ILAHIA COLLEGE OF ENGINEERING AND TECHNOLOGY

Computer Science and Engineering

OPERATING SYATEMS (S4 CSE)

Module V:

Memory Management: Main Memory – Swapping– Contiguous Memory allocation – Segmentation Paging – Demand paging

Main Memory

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be in main memory during execution. To improve CPU utilization computer must keep several processes in memory.

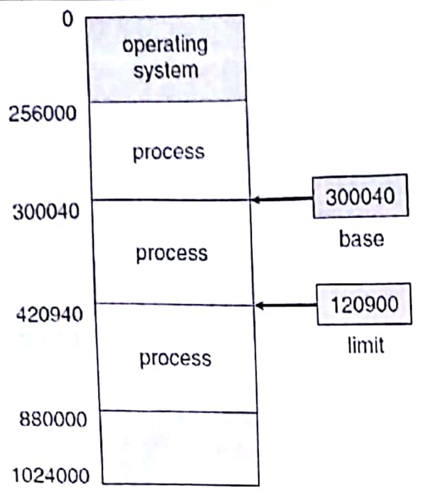
Memory is central to the operation of a modern computer system,. Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter.

Main memory and the registers built into the processor are the only storage that the CPU can access directly. Registers are generally accessible within one cycle of the CPU clock. So CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. But main memory access may take many cycles of the CPU clock to complete, in which case the processor normally needs to **stall (stop running)** , since it does not have the data required to complete the instruction that it is executing. The remedy is to add fast memory between the CPU and main memory called a **cache.**

To protect the operating system from access by user processes and to protect user processes from one another we need to make sure that each process has a separate memory space. There is a range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. To provide this protection two registers are used.

- Base register- holds the smallest legal physical memory address.
- Limit register- specifies the size of the range

For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940.



A base and a limit register define a logical address space.

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, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.

Protection of memory space is accomplished by comparing the address generated in user mode with the base and limit registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system. The user generated address should be greater than the base register and lower than base + limit register

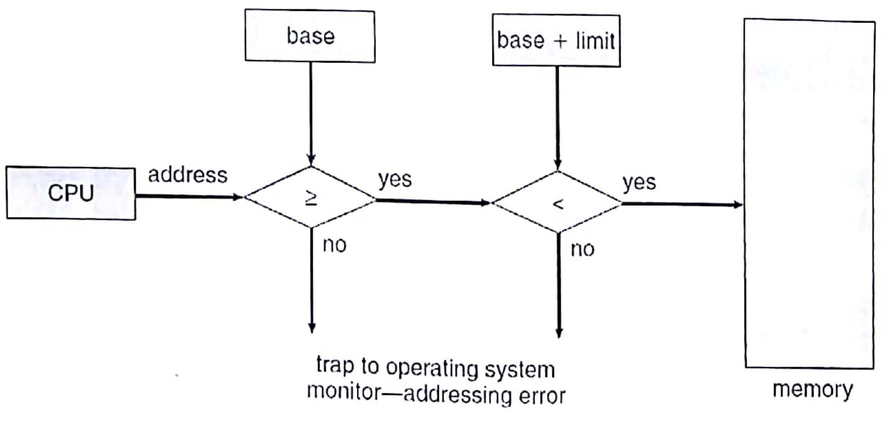


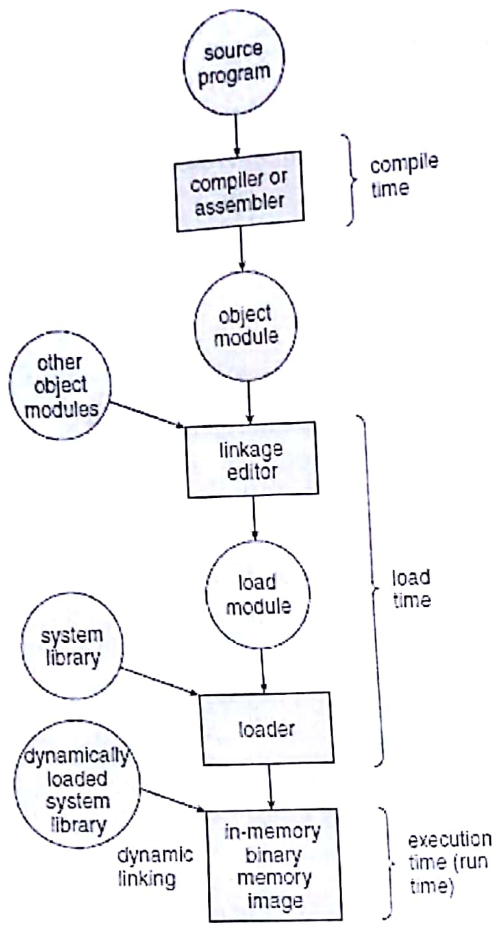
Figure 8.2 Hardware address protection with base and limit registers.

ddress Binding

A program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. As the process is executed, it accesses instructions and data from memory during its execution. Eventually, the process terminates, and its memory space is declared available.

Most systems allow a user process to reside in any part of the physical memory. The address space of the computer starts at 00000, the first address of the user process need not be 00000. Addresses in the source program are generally symbolic (such as *count). A* compiler will typically **bind** these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses (such as 74014). Each **binding is a mapping from one address space to another**. Binding of instructions and data to memory addresses can be done at any step along the way:

1. **Compile time:-** If you know at compile time where the process will reside in memory, then **absolute code** can be generated.
2. **Load time.** If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code.** In this case, final binding is delayed until load time
3. **Execution time.** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.



