Chapter 8: Main Memory Management Strategies

Every program to be executed has to be executed must be in memory. The instruction must be fetched from memory before it is executed.

In multi-tasking OS memory management is complex, because as processes are swapped in and out of the CPU, their code and data must be swapped in and out of memory.

Main memory, cache and CPU registers in the processors are the only storage spaces that CPU can access directly.

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.

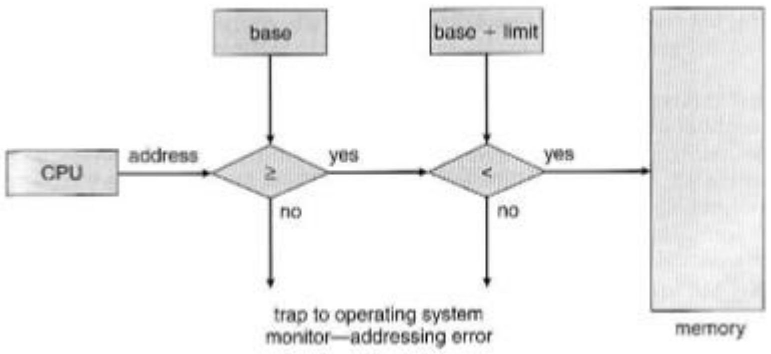
For example, if the base register holds 1000 and limit register is 500, then the program can legally access all addresses from 1000 through 1500 (inclusive).

Protection of memory space is done. Any attempt by an executing program to access operating system memory or other program memory results in a trap to the operating system, which treats the attempt as a fatal error.

This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction.

Since privileged instructions can be executed only in kernel mode only the operating system can load the base and limit registers.



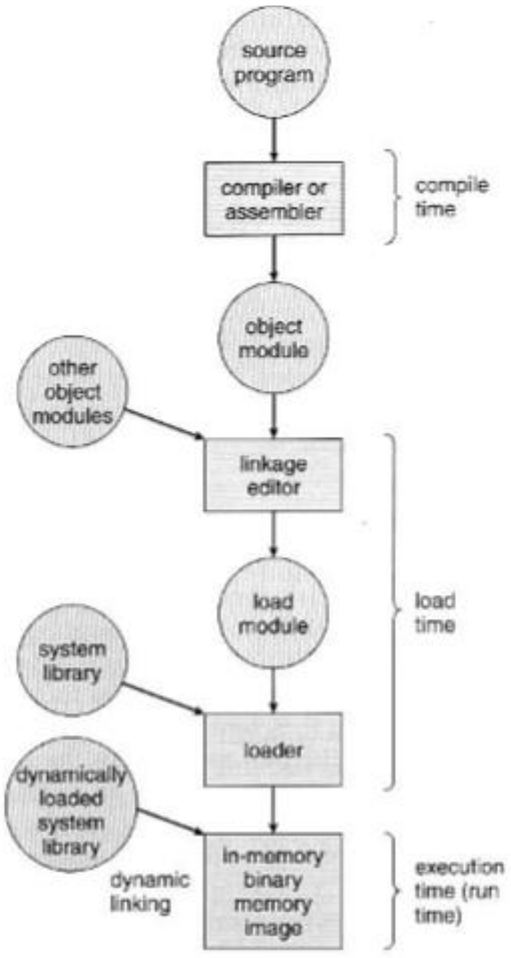
8.1.2 Address Binding

- ✓ User programs typically refer to memory addresses with symbolic names such as "i", "count", and "average Temperature".
- ✓ These symbolic names must be mapped or bound to physical memory addresses, which typically occurs in several stages:
- ✓ **Compile Time** - If it is known at compile time where a program will reside in physical memory, then absolute code can be generated by the compiler, containing actual physical addresses. However if the load address changes at some later time, then the program will have to be recompiled.
- ✓ **Load Time** - If the location at which a program will be loaded is not known at

compile time, then the compiler must generate relocatable code, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.

- ✓ **Execution Time** - If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time.

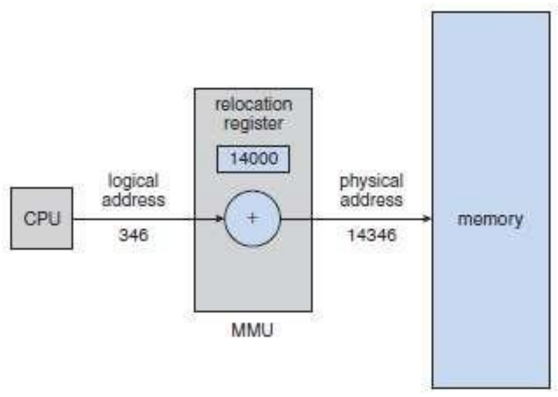
Below Figure shows the various stages of the binding processes and the units involved in each stage:



Multistep processing of a user-program

8.1.3 Logical Versus Physical Address Space

- ✓ The address generated by the CPU is a logical address, whereas the memory address where programs are actually stored is a physical address.
- ✓ The set of all logical addresses used by a program composes the logical address space, and the set of all corresponding physical addresses composes the physical address space.
- ✓ The run time mapping of logical to physical addresses is handled by the memory-management unit, MMU.
    - o The MMU can take on many forms. One of the simplest is a modification of the base-register scheme described earlier.
    - o The base register is now termed a relocation register, whose value is added to every memory request at the hardware level.



Dynamic relocation using a relocation-register

8.1.4 Dynamic Loading

- ✓ Rather than loading an entire program into memory at once, dynamic loading loads up each routine as it is called.
- ✓ The advantage is that unused routines need not be loaded, thus reducing total memory usage and generating faster program startup times.
- ✓ The disadvantage is the added complexity and overhead of checking to see if a routine is loaded every time it is called and then loading it up if it is not already loaded.

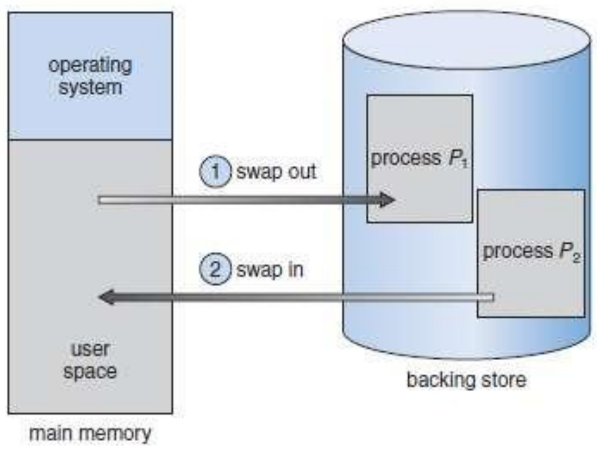
8.1.5 Dynamic Linking and Shared Libraries

- ✓ With static linking library modules get fully included in executable modules, wasting both disk space and main memory usage, because every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code.
- ✓ With dynamic linking, however, only a stub is linked into the executable module, containing references to the actual library module linked in at run time.
- ✓ This method saves disk space, because the library routines do not need to be fully included in the executable modules, only the stubs.
- ✓ An added benefit of dynamically linked libraries (DLLs, also known as shared libraries or shared objects on UNIX systems) involves easy upgrades and updates.

8.2 Swapping

- ✓ A process must be loaded into memory in order 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 backing store and moving another process from backing store to memory.
- ✓ Swapping is a very slow process compared to other operations.
- ✓ 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 is to either swap only totally idle processes, or do I/O operations only into and out of OS buffers, which are then transferred to or from process's main memory as a second step.

- ✓ Most modern OSes no longer use swapping, because it is too slow and there are faster alternatives available. (e.g. Paging.)
- ✓ However some UNIX systems will still invoke swapping if the system gets extremely full, and then discontinue swapping when the load reduces again.
- ✓ Windows 3.1 would use a modified version of swapping that was somewhat controlled by the user, swapping process's out if necessary and then only swapping them back in when the user focused on that particular window.



