memory)
- ✓ The paging hardware will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory.

✓ The procedure for handling this page fault is shown in figure 3

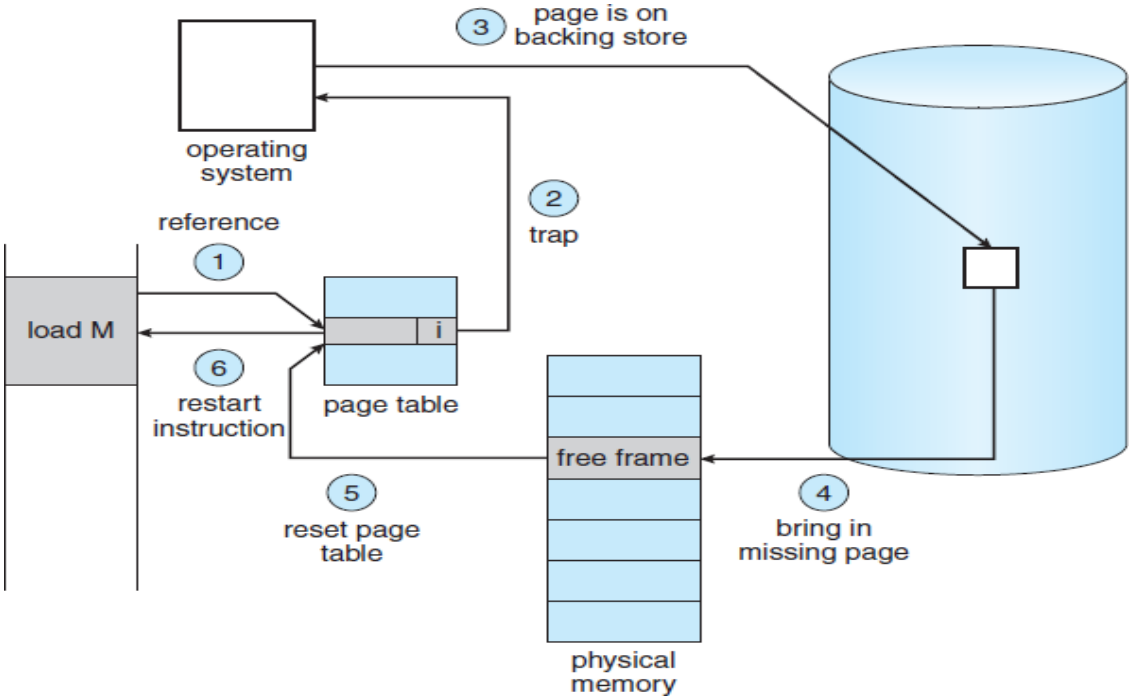

Figure 3: Steps in handling a page fault

1 We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2 If the reference was *invalid*, we terminate the process. If it was *valid* but we have not yet brought in that page, we now page it in.
3 We find a free frame (by taking one from the free-frame list, for example).
4 We schedule a disk operation to read the desired page into the newly allocated frame.
5 When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6 We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

✓ When the OS 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 that point, it can execute with no more faults. This scheme is ***pure demand paging***: never bring a page into memory until it is required.
✓ 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 a 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***.

- ✓ An important requirement for demand paging is the ability to restart instruction after a page fault
- ✓ A page fault may occur at any memory reference
  - If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again
  - If a page fault occurs while fetching an operand, we much fetch and decode the instruction again and them fetch the operand

4.1.2 Performance of Demand Paging

- ✓ The performance of demand paging can be measured by ***Effective Access Time***
- ✓ The *memory-access time* is denoted by *ma.*
- ✓ If NO page faults, the effective access time is equal to the memory access time
- ✓ Let p be the probability of a page fault ($0 \le p \le 1$). We would expect p to be close to zero—that is, we would expect to have only a few page faults.
- ✓ The **effective access time** is given by

$$\textbf{effective access time} = (1 - p) \times ma + p \times \textbf{page fault time}$$

- ✓ To compute the effective access time, we must know how much time is needed to service a 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. Receive an interrupt from the disk I/O subsystem (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, and then resume the interrupted instruction.