<u>**Logical Versus Physical Address Space**</u>

An address generated by the CPU is commonly referred to as a **logical address,** whereas an address seen by the memory unit is commonly referred to as a **physical address**

- The set of all logical addresses generated by a program is a **logical** address **space**
- The set of all physical addresses corresponding to these logical addresses is a **physical** address **space.**

The run- time mapping from logical to physical addresses is done by a hardware device called the **memory-management unit** (MMU).

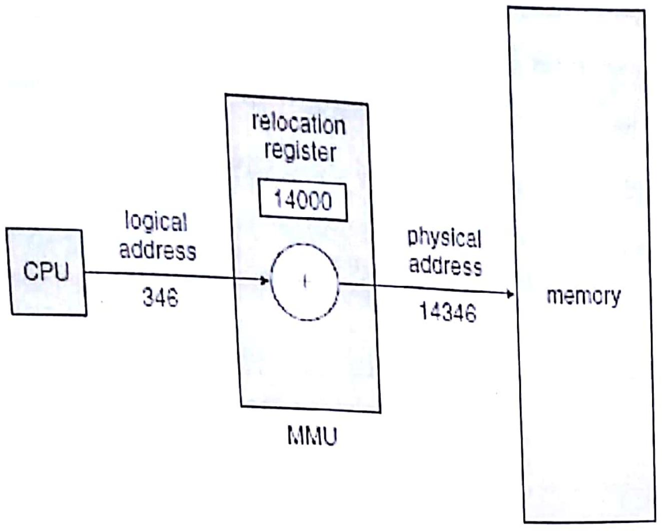


Figure 8.4 Dynamic relocation using a relocation register.

The base register is now called a **relocation register.** The value in the relocation register is *added* to every address generated by a user process at the time it is sent to memory

For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346. The user program deals with *logical* addresses. The memory-mapping hardware converts logical addresses into physical addresses.

**Logical addresses (in the range 0 to *max)* and physical addresses (in the range *R* + 0 to *R* + *max* for a base value *R).*

Dynamic Loading

The entire program and all data of a process must be in physical memory for the process to execute. The size of a process is thus limited to the size of physical memory. To obtain better memory utilization, we use **dynamic loading.**

With dynamic loading, **a routine is not loaded until it is called.** All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the relocatable linking loader is called to load the desired routine into memory. Then control is passed to the newly loaded routine.

The advantage of dynamic loading is that an unused routine is never loaded. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size may be large, the portion that is loaded may be much smaller.

Dynamic loading does not require special support from the operating system.

Swapping

A process must be in memory to be executed. A process can be **swapped** temporarily out of memory to a **backing store** and then brought into memory for continued execution.

Standard swapping involves moving processes between main memory and a backing store.

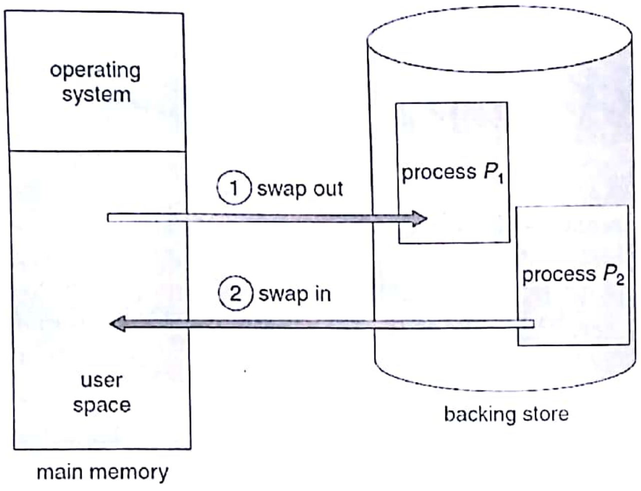


Figure 8.5 Swapping of two processes using a disk as a backing store.

The system maintains a **ready queue** consisting of all processes whose memory images are on the backing store . 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.

The context-switch time in such a swapping system is fairly high. The major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped.

Swapping is constrained by other factors also. If we want to swap a process, we must be sure that it is completely idle. Consider a process with pending i/o .ie, the process may be waiting for an i/o operation when
we want to swap out that process to free up memory. If we want to swap out process $P1$ and swap in process $P2$, the I/O operation might then attempt to use memory that now belongs to process $P2$.
There are two main solutions to this problem:

1. **Never swap a process with pending I/O**

2. Execute I/O operations only into operating-system buffers.

Transfers between operating-system buffers and process memory then occur when the process is swapped in and this **double buffering** itself adds overhead.

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called **rollout, roll in.**

Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible.

The memory is usually divided into two partitions:

1. for the resident operating system
2. for the user processes.

We usually want several user processes to reside in memory at the same time. **In contiguous memory allocation, each process is contained in a single contiguous section of memory.**

Memory mapping and protection

Memory mapping and protection is done with the help of **relocation 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 sent *to* memory

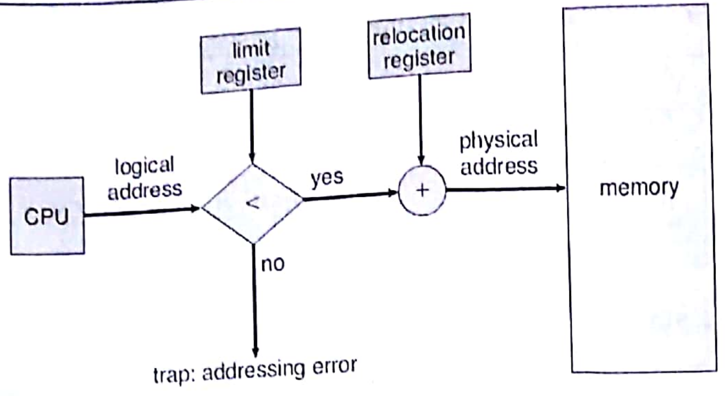


Figure 8.6 Hardware support for relocation and limit registers.