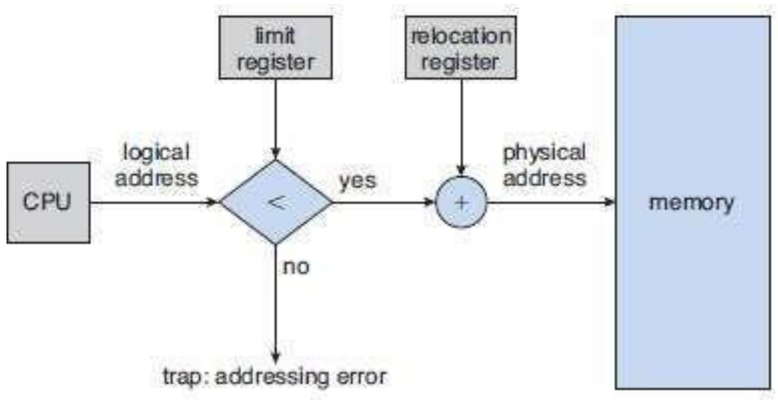
Swapping of two processes using a disk as a backing-store

8.3 Contiguous Memory Allocation

- ✓ One approach to memory management is to load each process into a contiguous space.
- ✓ The operating system is allocated space 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).
- ✓ Here each process is contained in a single contiguous section of memory.

8.3.1 Memory Mapping and Protection

• The system shown in figure below allows protection against user programs accessing areas that they should not, allows programs to be relocated to different memory starting addresses as needed, and allows the memory space devoted to the OS to grow or shrink dynamically as needs change.



8.3.2 Memory Allocation

• One method of allocating contiguous memory is to divide all available memory into equal sized partitions, and to assign each process to their own partition (called as MFT). This restricts both the number of simultaneous processes and the maximum size of each process, and is no longer used.

• An alternate approach is to keep a list of unused (free) memory blocks (holes), and to find a hole of a suitable size whenever a process needs to be loaded into memory (called as MVT).

There are many different strategies for finding the "best" allocation of memory to processes, including the three most commonly discussed:

1. **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 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.

2. **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.

3. **Worst fit** - Allocate the largest hole available, thereby increasing the likelihood that the remaining portion will be usable for satisfying future requests. Simulations show that either first or best fit are better than worst fit in terms of both time and storage utilization. First and best fits are about equal in terms of storage utilization, but first fit is faster.

8.3.3 Fragmentation

The allocation of memory to process leads to fragmentation of memory. A hole is the free space available within memory. The two types of fragmentation are –

External fragmentation –

holes present in between the process

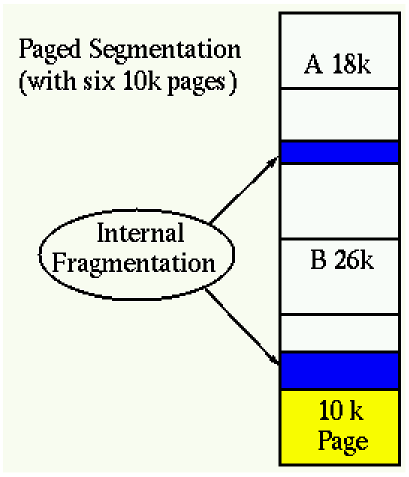
Internal fragmentation - holes are present within the process itself. ie. There is free space within a process.

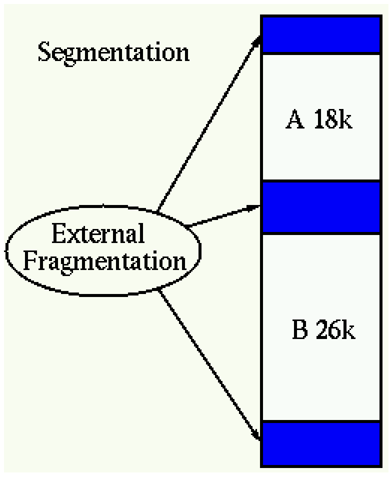
Internal fragmentation occurs with all memory allocation strategies.

This is caused by the fact that memory is allocated in blocks of a fixed size, whereas the actual memory needed will rarely be that exact size.

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.





8.4 Paging

- ✓ Paging is a memory management scheme that allows processes to be stored in physical memory discontinuously.
- ✓ It eliminates problems with fragmentation by allocating memory in equal sized blocks known as pages.
- ✓ Paging eliminates most of the problems of the other methods discussed previously, and is the predominant memory management technique used today.

8.4.1 Basic Method

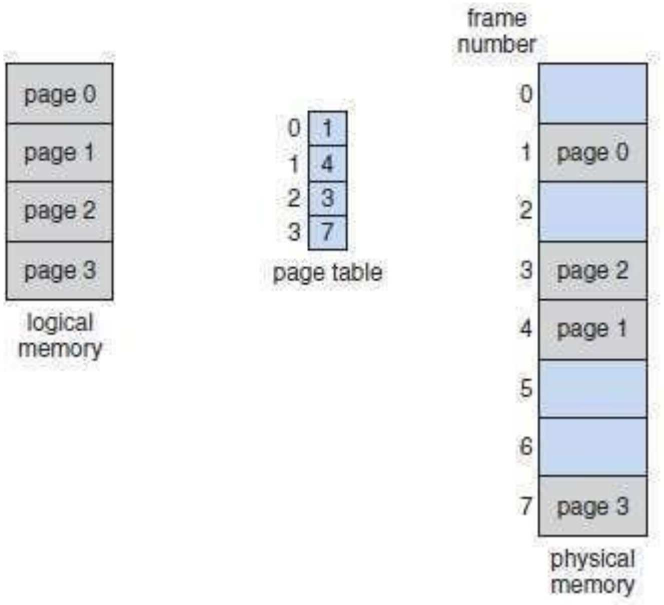
- ✓ The basic idea behind paging is to divide physical memory into a number of equal sized blocks called frames, and to divide a program's logical memory space into blocks of the same size called pages.

- ✓ Any page (from any process) can be placed into any available frame.
- ✓ The page table is used to look up which frame a particular page is stored in at the moment.
- ✓ In the following example, for instance, page 2 of the program's logical memory is currently stored in frame 3 of physical memory.
- ✓ A logical address consists of two parts: A page number in which the address

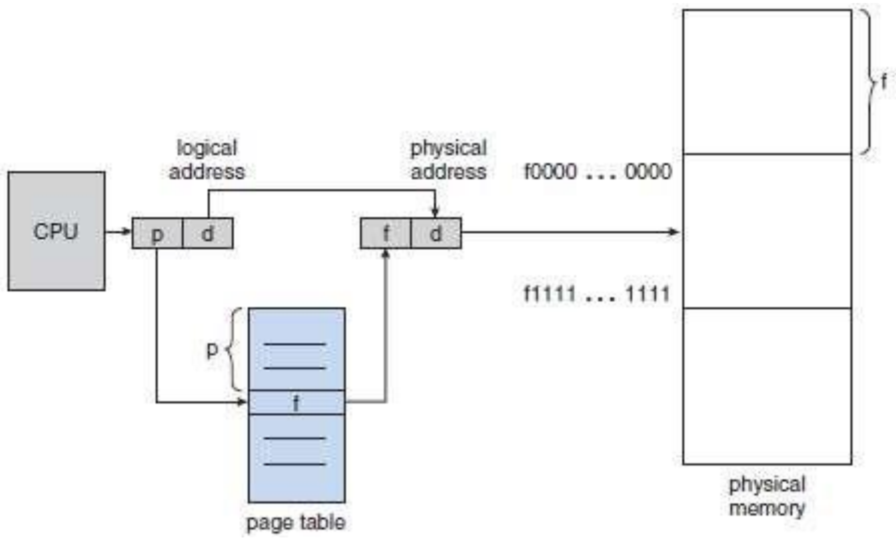
resides, and an offset from the beginning of that page. (The number of bits in the page number limits how many pages a single process can address.

- ✓ The number of bits in the offset determines the maximum size of each page, and should correspond to the system frame size.)
- ✓ The page table maps the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame.
- ✓ The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame.
- ✓ Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of two, allowing addresses to be split at a certain number of bits.

For example, if the logical address size is 2^m and the page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.