- ✓ Three Major components of the page-fault are:
  - Service the page-fault interrupt
  - Read in the page
  - Restart the process

- ✓ Effective access time is directly proportional to the page-fault rate
- ✓ It is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically

4.2 Page Replacement

- ✓ In order to make the most use of virtual memory, we load several processes into memory at the same time. Since we only load the pages that are actually needed by each process at any given time, there are frames to load many more processes in memory.
- ✓ If some process suddenly decides to use more pages and there aren't any free frames available. Then there are several possible solutions to consider:
  - Adjust the memory used by I/O buffering, etc., to free up some frames for user processes.
  - Put the process requesting more pages into a *wait queue* until some free frames become available.
  - Swap some process out of memory completely, freeing up its page frames.
  - Find some page in memory that isn't being used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it. This is known as ***page replacement***, and is the most common solution. There are many different algorithms for page replacement.

4.2.1 Basic Page Replacement

- ✓ Page replacement takes the following approach
- ✓ If no frames is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table to indicate that the page is no longer in memory as shown in figure 4
- ✓ Now the page-fault handling must be modified to free up a frame if necessary, as follows:

1 Find the location of the desired page on the disk.
2 Find a free frame:
a. If there is a free frame, use it.
b. If there is no free frame, use a page-replacement algorithm to select a ***victim frame***.
c. Write the victim frame to the disk; change the page and frame tables accordingly.
3 Read the desired page into the newly freed frame; change the page and frame tables
4 Continue the user process from where the page fault occurred.

- ✓ Page replacement is basic to demand paging
- ✓ If a page that has been modified is to be replaced, its contents are copied to the disk. A later reference to that page will cause a page fault. At the same time, the page will be brought back into memory, by replacing some other page in the process
- ✓ Major problems to implement demand paging: ***frame-allocation algorithm*** and ***page-replacement algorithm***
- ✓ If we have multiple processes in memory, we must decide how many frames to allocate to each process; and when page replacement is required, we must select the frames that are to be replaced

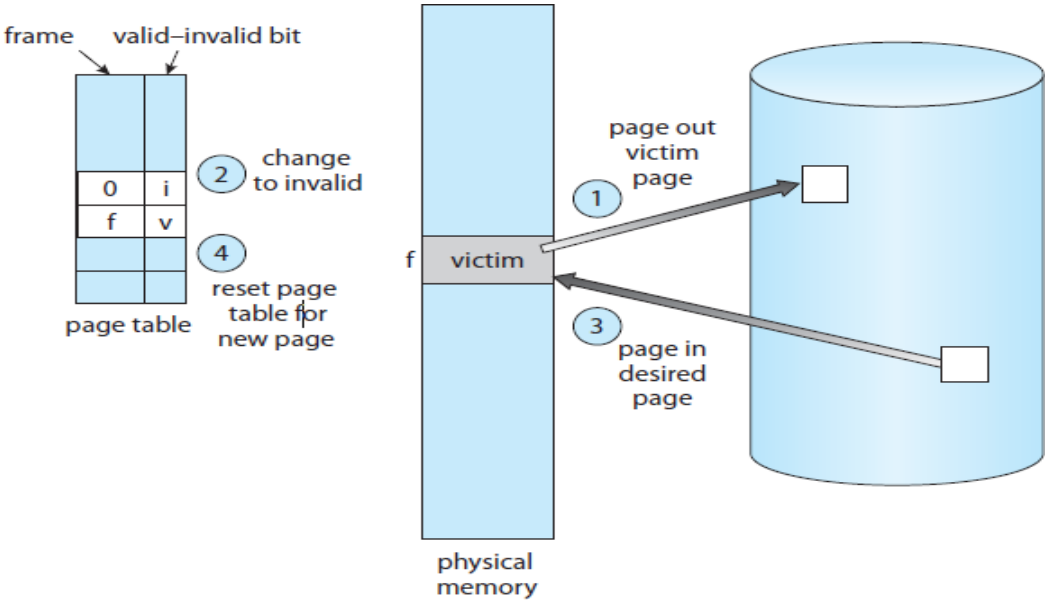

Figure 4: Page Replacement

✓ The string of memory references is called a **reference string**

4.2.2 FIFO (First-in, first-out) Page Replacement

✓ A simple and obvious page replacement strategy is ***FIFO***, i.e. first-in-first-out.
✓ This algorithm associates with each page the time when that page was brought into memory.
✓ When a page must be replaced, the oldest page is chosen.
✓ A FIFO queue can be created to hold all pages in memory. As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim.
✓ In the following example, a reference string is given and there are 3 free frames. There are 20 page requests, which results in **15 page faults**.