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. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

Memory Allocation

There are two types of memory allocation schemes:-

1. Fixed-sized partition schemes
2. Variable sized partition scheme

In fixed-sized partitions each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple partitioned 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 the variable partition scheme, all memory is available for user processes and is considered o large block of available memory, a **hole**. Eventually, memory contains a set of holes of various sizes. As processes enter the system, they are put into an input queue. The operating system 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 space, it is loaded into memory. When a process terminates, it releases its memory which the operating system 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.

At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

Dynamic storage allocation problem concerns how to satisfy a request of size n from a list of free holes. Solutions for this problems are

- **First fit**. Allocate the **first hole** that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best fit**. Allocate the **smallest hole** that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit**. Allocate the **largest hole**. Search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach

<u>**Memory fragmentation**</u>

It can be internal as well as external.

Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation.

- As processes are loaded and removed from memory, the free memory space is broken into little pieces. <u>**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.</u>
- In the worst case, it could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, it might be able to run several more processes.
- Statistical analysis of first fit, for instance, reveals that, even with some optimization, given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the **50-percent rule.**

The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. The memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**; memory that is internal to a partition.

Compaction is the solution for external fragmentation- shuffles the memory contents so as to place all free memory together in one large block. **Compaction is not always possible, If relocation is static and is done at assembly or load time, compaction cannot be done; compaction is possible *only* if relocation is dynamic and is done at execution time**

Another possible solution to the external-fragmentation problem is **paging and segmentation**

SEGMENTATION

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 therefore specifies each address by two quantities: **a segment name and an offset.**

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two tuple:*

<segment-number, offset>.

A C compiler might create separate segments for the following:

- The code
- Global variables
- The heap, from which memory is allocated
- The stacks used by each thread
- The standard C library

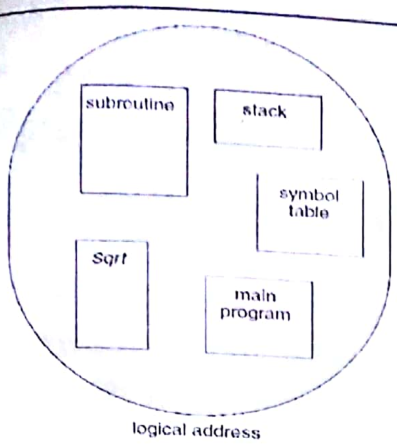


Figure 8.7 Programmer's view of a program.

Segmentation Hardware

We must define an implementation to map two dimensional user-defined addresses into one-dimensional physical addresses. This mapping is affected by a segmentation table.

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
- The segment limit specifies the **length of the segment**.

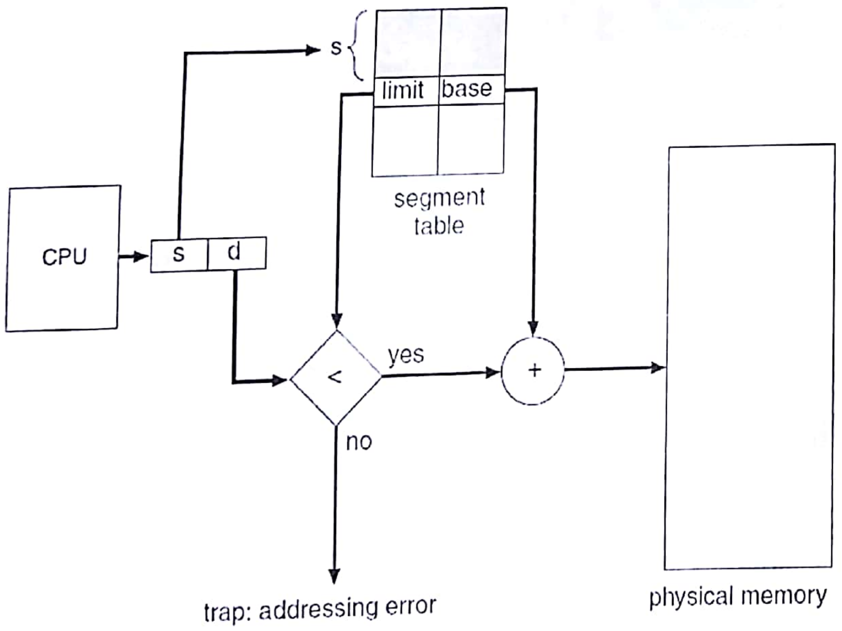
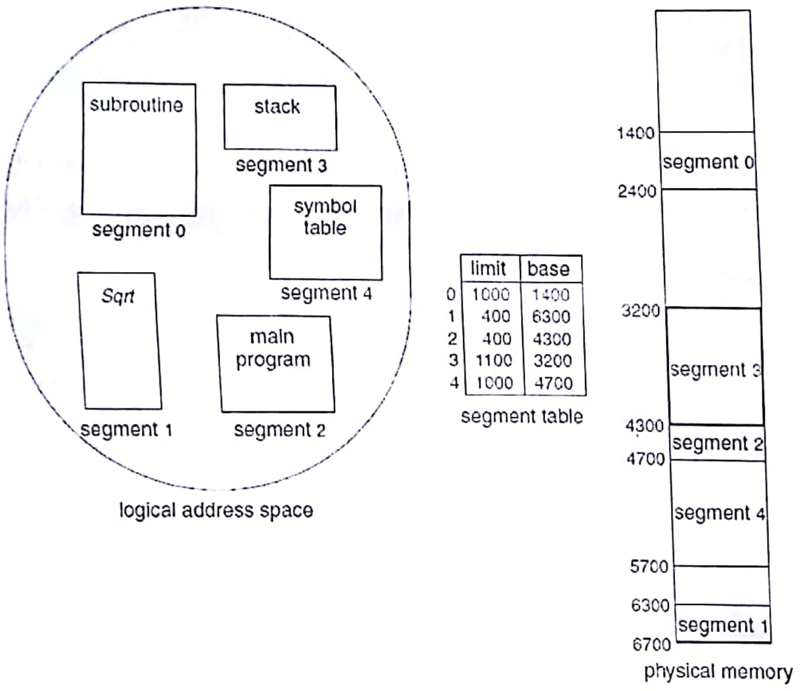