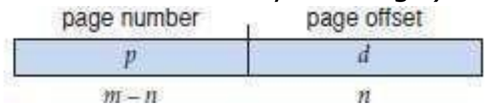


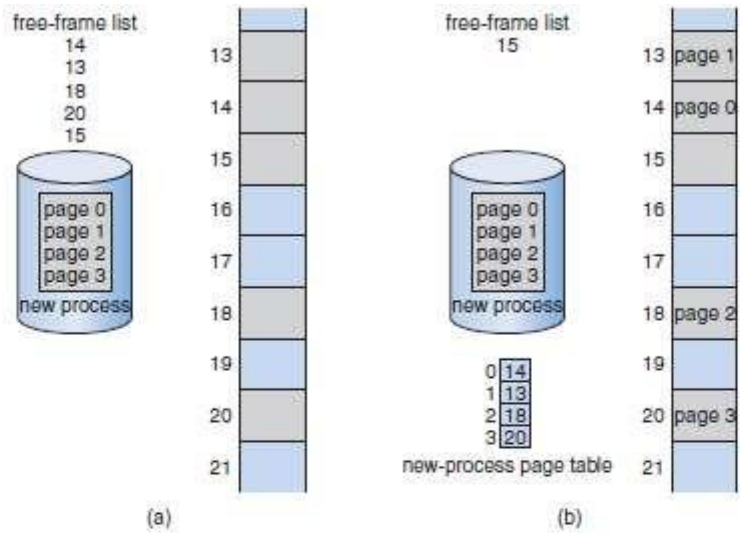
Paging model of logical and physical-memory



Paging hardware

- ✓ Note that paging is like having a table of relocation registers, one for each page of the logical memory.
- ✓ There is no external fragmentation with paging. All blocks of physical memory are used, and there are no gaps in between and no problems with finding the right sized hole for a particular chunk of memory.
- ✓ There is, however, internal fragmentation. Memory is allocated in chunks the size of a page, and on the average, the last page will only be half full, wasting on the average half a page of memory per process.
- ✓ Larger page sizes waste more memory, but are more efficient in terms of overhead.
- ✓ Modern trends have been to increase page sizes, and some systems even have multiple size pages to try and make the best of both worlds.
- ✓ When a process requests memory (e.g. when its code is loaded in from disk), free frames are allocated from a free-frame list, and inserted into that process's page table.
- ✓ Processes are blocked from accessing anyone else's memory because all of their memory requests are mapped through their page table.
- ✓ There is no way for them to generate an address that maps into any other process's memory space.
- ✓ The operating system must keep track of each individual process's page table, updating it whenever the process's pages get moved in and out of memory, and applying the correct page table when processing system calls for a particular process.
- ✓ This all increases the overhead involved when swapping processes in and out of the CPU. (The currently active page table must be updated to reflect the process that is currently running.)





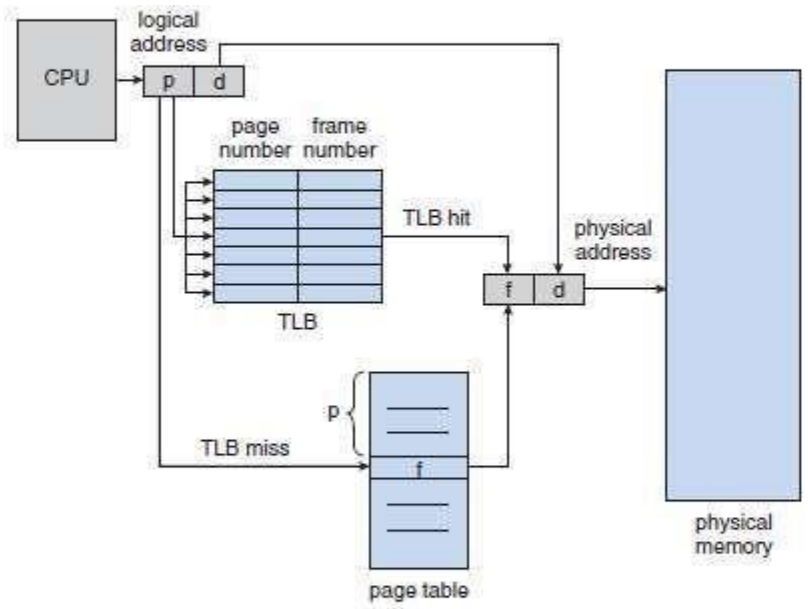
Free frames (a) before allocation and (b) after allocation

8.4.2 Hardware Support for Paging

- Most OS's store a page-table for each process.
- A pointer to the page-table is stored in the PCB.

Translation Lookaside Buffer

- The TLB is associative, high-speed memory.
- The TLB contains only a few of the page-table entries.
- Working:
  - ➢ When a logical-address is generated by the CPU, its page-number is presented to the TLB.
  - ➢ If the page-number is found (**TLB hit**), its frame-number is
    - → immediately available and
    - → used to access memory.
  - ➢ If page-number is not in TLB (**TLB miss**), a memory-reference to page table must be made.
  - ➢ The obtained frame-number can be used to access memory (Figure 3.19).
  - ➢ In addition, we add the page-number and frame-number to the TLB, so that they will be found quickly on the next reference.
- If the TLB is already full of entries, the OS must select one for replacement.
- Percentage of times that a particular page-number is found in the TLB is called **hit ratio**.
- Advantage: Search operation is fast.
  Disadvantage: Hardware is expensive.
- Some TLBs have wired down entries that can't be removed.
- Some TLBs store ASID (address-space identifier) in each entry of the TLB that uniquely
  → identify each process and
→ provide address space protection for that process.



Paging hardware with TLB

Some TLBs store address-space identifiers, ASIDs, to keep track of which process "owns" a particular entry in the TLB.

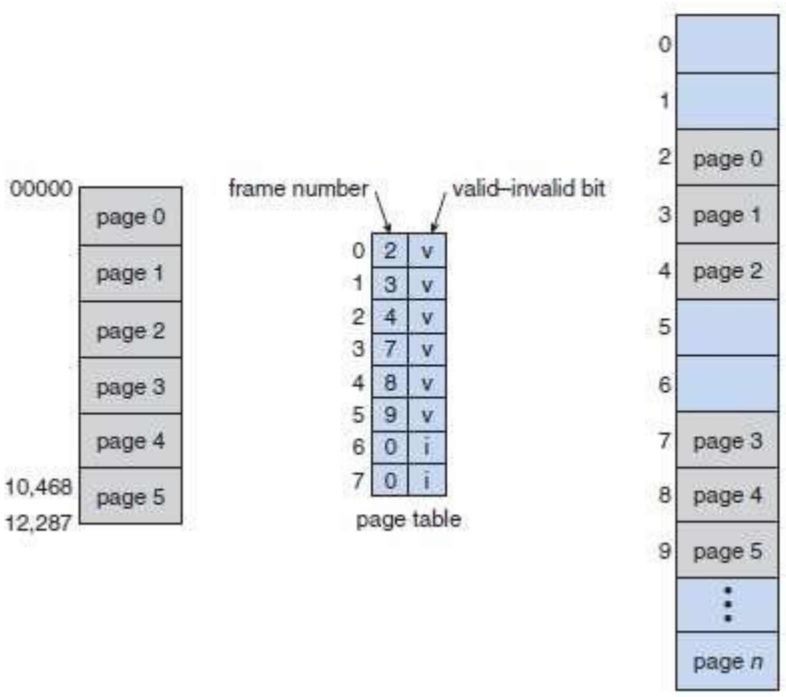
This allows entries from multiple processes to be stored simultaneously in the TLB without granting one process access to some other process's memory location. Without this feature the TLB has to be flushed clean with every process switch. ▪

The percentage of time that the desired information is found in the TLB is termed the hit ratio.

▪ For example, suppose that it takes 100 nanoseconds to access main memory, and only 20 nanoseconds to search the TLB. So a TLB hit takes 120 nanoseconds total (20 to find the frame number and then another 100 to go get the data), and a TLB miss takes 220 (20 to search the TLB, 100 to go get the frame number, and then another 100 to go get the data.) So with an 80% TLB hit ratio, the average memory access time would be: 0.80 * 120 + 0.20 * 220 = 140 nanoseconds for a 40% slowdown to get the frame number. A 98% hit rate would yield 122 nanoseconds average access time (you should verify this), for a 22% slowdown.

8.4.3 Protection

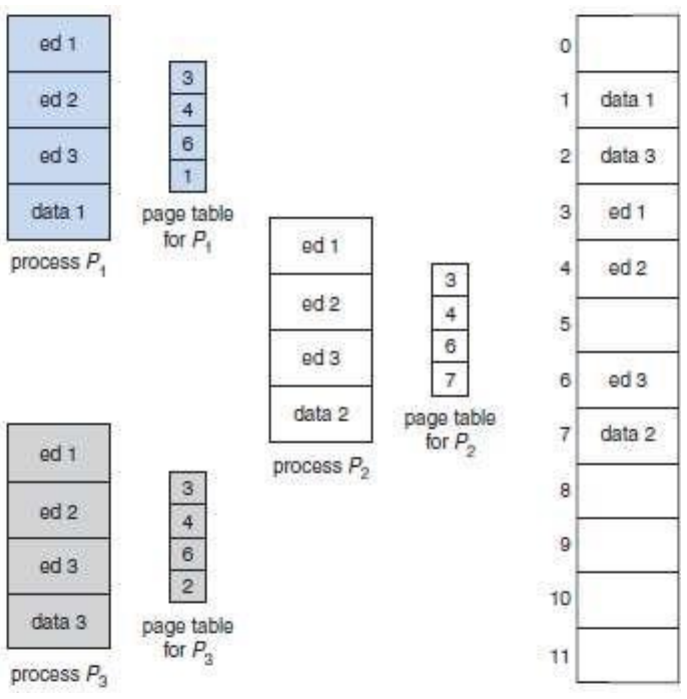
• The page table can also help to protect processes from accessing memory. • A bit can be added to the page table. Valid / invalid bits can be added to the page table. The valid bit 'V' shows that the page is valid and updated, and the invalid bit 'i' shows that the page is not valid and updated page is not in the physical memory. • Note that the valid / invalid bits described above cannot block all illegal memory accesses, due to the internal fragmentation. Many processes do not use all the page table entries available to them. • Addresses of the pages 0,1,2,3,4 and 5 are mapped using the page table as they are valid. • 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.