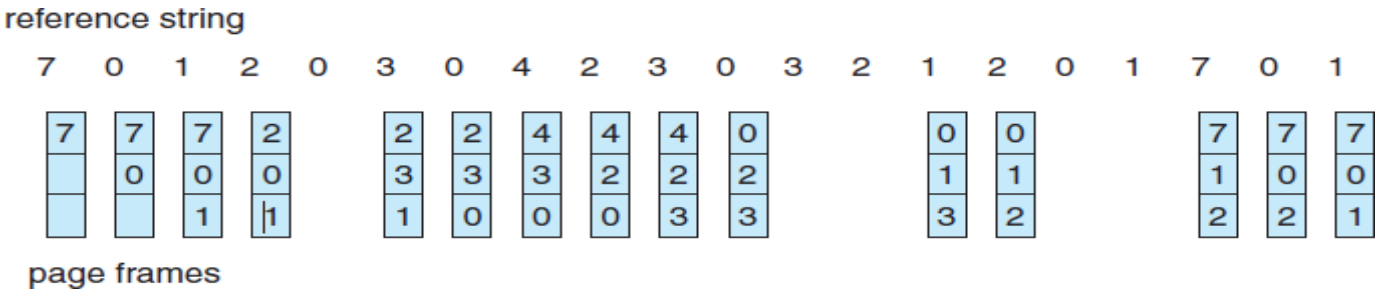

Figure 5: FIFO page-replacement algorithm

Advantage: It is easy to understand and program
Disadvantage: Performance is not always good

✓ **Belady's Anomaly:** For some page-replacement algorithm, the page-fault rate may *increase* as the number of allocated frames increases.

4.2.3 Optimal Page Replacement

- ✓ The discovery of Belady's anomaly lead to the search for an ***optimal page-replacement algorithm***, which is simply that which yields the lowest of all possible page-faults, and which does not suffer from Belady's anomaly
- ✓ Optimal page replacement algorithm states that, Replace the page that will not be used for the longest period of time

Advantage: Lowest page fault of all algorithms

Disadvantage: It is difficult to implement, because it requires future knowledge of the reference string. Hence, it is used for comparison studies

- ✓ In the following example, a reference string is given and there are 3 free frames. There are 20 page requests, which results in **09 page faults**

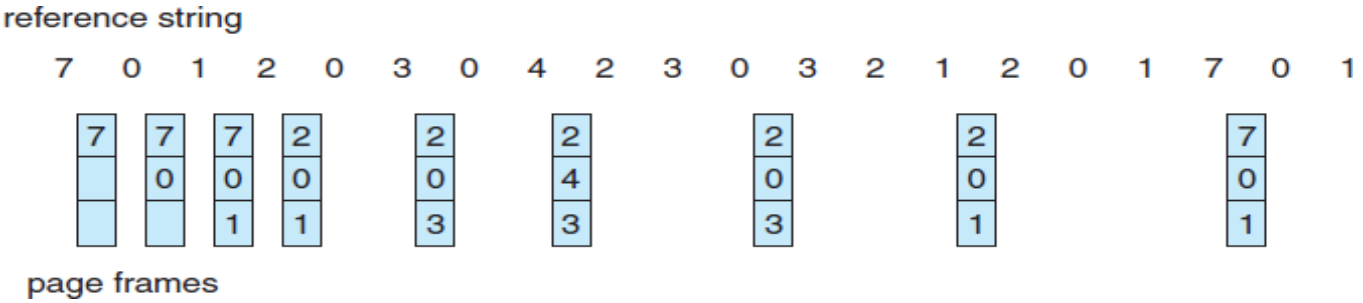

Figure 6: Optimal Page-replacement algorithm

4.2.4 LRU (Least Recently Used) Page Replacement

- ✓ The ***LRU (Least Recently Used)*** algorithm, predicts that the page that has not been used in the longest time is the one that will not be used again in the near future.
- ✓ LRU looks backward in order to replace the page
- ✓ In the following example, a reference string is given and there are 3 free frames. There are 20 page requests, which results in **12 page faults**