Figure 8.8 Segmentation hardware.

A logical address consists of two parts: a segment number, *s*, and an offset into that segment, 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.

As an example, consider the situation shown in Figure 8.9. 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.



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Figure 8.9 Example of segmentation.

Paging

Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous.

Paging avoids:

- External fragmentation and the need for compaction.

- It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store.

Basic Method

- Breaks physical memory into fixed-sized blocks called **frames** and logical memory 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 backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

The hardware support for paging is illustrated in Figure 8.10. Every address generated the CPU is divided into two parts: **a page number(p) and a offset(d).** . The page number is used as an index into a The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

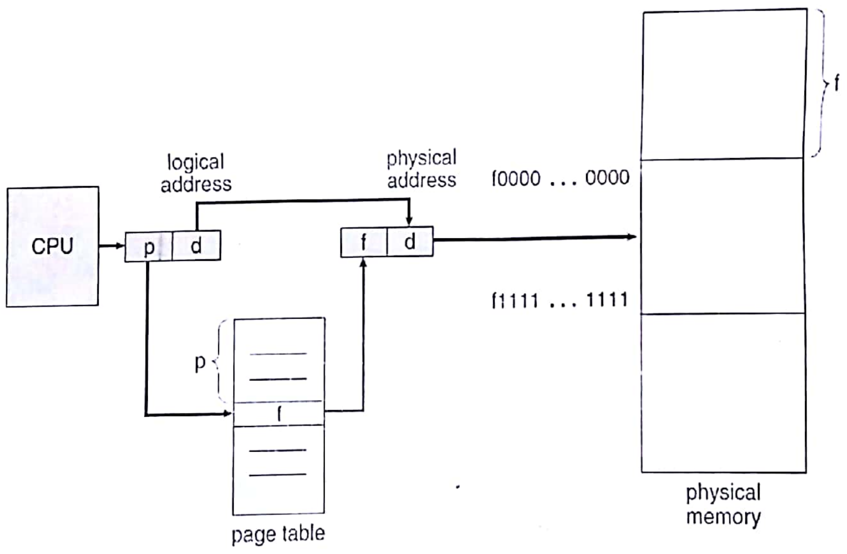


Figure 8.10 Paging hardware.

The paging model of memory is shown in Figure 8.11

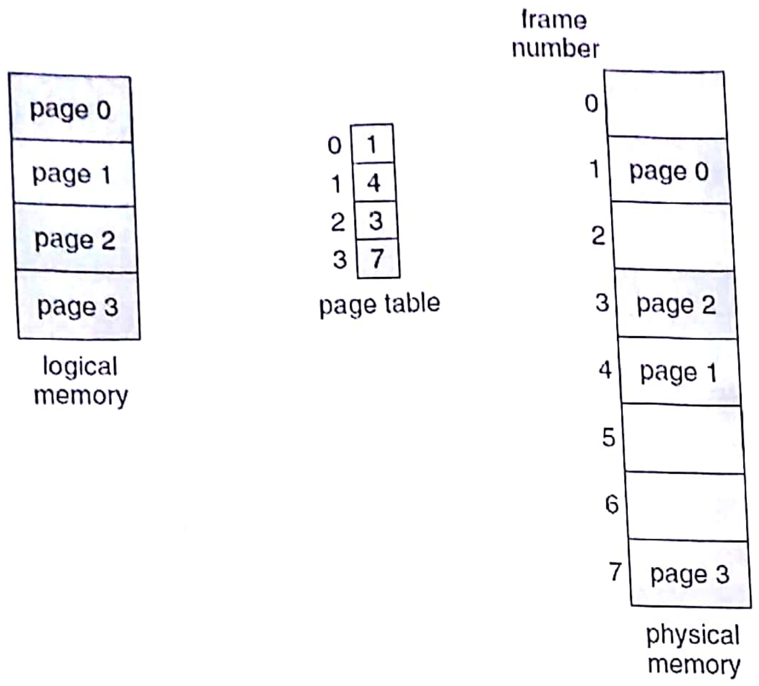
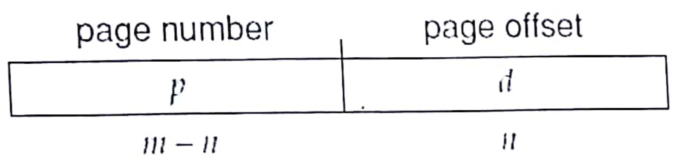


Figure 8.11 Paging model of logical and physical memory.