Valid (v) or invalid (i) bit in a page-table

8.4.4 Shared Pages

• Paging systems can make it very easy to share blocks of memory, by simply duplicating page numbers in multiple page frames. This may be done with either code or data. • If code is reentrant(read-only files) that means that it does not write to or change the code in any way. More importantly, it means the code can be shared by multiple processes, so long as each has their own copy of the data and registers, including the instruction register. • In the example given below, three different users are running the editor simultaneously, but the code is only loaded into memory (in the page frames) one time. • Some systems also implement shared memory in this fashion.



8.5 Structure of the Page Table

1) Hierarchical Paging
2) Hashed Page-tables
3) Inverted Page-tables

8.5.1 Hierarchical Paging

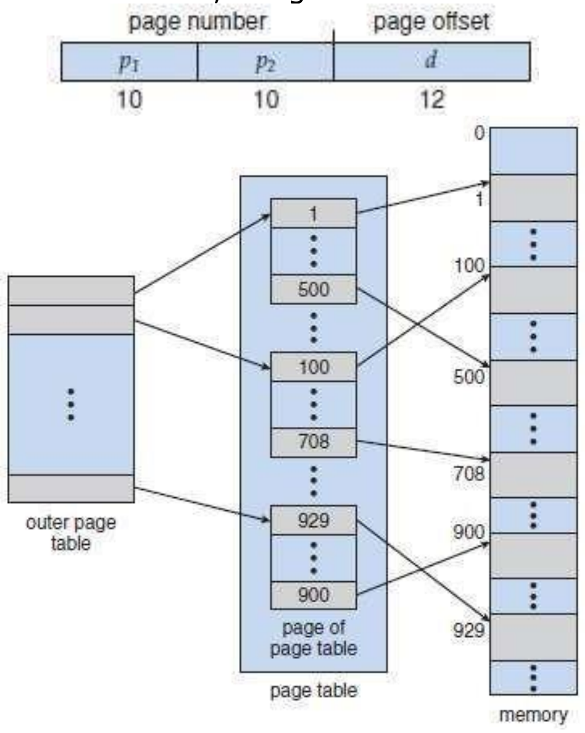
• Problem: Most computers support a large logical-address space (2^{32} to 2^{64}). In these systems, the page-table itself becomes excessively large.
Solution: Divide the page-table into smaller pieces.

Two Level Paging Algorithm

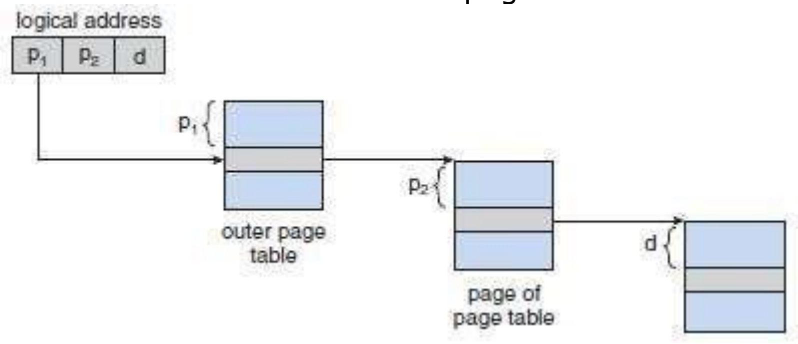
• The page-table itself is also paged (Figure 3.22).
• This is also known as a forward-mapped page-table because address translation works from the outer page-table inwards.
• For example (Figure 3.23):
  - ➢ Consider the system with a 32-bit logical-address space and a page-size of 4 KB.
  - ➢ A logical-address is divided into
    - → 20-bit page-number and
    - → 12-bit page-offset.
  - ➢ Since the page-table is paged, the page-number is further divided into

→ 10-bit page-number and
→ 10-bit page-offset.

➢ Thus, a logical-address is as follows:



two-level page-table scheme



Address translation for a two-level 32-bit paging architecture

8.5.2 Hashed Page Tables

• This approach is used for handling address spaces larger than 32 bits.
• The hash-value is the virtual page-number.
• Each entry in the hash-table contains a linked-list of elements that hash to the same location (to handle collisions).
• Each element consists of 3 fields:
1) Virtual page-number
2) Value of the mapped page-frame and
3) Pointer to the next element in the linked-list.
• The algorithm works as follows (Figure 3.24):
1) The virtual page-number is hashed into the hash-table.
2) The virtual page-number is compared with the first element in the linked-list.
3) If there is a match, the corresponding page-frame (field 2) is used to form the desired physical-address.
4) If there is no match, subsequent entries in the linked-list are searched for a matching virtual page-number.

Clustered Page Tables

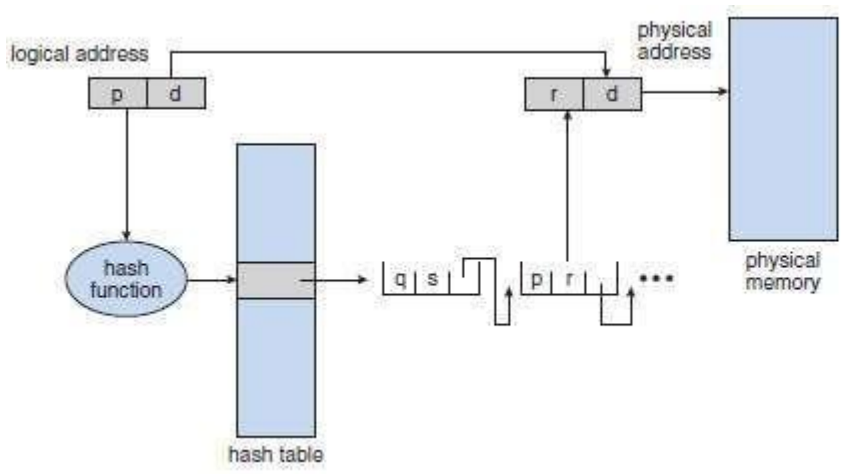
• These are similar to hashed page-tables except that each entry in the hash-table refers to several pages rather than a single page.

• Advantages:

1) Favorable for 64-bit address spaces.

Useful for address spaces, where memory-references are noncontiguous and scattered throughout the address space.

Hashed page-table



8.5.3 Inverted Page Tables

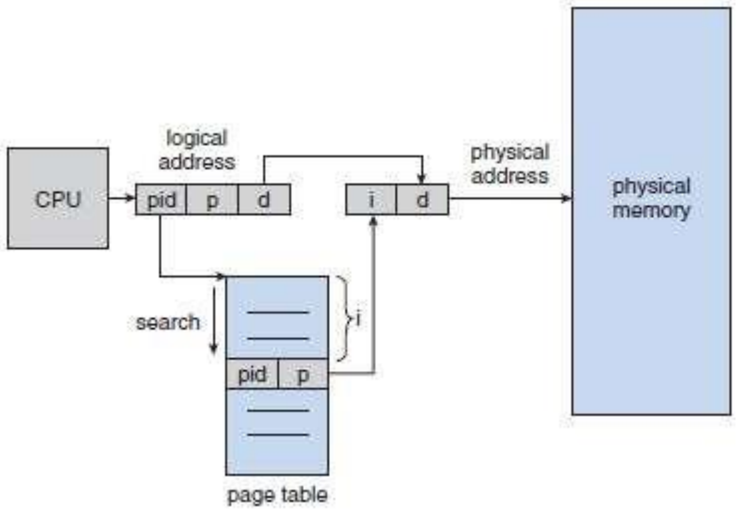
• Has one entry for each real page of memory.