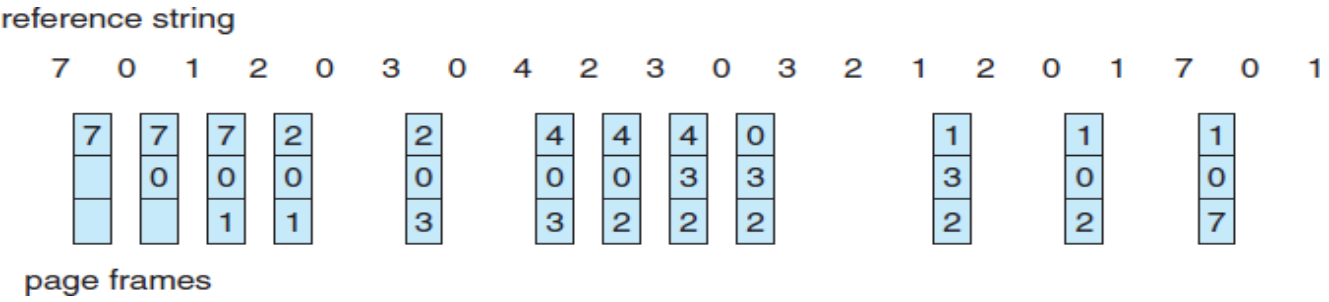

Figure 7: LRU page-replacement algorithm

- ✓ LRU is considered a good replacement policy, and is often used. There are two simple approaches commonly used to implement this:

  - **Counters:** With each page-table entry a time-of-use field is associated. 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, we always have the "time" of the last reference to each page. This scheme requires a search of the page table to find the LRU page and a write to memory for each memory access.

- **Stack:** Another approach is to use a stack. Whenever a page is referred, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom. Because this requires removing objects from the middle of the stack, a doubly linked list with a head pointer and a tail pointer is the recommended data structure.

✓ Like optimal, LRU replacement does not suffer from Belady's anomaly. Both belong to the class of page-replacement algorithm, called **stack algorithms**

4.2.5 LRU-Approximation Page Replacement

✓ Many systems offer some degree of hardware support, enough to approximate LRU.
✓ In particular, many systems provide a ***reference bit*** for every entry in a page table, which is set anytime that page is accessed.
✓ Initially all bits are set to zero, by OS. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used ad which have not been used by examining the reference bits, although we do not know the *order* of use

4.2.5.1 Additional-Reference-Bits Algorithm

✓ We can gain additional ordering information by recording the reference bits at regular intervals
✓ An 8-bit byte(reference bit) is stored for each page in a table in memory.
✓ At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the OS. The OS shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.
✓ These 8-bit shift registers contain the history of page use for the last eight time periods.
✓ If the shift register contains 00000000, then the page has not been used for eight time periods.
✓ A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.
✓ If we interrupt these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced

4.2.5.2 Second-Chance Algorithm

✓ The ***second chance algorithm*** is a FIFO replacement algorithm, except the reference bit is used to give pages a second chance at staying in the page table.
✓ When a page must be replaced, the page table is scanned in a FIFO (circular queue) manner.
✓ If a page is found with its reference bit as '0', then that page is selected as the next victim.

✓ If the reference bit value is '1', then the page is given a second chance and its reference bit value is cleared(assigned as'0').