The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.

If the size of the logical address space is 2^m, and a page size is 2^n 1addressing units (bytes or words) then the high-order *m*- *n* bits of a logical address designate the page number, and the *n* low-order bits designate the page offset. Thus, the logical address is as follows:



where *p* is an index into the page table and *d* is the displacement within the page.

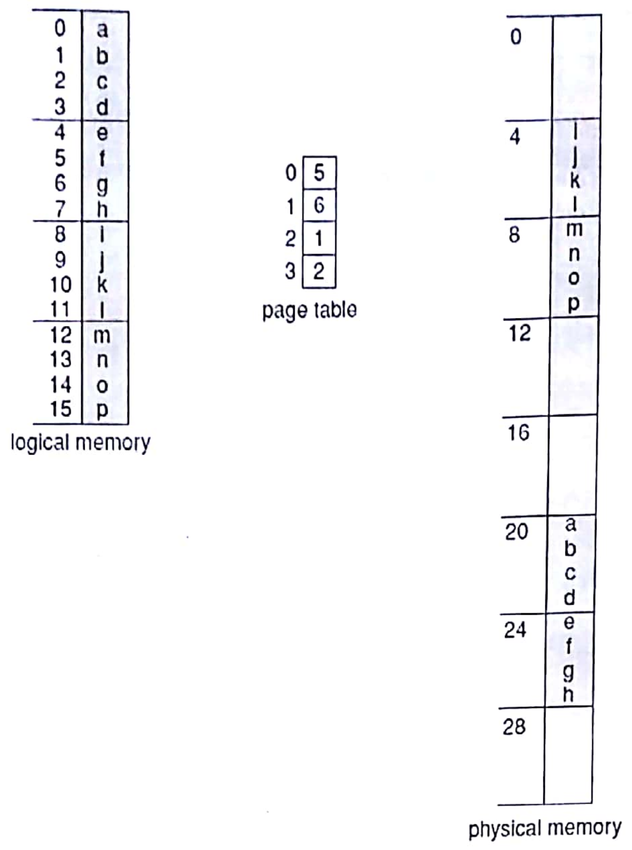


Figure 8.12 Paging example for a 32-byte memory with 4-byte pages.

For example, consider the memory in Figure 8.11. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory. 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 x 4) + 0). Logical address 3 (page 0, offset 3) maps to physical address 23 {- (5x4) + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6x4) + 0). Logical address 13 maps to physical address 9.

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 inJo 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 (Figure 8.13)

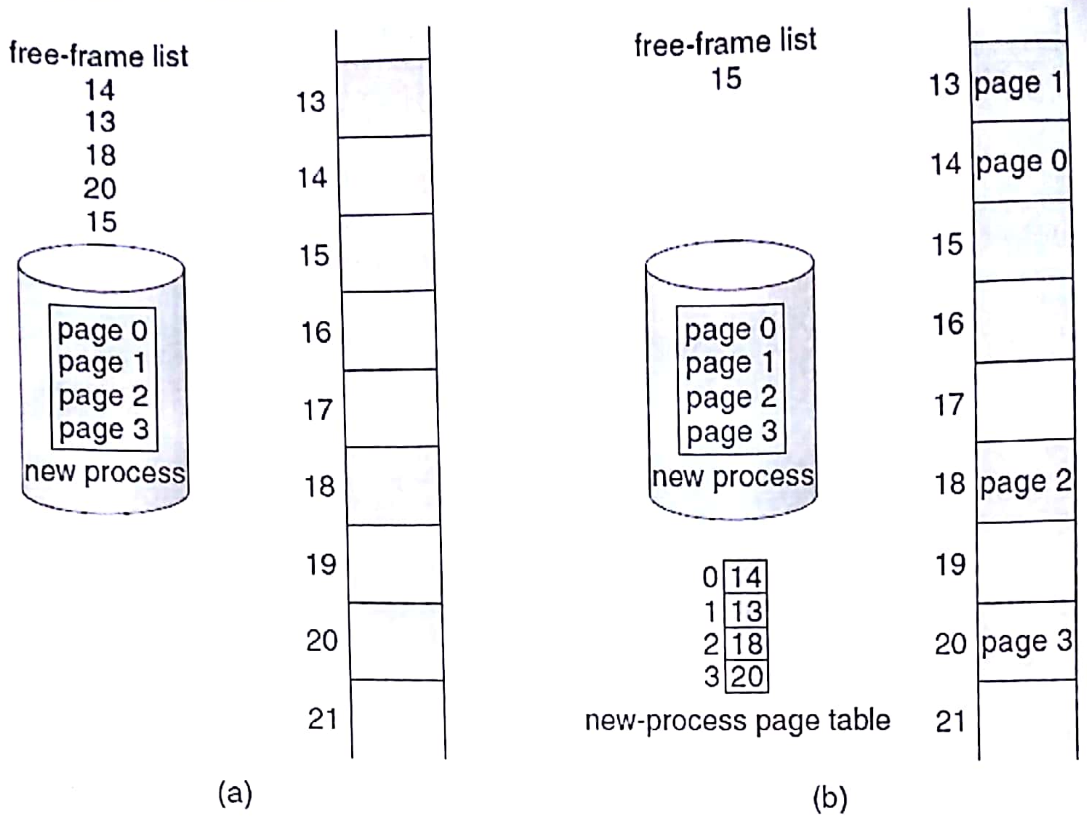


Figure 8.13 Free frames (a) before allocation and (b) after allocation.