• Each entry consists of

→ virtual-address of the page stored in that real memory-location and

→ information about the process that owns the page.

• Each virtual-address consists of a triplet (Figure 3.25):



<process-id, page-number, offset>.

• Each inverted page-table entry is a pair <process-id, page-number>

• The algorithm works as follows:

4) When a memory-reference occurs, part of the virtual-address, consisting of <process-id, page-number>, is presented to the memory subsystem.

5) The inverted page-table is then searched for a match.

6) If a match is found, at entry i-then the physical-address <i, offset> is generated.

7) If no match is found, then an illegal address access has been attempted.

• Advantage:

1) Decreases memory needed to store each page-table

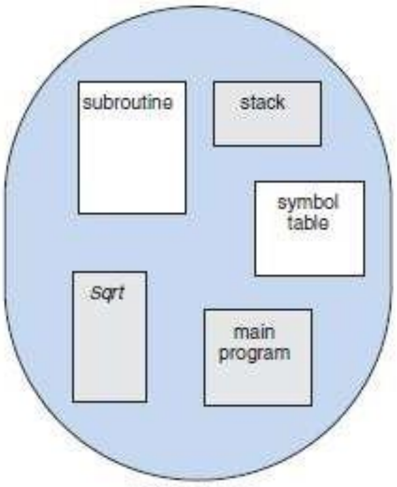
• Disadvantages:
  1) Increases amount of time needed to search table when a page reference occurs.
  2) Difficulty implementing shared-memory.

8.6 Segmentation

8.6.1 Basic Method

• This 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
  → segment-name and
  → offset within the segment.
• Normally, the user-program is compiled, and the compiler automatically constructs segments reflecting the input program.

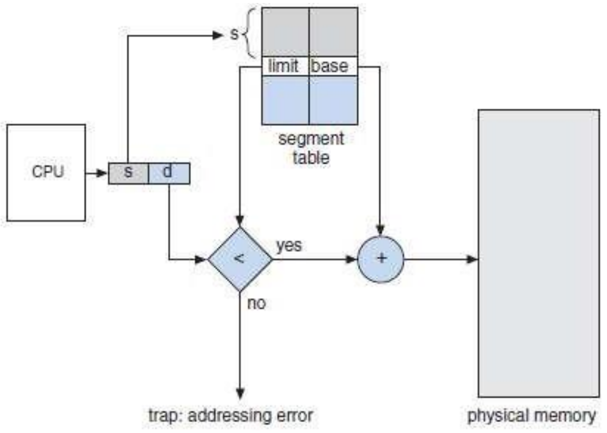
For ex:
→ The code
→ Global variables
→ The heap, from which memory is allocated
→ The stacks used by each thread
→ The standard C library



logical address

8.6.2 Hardware Support

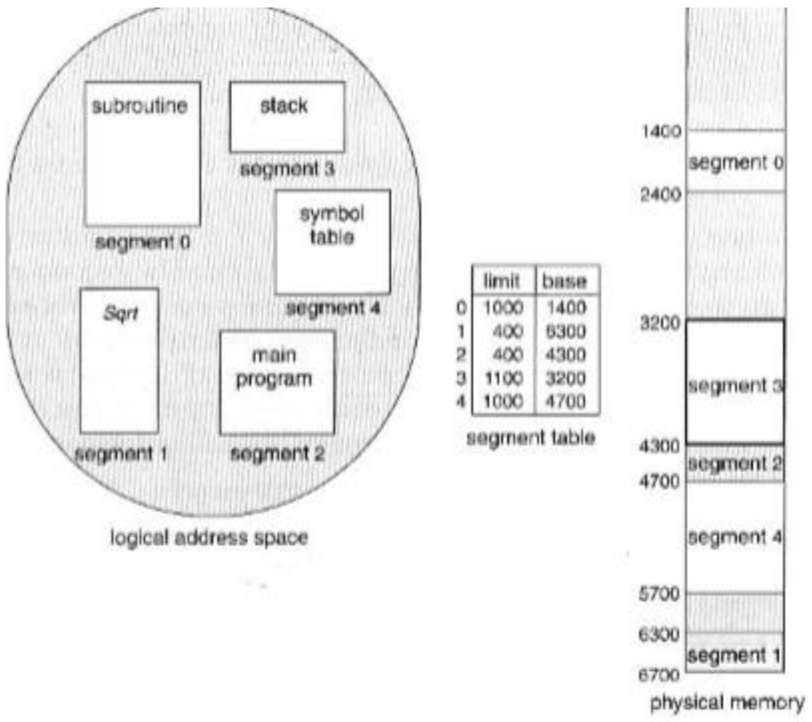
• Segment-table maps 2 dimensional user-defined addresses into one-dimensional physical-addresses.
• In the segment-table, each entry has following 2 fields:
  Segment-base contains starting physical-address where the segment resides in memory.
  Segment-limit specifies the length of the segment (Figure 3.27).
• A logical-address consists of 2 parts:
  1) Segment-number(s) is used as an index to the segment-table .
  2) Offset(d) must be between 0 and the segment-limit.
• If offset is not between 0 & segment-limit, then we trap to the OS(logical-addressing attempt beyond end of segment).
• If offset is legal, then it is added to the segment-base to produce the physical-memory address.



Segmentation hardware

Hardware

• A segment table maps segment-offset addresses to physical addresses, and simultaneously checks for invalid addresses. • Each entry in the segment table has a segment base and a segment limit. The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment. • 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. When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base and limit register pairs.

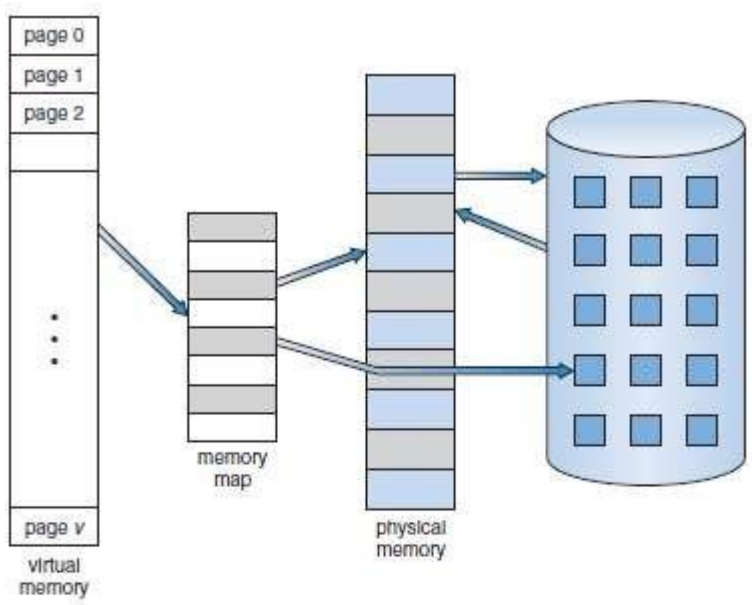


Chapter 9: Virtual-Memory Management

9.1 Virtual memory is a technique that allows the execution of processes that are not completely in memory.

✓ One major advantage of this scheme is that programs can be larger than physical memory.

- ✓ In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:
  - ▪ Error handling code is not needed unless that specific error occurs, some of which are quite rare.
  - ▪ Certain features of certain programs are rarely used.
- ✓ The ability to load only the portions of processes that are actually needed has several benefits:
  - ▪ Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer.
  - ▪ Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
  - ▪ Less I/O is needed for swapping processes in and out of RAM, speeding things up.
  - ▪ System libraries can be shared by mapping them into the virtual address space of more than one process.
  - ▪ Processes can also share virtual memory by mapping the same block of memory to more than one process.
- ✓ The figure below shows the general layout of virtual memory, which can be much larger than physical memory:



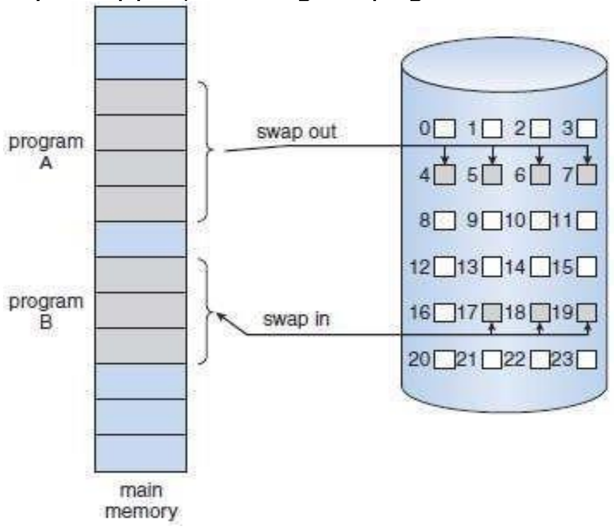
- ✓ Virtual-memory can be implemented by:
  - o Demand paging and
  - o Demand segmentation.
- ✓ The virtual (or logical) address-space of a process refers to the logical (or virtual) view of how a process is stored in memory.
- ✓ Physical-memory may be organized in page-frames and that the physical page-frames assigned to a process may not be contiguous.
- ✓ It is up to the MMU to map logical-pages to physical page-frames in memory.