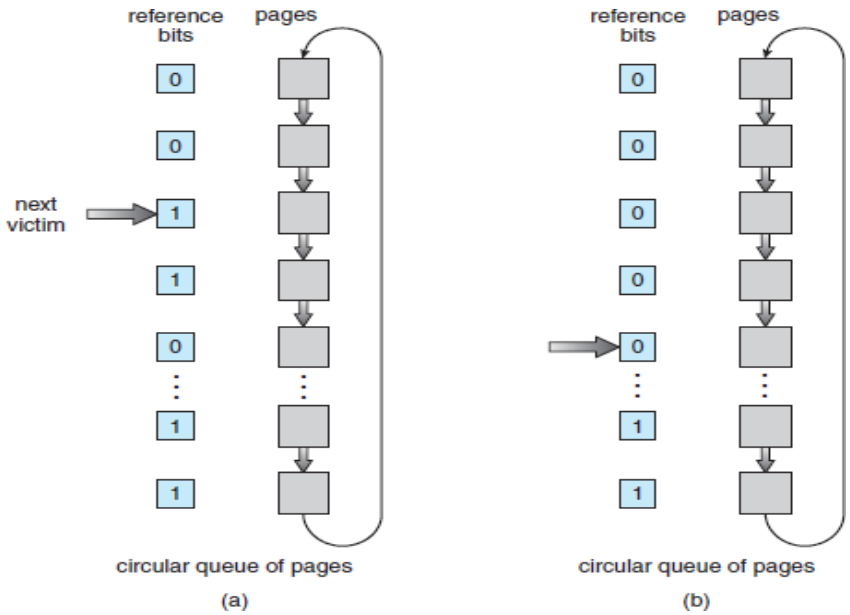

Figure 8: Second-chance (clock) page-replacement algorithm

✓ Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often, then it sets its reference bit again.

✓ This algorithm is also known as the ***clock*** algorithm.

✓ One way to implement the second-chance algorithm is as a circular queue. *A* pointer indicates which page is *to* be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits. Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.

4.2.5.3 Enhanced Second-Chance Algorithm

✓ The **enhanced second chance algorithm** looks at the reference bit and the modify bit (dirty bit) as an ordered page, and classifies pages into one of four classes:

1 (0, 0) - Neither recently used nor modified.
2 (0, 1) - Not recently used, but modified.
3 (1, 0) - Recently used, but clean.
4 (1, 1) - Recently used and modified.

✓ This algorithm searches the page table in a circular fashion, looking for the first page it can find in the lowest numbered category. i.e. it first makes a pass looking for a (0, 0), and then if it can't find one, it makes another pass looking for a (0, 1), etc.

✓ The main difference between this algorithm and the previous one is the preference for replacing clean pages if possible

4.2.6 Counting-Based Page Replacement

- ✓ There are several algorithms based on counting the number of references that have been made to a given page, such as:
  - ***Least Frequently Used (LFU):*** Replace the page with the lowest reference count. A problem can occur if a page is used frequently initially and then not used any more, as the reference count remains high. A solution to this problem is to right-shift the counters periodically, yielding a time-decaying average reference count.
  - ***Most Frequently Used (MFU):*** Replace the page with the highest reference count. The logic behind this idea is that pages that have already been referenced a lot have been in the system a long time, and we are probably done with them, whereas pages referenced only a few times have only recently been loaded, and we still need them.

4.2.7 Page-Buffering Algorithm

- ✓ Maintain a certain minimum number of free frames at all times. When a page-fault occurs, go ahead and allocate one of the free frames from the free list first, so that the requesting process is in memory as early as possible, and then select a victim page to write to disk and free up a frame.
- ✓ Keep a list of modified pages, and when the I/O system is idle, these pages are written to disk, and then clear the modify bits, thereby increasing the chance of finding a "clean" page for the next potential victim and page replacement can be done much faster.

4.3 Allocation of Frames

- ✓ Under *pure demand paging*, for example: all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults. The first 93 page faults would all get free frames from the free-frame list.
- ✓ When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

4.3.1 Minimum Number of Frames

- ✓ We must allocate at least a minimum number of frames
- ✓ One reason for allocating at least a minimum number of frames involves performance
- ✓ As the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution. In addition, when a page fault occurs before an executing instruction is complete, the instruction must be restarted
- ✓ The minimum number of frames is defined by the computer architecture
- ✓ The maximum number is defined by the amount of available physical memory