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 operating system. Since the operating system is managing physical memory, it 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.** The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

Hardware Support

Each operating system has its own methods for storing page tables. Most allocate a page table for each process. A pointer to the page table is stored with in a register (like the instruction counter) in process control block.

The hardware implementation of the page table can be done in several ways .

1. In the simplest case, the page table is implemented as a set of dedicated **registers.**

The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries).

2. If w the page table is very large the use of registers is not feasible. So the page table is kept in main memory, and a **page table base register(PTBR)** points to the page table.

The problem with this approach is the 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. This task requires a memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, *two* memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances.

The standard solution to this problem is to use a special, small, fast lookup hardware cache, called **a translation lookaside buffer**(TLB)

* TLB is associative, high-speed memory

* Each entry in the TLB consists of two parts: **a key (or tag) and a value.**

The TLB contains only a few of the page-table entries. 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 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 (Figure 8.14).We add the page number and frame number to the TLB, so that they will be found quickly on the next reference.

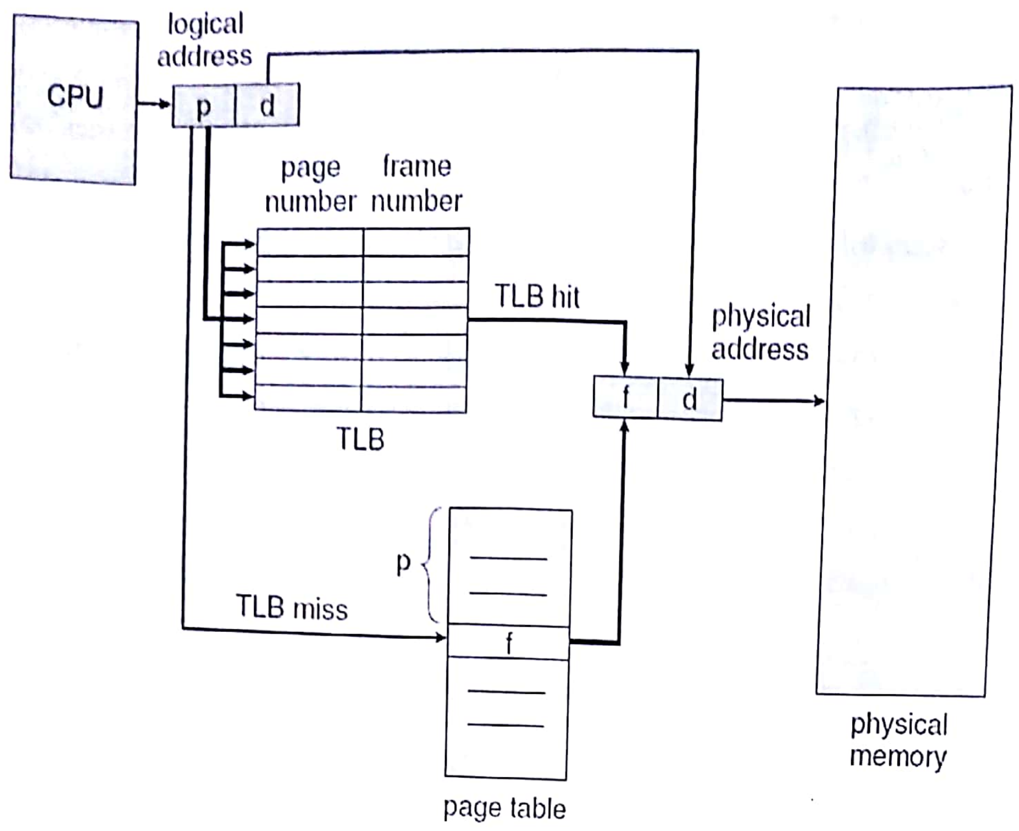


Figure 8.14 Paging hardware with TLB.

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

An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time. If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds.

To find the **effective memory-access time,** we weight each case by its probability:

effective access time = 0.80 x 120 + 0.20 x 220 = 140 nanoseconds.

In this example, we suffer a 40-percent slowdown in memory-access time (from 100 to 140 nanoseconds).