9.2 Demand Paging

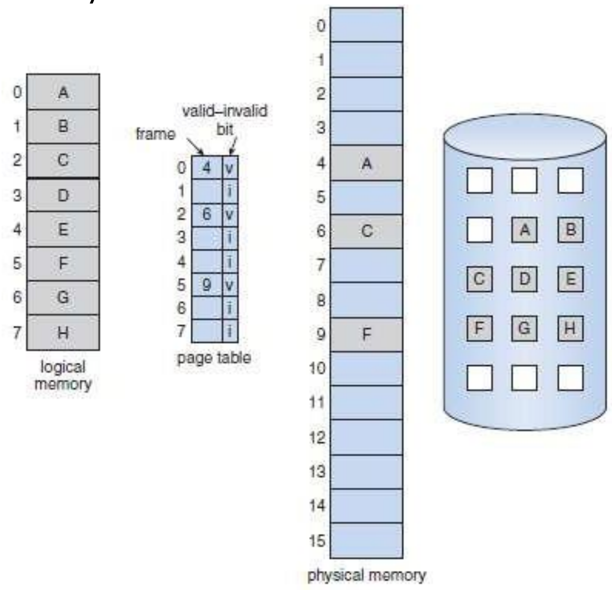
- ✓ A demand-paging system is similar to a paging-system with swapping.
- ✓ The basic idea behind demand paging is that when a process is swapped in, its pages are not swapped in all at once.
- ✓ Rather they are swapped in only when the process needs them (the pager only

loads into memory those pages that is needed presently or on demand.)

- ✓ This is termed as lazy swapper, although a pager is a more accurate term.



- ✓ Instead of swapping in a whole process, the pager brings only those necessary pages into memory
- ✓ **Advantages**:
  - ▪ Avoids reading into memory-pages that will not be used,
  - ▪ Decreases the swap-time and
  - ▪ Decreases the amount of physical-memory needed.
- ✓ The **valid-invalid bit** scheme can be used to distinguish between
  → pages that are in memory and
  → pages that are on the disk.
- ✓ 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 (i.e. not in the logical-address space of the process) or
  → is valid but is currently on the disk



Page-table when some pages are not in main-memory

- ✓ A **page-fault** occurs when the process tries to access a page that was not brought into memory.
- ✓ Procedure for handling the page-fault as shown in below figure,
  - ▪ Check an internal-table to determine whether the reference was a valid or an invalid memory access.
  - ▪ If the reference is invalid, we terminate the process.

- If reference is valid, but we have not yet brought in that page, we now page it in.
- Find a free-frame (by taking one from the free-frame list, for example).
- Read the desired page into the newly allocated frame.
- Modify the internal-table and the page-table to indicate that the page is now in memory.
- Restart the instruction that was interrupted by the trap.

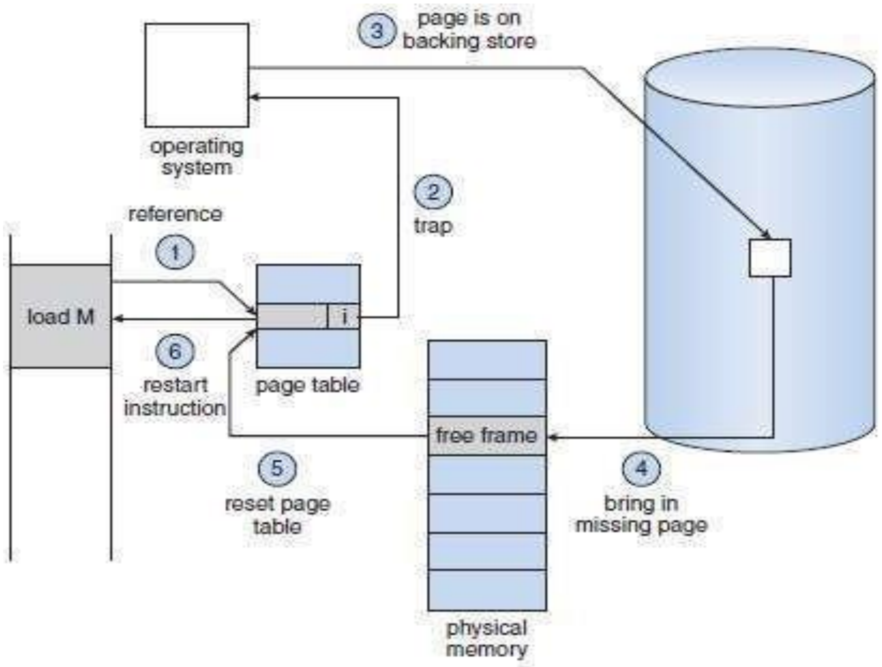


Fig: Steps in handling a page-fault

Pure demand paging:

✓ Never bring pages into memory until required.

✓ Some programs may access several new pages of memory with each instruction, causing multiple page-faults and poor performance.

✓ Programs tend to have locality of reference, so this results in reasonable performance from demand paging.

✓ Hardware support:
  - o **Page-table**
    - Mark an entry invalid through a valid-invalid bit.
  - o **Secondary memory**
    - It holds pages that are not present in main-memory.
    - It 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).

9.2.2 Performance:

✓ Demand paging can significantly affect the performance of a computer-system.

✓ Let p be the probability of a page-fault ($0 \le p \le 1$).

  if p = 0, no page-faults.

  if p = 1, every reference is a fault.

✓ effective access time(EAT)=[(1 - p) *memory access]+ [p *page-fault time]

✓ A page-fault causes the following events to occur:
  - Trap to the OS.
  - Save the user-registers and process-state.
  - Determine that the interrupt was a page-fault. '

  - Check that the page-reference was legal and determine the location of the page on the disk.
  - Issue a read from the disk to a free frame:
  - Wait in a queue for this device until the read request is serviced.
  - Wait for the device seek time.
  - Begin the transfer of the page to a free frame.
  - While waiting, allocate the CPU to some other user.
  - Receive an interrupt from the disk I/O subsystem (I/O completed).
  - Save the registers and process-state for the other user (if step 6 is executed).
  - Determine that the interrupt was from the disk.
  - Correct the page-table and other tables to show that the desired page is now in memory.
  - Wait for the CPU to be allocated to this process again.
  - Restore the user-registers, process-state, and new page-table, and then resume the interrupted instruction.

9.3 Copy-on-Write

- ✓ This technique allows the parent and child processes initially to share the same pages.
- ✓ If either process writes to a shared-page, a copy of the shared-page is created.
- ✓ For example:
  - Assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write.
  - OS will then create a copy of this page, mapping it to the address space of the child process.
  - Child process will then modify its copied page & not the page belonging to the parent process.

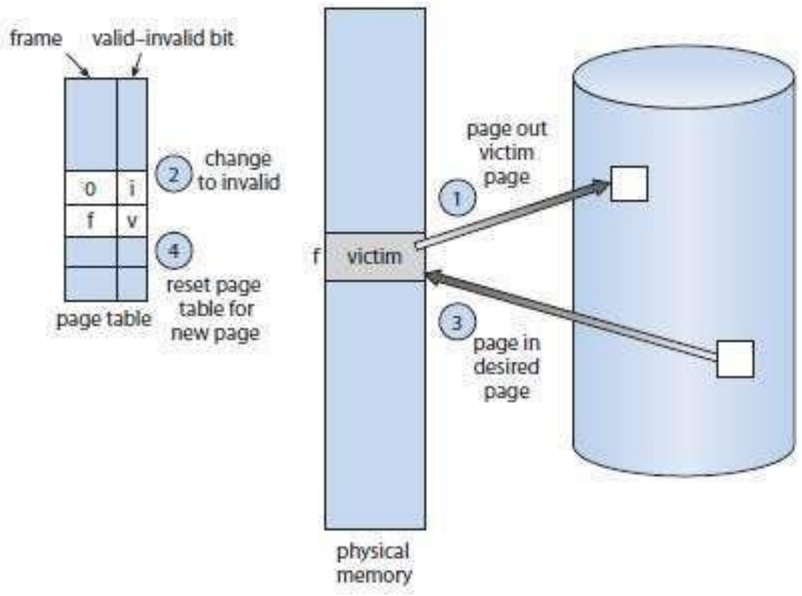
9.4 Page Replacement

- ✓ 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.