4.3.2 Allocation Algorithms

- ✓ **Equal Allocation:** The easiest way to split m frames among n processes is to give everyone an equal share m/n frames
  For example: If there are 93 frames and 5 processes, each process will get 18 frames. The 3 leftover frames can be used as a free-frame buffer pool.
- ✓ **Problem:** Various Process need differing amount of memory. Hence some amount of memory will be wasted
- ✓ To solve this problem, **Proportional Allocation** is used. In this allocation, we allocate available memory to each process according to its size
- ✓ Let the size of the virtual memory for process p_i be s_i, and define

$$S = \sum s_i$$

  Then, if the total number of available frames is m, we allocate a_i frames to process p_i, where a_i is approximately

$$a_i = s_i/S \times m$$

- ✓ Consider a system with a 1KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames.
- ✓ With proportional allocation, we would split 62 frames between two processes, as follows-

  m=62, S = (10+127)=137
  Allocation for process 1 = 10/137*62 ~ 4
  Allocation for process 2 = 127/137*62 ~57

  Thus allocates 4 frames and 57 frames to student process and database respectively

- ✓ With either equal or proportional allocation, a high priority process is treated the same as low-priority process.
- ✓ **Solution**: To use a proportional allocation scheme wherein the ratio of frames depends not on the relative sizes of processes but rather on the priorities of processes or on a combination of size and priority

4.3.3 Global versus Local allocation

- ✓ Page-replacement algorithm is classified into two broad categories: **global replacement** and **local replacement**
- ✓ **Global replacement:** Allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process i.e, one process can take a frame from another
  - A process can select a replacement from among its own frames or the frames of any lower-priority process
  - A process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it
  - **Problem**: Process cannot control its own page-fault rate. The set of pages in memory for a process depends not only on the paging behaviour of that process but also on the paging behaviour of other process

- Global replacement results in greater system throughout and is therefore the more common method

✓ **Local replacement:** Each process select from only its own set of allocated frames

- The number of frames allocated to process does not change
- The set of pages in memory for a process is affected by the paging behaviour of only that process

4.3.4 Non-Uniform Memory Access (NUMA)

✓ Usually the time required to access all memory in a system is equivalent.
✓ This may not be the case in *multiple-processor systems*, because a given CPU can access some sections of main memory faster than it can access others
✓ This difference is caused by how CPUs and memory are interconnected in the system
✓ Such a system is made up of several system boards, each containing multiple CPUs and some memory
✓ Systems in which memory access times vary significantly are known as **Non-Uniform Memory Access (NUMA)** systems, and without exception they are slower than systems in which memory and CPUs are located on the same motherboard
✓ Managing which page frames are stored at which locations can significantly affect performance in NUMA systems

4.4 Thrashing

✓ Thrashing is the state of a process where there is **high paging activity**. A process that is spending more time paging than executing is said to be ***thrashing.***

4.4.1 Cause of Thrashing

✓ Results in severe performance problems
✓ The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system through global page replacement algorithm
✓ Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes.
✓ As processes wait for the paging device, CPU utilization decreases
✓ The CPU scheduler sees the decreasing CPU utilization and ***increases*** the degree of multiprogramming as a result. Thus thrashing occurred and hence Page-fault rate and effective memory-access time also increases
✓ No work is done, because the processes are spending all their time in paging
✓ The concept of thrashing is illustrated in figure 9, in which CPU utilization is plotted against the degree of multiprogramming.
✓ As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply.

✓ At this point, to increase CPU utilization and stop thrashing, we must ***decrease*** the degree of multiprogramming.

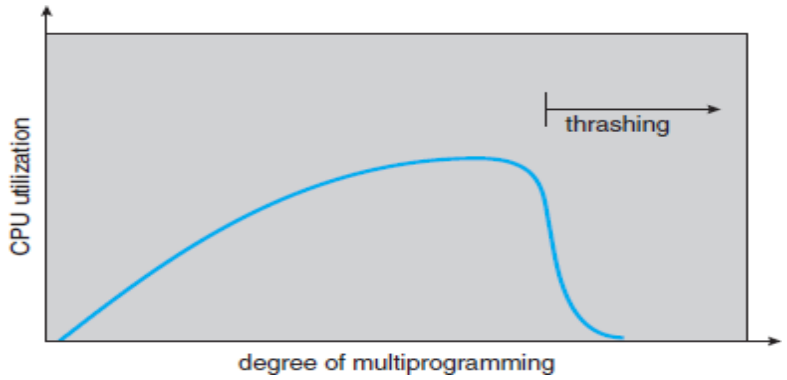

Figure 9: Thrashing

✓ The effect of thrashing can be limited by using a **local replacement algorithm (or priority replacement algorithm)**
✓ With local replacement, if one process starts thrashing, it cannot get frames from another process and cause the latter to thrash as well
✓ To prevent thrashing, we must provide a process with as many frames as it needs
✓ The ***working-set strategy*** looks at how many frames a process is actually using. This approach defines the ***locality model*** of process execution
✓ **Locality Model:** As process executes, it moves form locality to locality. A locality is a set of pages that are actively used together.
✓ A program is composed of several different localities, which may overlap