9.4.1 Basic page replacement approach is:

→ If no frame is free, we find one that is not currently being used and free it.

- ✓ Page replacement takes the following steps:
  - Find the location of the desired page on the 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-frame.
    - Write the victim-frame to the disk; change the page and frame-tables accordingly.
  - Read the desired page into the newly freed frame; change the page and frame-tables.
  - Restart the user-process.



Page replacement

- ✓ **Problem**: If no frames are free, 2 page transfers (1 out & 1 in) are required. This situation
  → doubles the page-fault service-time and
  → increases the EAT accordingly.
- ✓ **Solution**: Use a modify-bit (or dirty bit).
- ✓ Each page or frame has a modify-bit associated with the hardware.
- ✓ The **modify-bit** for a page is set by the hardware whenever any word is written into the page (indicating that the page has been modified).
- ✓ **Working**:
  - When we select a page for replacement, we examine it"s modify-bit.
  - If the modify-bit =1, the page has been modified. So, we must write the page to the disk.
  - If the modify-bit=0, the page has not been modified. So, we need not write the page to the disk, it is already there.
- ✓ **Advantage**:
  Can reduce the time required to service a page-fault.
- ✓ We must solve 2 major problems to implement demand paging:
  - Develop a **Frame-allocation algorithm:**
    If we have multiple processes in memory, we must decide how many frames to allocate to each process.
  - Develop a **Page-replacement algorithm:**
    We must select the frames that are to be replaced.

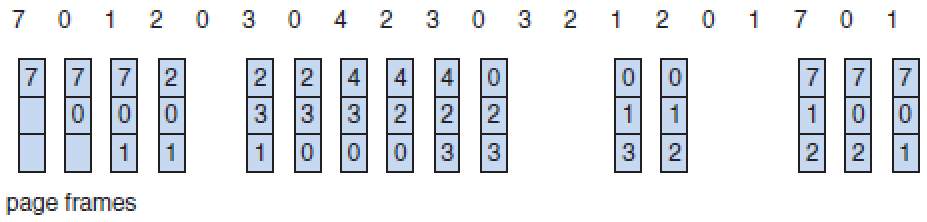
Page Replacement algorithm

- FIFO page replacement
- Optimal page replacement
- LRU page replacement (Least Recently Used)
- LFU page replacement (Least Frequently Used)

9.4.2 FIFO Page Replacement

- Each page is associated with the time when that page was brought into memory.
- When a page must be replaced, the oldest page is chosen.
- We use a FIFO queue to hold all pages in memory
  - When a page must be replaced, we replace the page at the head of the queue
  - When a page is brought into memory, we insert it at the tail of the queue.

Example: Consider the following references string with frames initially empty.



- The first three references(7, 0, 1) cause page-faults and are brought into these empty frames.
- The next reference(2) replaces page 7, because page 7 was brought in first.
- Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
- The first reference to 3 results in replacement of page 0, since it is now first in line.
- This process continues till the end of string.
- There are fifteen faults altogether.

Advantage:

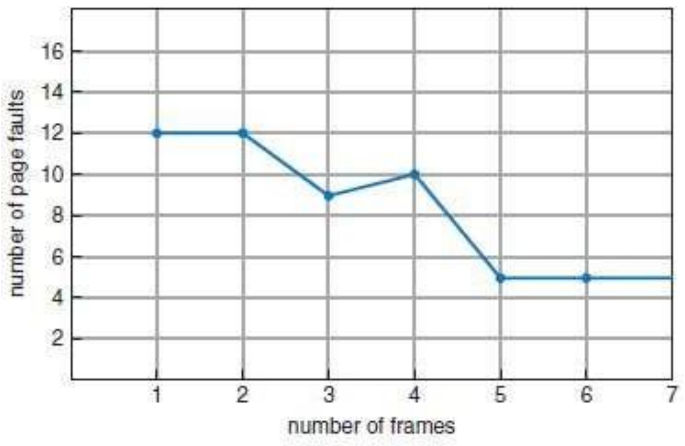
Easy to understand & program.

Disadvantages:

- Performance is not always good .
- A bad replacement choice increases the page-fault rate (Belady's anomaly).

For some algorithms, the page-fault rate may increase as the number of allocated frames increases. This is known as Belady's anomaly.

Example: Consider the following reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
For this example, the number of faults for four frames (ten) is greater than the number of faults for three frames (nine)!



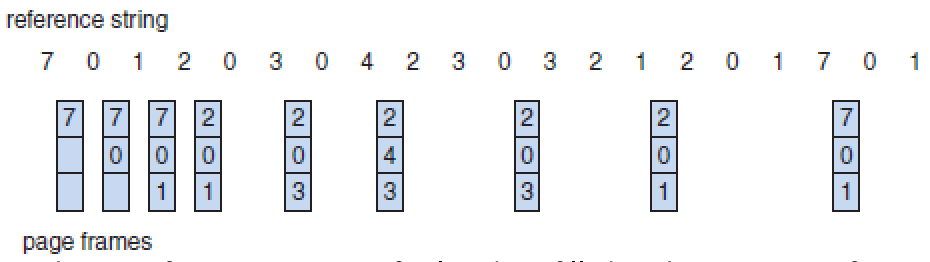
Page-fault curve for FIFO replacement on a reference string

****

9.4.3 Optimal Page Replacement

- Working principle: Replace the page that will not be used for the longest period of time.
- This is used mainly to solve the problem of Belady"s Anamoly.
- This has the lowest page-fault rate of all algorithms.

Consider the following reference string:



- The first three references cause faults that fill the three empty frames.
- The reference to page 2 replaces page 7, because page 7 will not be used until reference 18.
- The page 0 will be used at 5, and page 1 at 14.
- With only nine page-faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults.

Advantage:

Guarantees the lowest possible page-fault rate for a fixed number of frames.

Disadvantage:

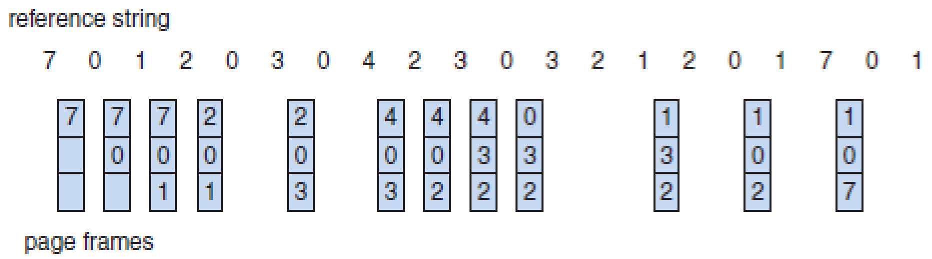
Difficult to implement, because it requires future knowledge of the reference string.

****

9.4.4 LRU Page Replacement

- The key difference between FIFO and OPT:
  → FIFO uses the time when a page was brought into memory.
  → OPT uses the time when a page is to be used.
- **Working principle**:
  - Replace the page that has not been used for the longest period of time.
  - Each page is associated with the time of that page's last use

Example: Consider the following reference string:



- The first five faults are the same as those for optimal replacement.
- When the reference to page 4 occurs, LRU sees that of the three frames, page 2 was used least recently. Thus, the LRU replaces page 2.
- The LRU algorithm produces twelve faults.
- **Two methods of implementing LRU:**
  1. Counters
     - Each page-table entry is associated with a time-of-use field.
     - A counter(or logical clock) 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.
     - We replace the page with the smallest time value.
  2. Stack
     - Keep a stack of page-numbers (Figure 4.11).
     - Whenever a page is referenced, the page is removed from the stack and put on the top.
     - The most recently used page is always at the top of the stack. The least recently used page is always at the bottom.
     - Stack is best implement by a doubly linked-list.

Advantage:
Does not suffer from Belady's anomaly.

Disadvantage:
Few computer systems provide sufficient h/w support for true LRU page replacement.

****

9.4.5 LRU-Approximation Page Replacement

- Some systems provide a reference bit for each page.
- Initially, all bits are cleared(to 0) by the OS.
- As a user-process executes, the bit associated with each page referenced is set (to 1) by the hardware.
- By examining the reference bits, we can determine
  → which pages have been used and
  → which have not been used.
- This information is the basis for many page-replacement algorithms that approximate LRU replacement.

9.4.5.1 Additional-Reference-Bits Algorithm:

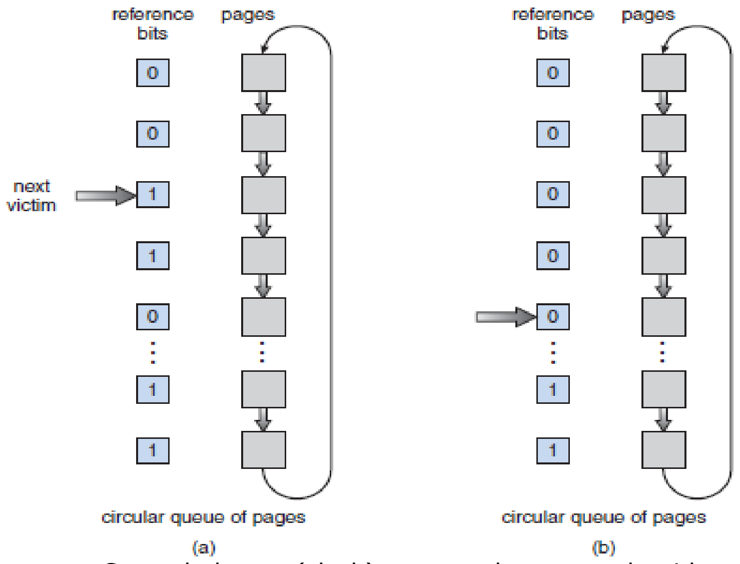
- We can gain additional ordering information by recording the reference bits at regular intervals.
- A 8-bit byte is used for each page in a table in memory.
- At regular intervals, a timer-interrupt transfers control to the OS.
- The OS shifts the reference bit for each page into the high-order bit of its 8-bit byte.
- These 8-bit shift registers contain the history of page use, for the last eight time

periods.

- Examples:
  00000000 - This page has not been used in the last 8 time units (800 ms). 11111111 - Page has been used every time unit in the past 8 time units. 11000100 has been used more recently than 01110111.
- The page with the lowest number is the LRU page, and it can be replaced.
- If numbers are equal, FCFS is used

9.4.5.2 Second-Chance Algorithm:

- The number of bits of history included in the shift register can be varied to make the updating as fast as possible.
- In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This
- algorithm is called the second-chance algorithm.
- Basic algorithm is a FIFO replacement algorithm.
- Procedure:
  - When a page has been selected, we inspect its reference bit.
  - If reference bit=0, we proceed to replace this page.
  - If reference bit=1, we give the page a second chance & move on to select next FIFO page.
  - When a page gets a second chance, its reference bit is cleared, and its arrival time is reset.



Second-chance (clock) page-replacement algorithm

- A circular queue can be used to implement the second-chance algorithm.
  - A pointer (that is, a hand on the clock) 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.

9.4.5.3 Enhanced Second-Chance Algorithm

- We can enhance the second-chance algorithm by considering
  - Reference bit and
  - modify-bit.
- We have following 4 possible classes:

- (0, 0) neither recently used nor modified -best page to replace.
- (0, 1) not recently used hut modified-not quite as good, because the page will need to be written out before replacement.
- (1, 0) recently used but clean-probably will be used again soon.
- (1, 1) recently used and modified -probably will be used again soon, and the page will be need to be written out to disk before it can be replaced.

- Each page is in one of these four classes.
- When page replacement is called for, we examine the class to which that page belongs.
- We replace the first page encountered in the lowest nonempty class.

****

9.4.6 Counting-Based Page Replacement

LFU page-replacement algorithm:

- **Working principle:** The page with the smallest count will be replaced.
- The reason for this selection is that an actively used page should have a large reference count.
- **Problem**:
  When a page is used heavily during initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.
- **Solution**:
  Shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

MFU (Most Frequently Used) page-replacement algorithm:

Working principle:

The page with the smallest count was probably just brought in and has yet to be used.

9.5 Allocation of Frames

- ✓ The absolute minimum number of frames that a process must be allocated is dependent on system architecture.
- ✓ The maximum number is defined by the amount of available physical memory.
- ✓ We must also allocate at least a minimum number of frames. One reason for this is 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.

Allocation Algorithms:

Equal Allocation:

We split m frames among n processes is to give everyone an equal share, m/n frames.

(For ex: if there are 93 frames and five processes, each process will get 18 frames. The three leftover frames can be used as a free-frame buffer pool).

Proportional Allocation:

- We can allocate available memory to each process according to its size.
- In both 1 & 2, the allocation may vary according to the multiprogramming level.
- If the multiprogramming level is increased, each process will lose some frames to

provide the memory needed for the new process.

- Conversely, if the multiprogramming level decreases, the frames that were allocated to the departed process can be spread over the remaining processes.

Global versus Local Allocation

Global Replacement	Local Replacement
Allows a process to a replacement frame from the set of all frames.	Each process selects from only its own set of allocated frames.
A process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it.	Number of frames allocated to a process does not change.
Disadvantage: A process cannot control its own page-fault rate.	Disadvantage: Might prevent a process by not making available to it other less used pages of memory.
Advantage: Results in greater system throughput.	

9.6 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.***

✓ If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - o low CPU utilization
  - o operating system thinks that it needs to increase the degree of multiprogramming
  - o another process added to the system.

✓ If the number of frames allocated to a low-priority process falls below the minimum number required, it must be suspended.

9.6.1 Cause of Thrashing

✓ Thrashing results in severe performance-problems (Figure 4.13).

✓ The thrashing phenomenon:
  - As processes keep faulting, they queue up for the paging device, so CPU utilization decreases
  - The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result.
  - The new process causes even more page-faults and a longer queue!

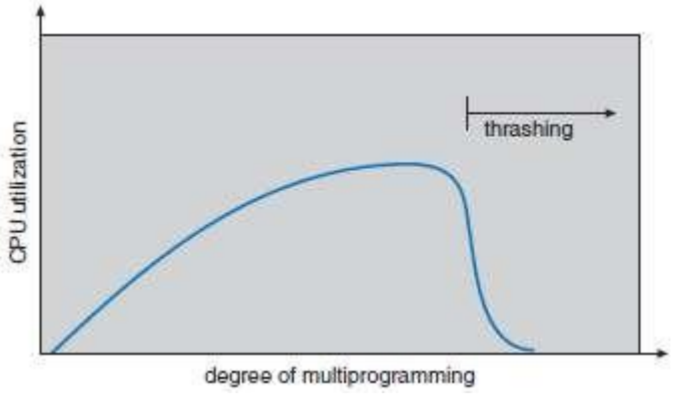


Figure:Thrashing

Methods to avoid thrashing:

- **Use Local Replacement**

  If one process starts thrashing, it cannot

  → steal frames from another process and

  → cause the latter to thrash as well.

- We must provide a process with as many frames as it needs. This approach defines the **locality model** of process execution.
  - Locality Model states that, As a process executes, it moves from locality to locality.
  - A locality is a set of pages that are actively used together.
  - A program may consist of several different localities, which may overlap.

<u>What is Belady's anomaly? Explain with an example.</u>

Solution:

Belady's Anomaly:

"On increasing the number of page frames, the no. of page faults do not necessarily decrease, they may also increase".

• Example: Consider the following reference string when number of frame used is 3 and 4: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

(i) FIFO with 3 frames:

Frames	1	2	3	4	1	2	5	1	2	3	4	5
1	1	2	3	4	1	2	5	5	5	3	4	4
2		1	2	3	4	1	2	2	2	5	3	3
3			1	2	3	4	1	1	1	2	5	5
No. of Page faults	√	√	√	√	√	√	√			√	√	

No. of page faults=9

(ii) FIFO with 4 frames:

Frames	1	2	3	4	1	2	5	1	2	3	4	5
1	1	2	3	4	4	4	5	1	2	3	4	5
2		1	2	3	3	3	4	5	1	2	3	4
3			1	2	2	2	3	4	5	1	2	3
4				1	1	1	2	3	4	5	1	2
No. of Page faults	√	√	√	√			√	√	√	√	√	√

No. of page faults=10

Conclusion: With 3 frames, No. of page faults=9. With 4 frames, No. of page faults=10.

Thus, Belady's anomaly has occurred, when no. of frames are increased from 3 to 4.