4.4.2 Working-Set Model

✓ **Working-set model** is based on the assumption of locality
✓ This model uses a parameter Δ, to define the working-set window
✓ The set of pages in the most recent Δ page references is the ***working set***
For example, given the sequence of memory references shown in Figure 10, if Δ= 10 memory references, then the working set at time t_1 is {1, 2, 5, 6, 7}. By time t_2, the working set has changed to {3, 4}.

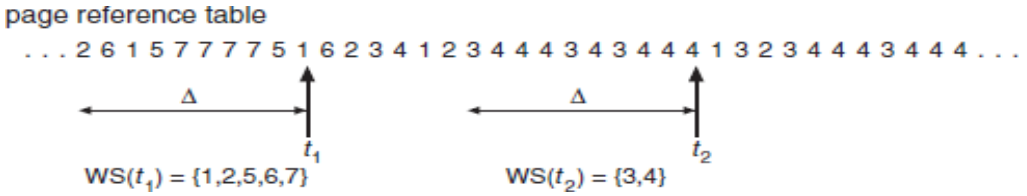

Figure 10: Working-set Model

✓ The accuracy of the working set depends on the Δ
✓ If Δ is too small, it will not encompass the entire locality; if Δ is too large, it may overlap several localities
✓ The most important property of the working set, is ***size***
✓ If we compute the working-set size, WSS_i for each process in the system, we get

$$D = \sum WSS_i$$

D is the total demand for frames

- ✓ If the total demand is greater than the total number of available frames ($D>m$), thrashing will occur, because some processes will not have enough frames
- ✓ Thrashing optimizes CPU utilization

4.4.3 Page-Fault Frequency

- ✓ Thrashing has a high page-fault rate. Thus, we control the page-fault rate.
- ✓ When it is too high, process needs more frames. Conversely, if the page-fault is too low, then the process may have too many frames
- ✓ Hence an upper and lower bound can be established on the desired page-fault rate as shown in figure 11

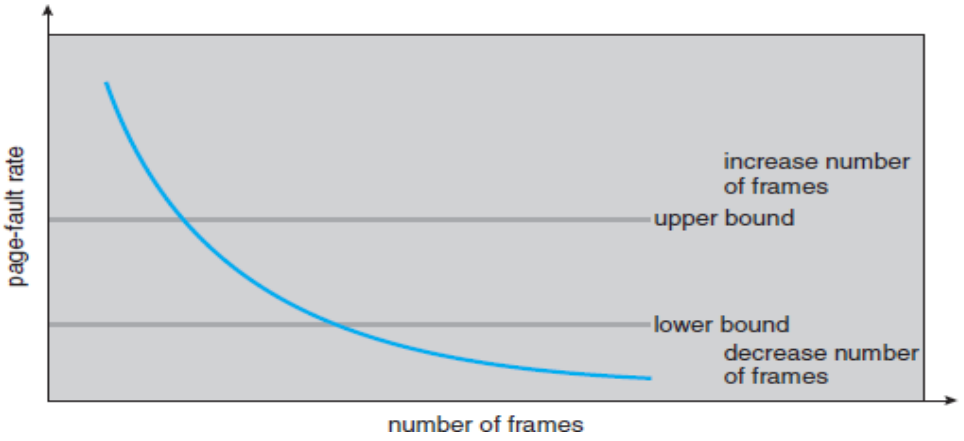

Figure 11: Page-fault frequency

- ✓ If the actual page-fault rate exceeds the upper limit, we allocate the process another frame.
- ✓ If the page-fault rate falls below the lower limit, we remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.