For a 98-percent hit ratio, we have

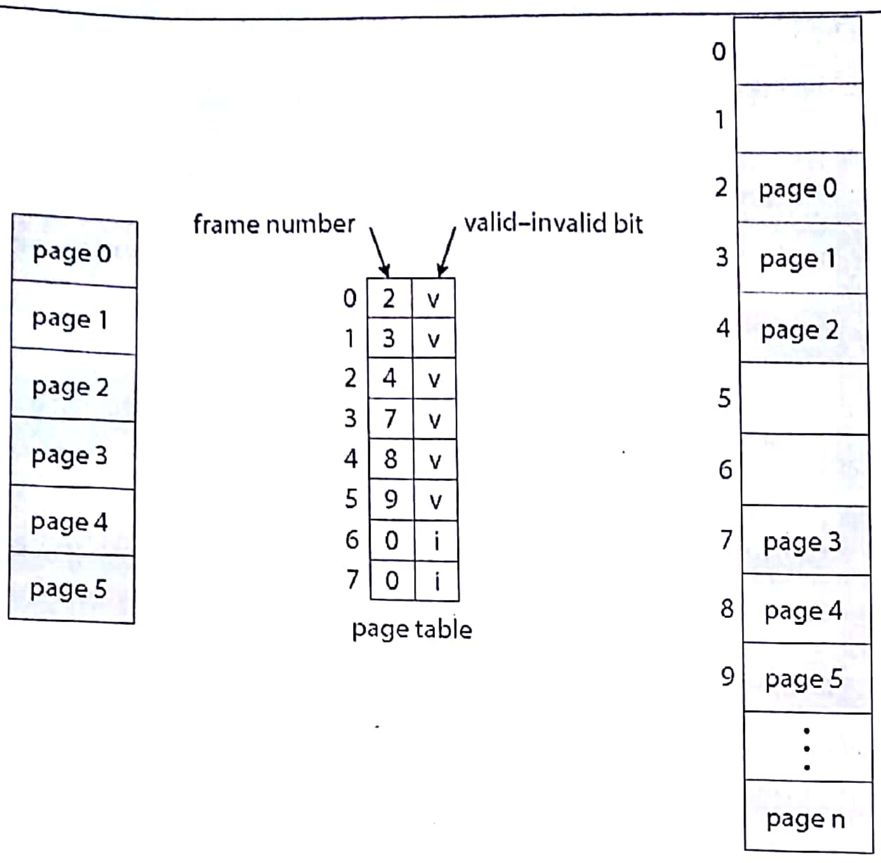
effective access time = 0.98 x 120 + 0.02 x 220 = 122 nanoseconds.

This increased hit rate produces only a 22 percent slowdown in access time.

Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame. These bits are kept in the page table.

- **One bit can define a page to be read-write or read-only.**
  - This protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system.
- **One additional bit** is generally attached to each entry in the page table: a **valid-invalid bit.**
  - When this bit is set to "**valid,**" the associated page is in the process's logical address space or present in the main memory and is thus a legal (or valid) page.
  - When the bit is set to "**invalid,**" the page is not in the process's logical address space or not present in the main memory. Illegal addresses are trapped by use of the valid -invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.

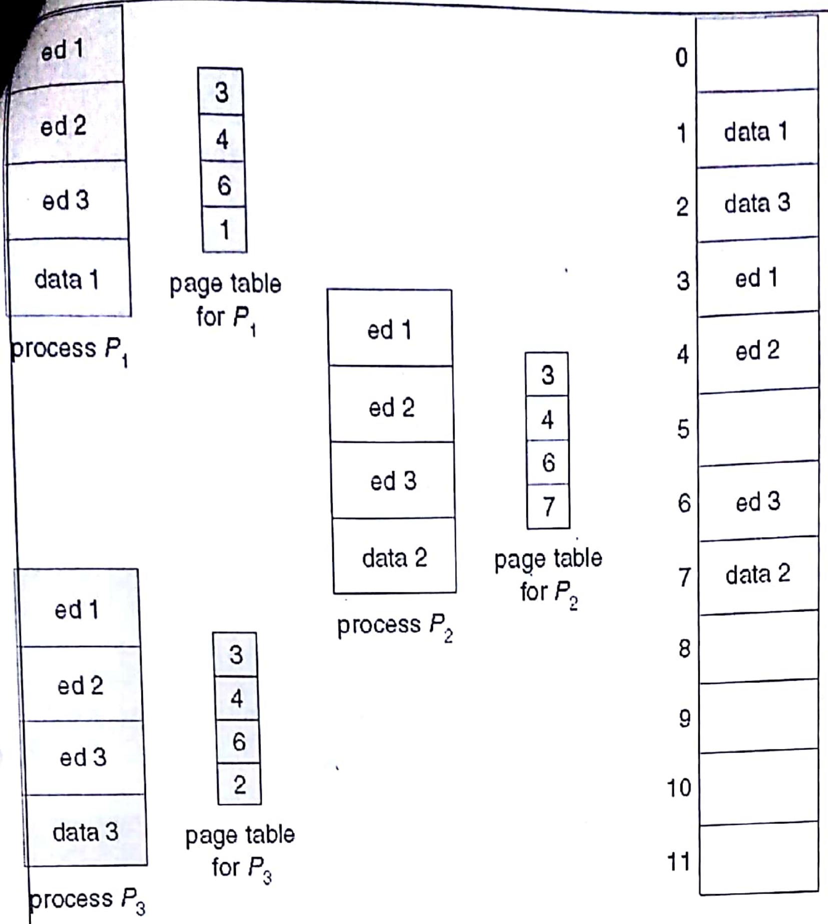


We have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, we get the situation shown in above Figure Addresses in pages 0,1, 2,3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, will find that the valid-invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

Shared Pages

An advantage of paging is the possibility of **sharing common code.**

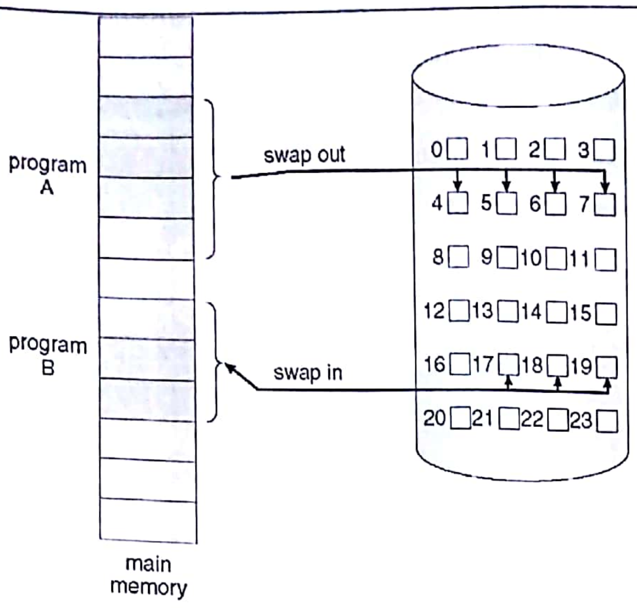
Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. Here we see a three-page editor ed1,ed2 and ed3 being shared among three processes. Each process has its own data page.



Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames

Virtual memory

- 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.
- Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. This technique frees programmers from the concerns of memory-storage limitations.
- Virtual memory also allows processes to share files easily and to implement shared memory



- Loading the entire program in physical memory at program execution time is not so good because the user doesn't *need* the entire program in memory at a time.
- An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging.**

With demand-paged virtual memory, pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory.

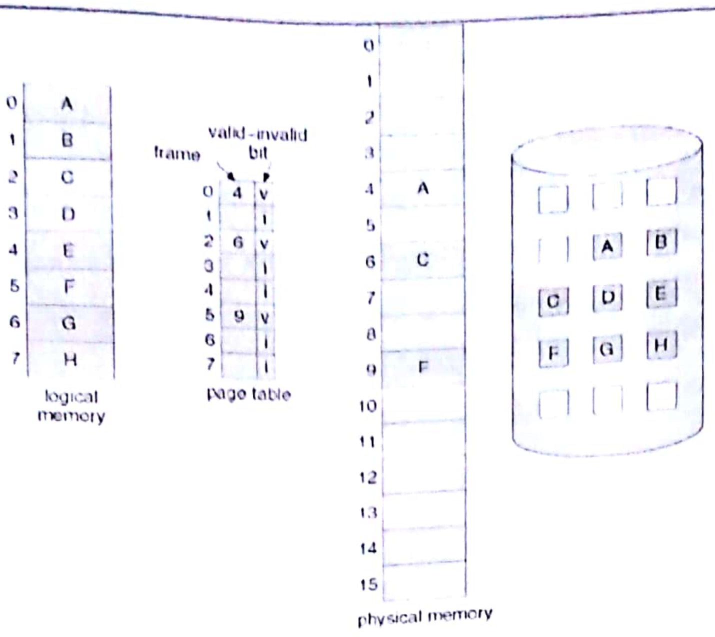
A demand-paging system is similar to a paging system with swapping.

But here it uses a **lazy swapper**; where it never swaps a page into memory unless that page will be needed.In Demand paging, **pager** is more suitable than swapper .

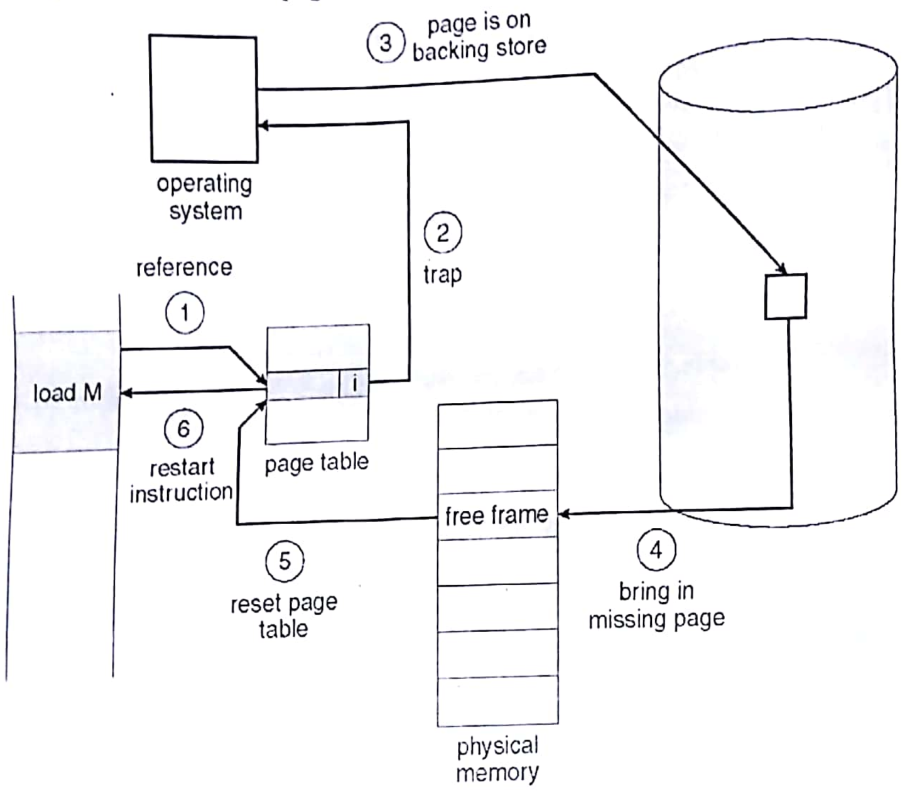
Basic Concepts

With swapping, pager guesses which pages will be used before swapping out again Instead, pager brings in only those pages into memory. Some form of hardware support is needed to distinguish between the pages that are in memory and the pages that are on the disk.

- The valid -invalid bit scheme can be used for this purpose.
  - With each page table entry a valid–invalid bit is associated (v ⇒ in-memory, i ⇒ not-in- memory)
  - Initially valid–invalid bit is set to i on all entries
  - During MMU address translation, if valid–invalid bit in page table entry is i then a page fault occurs.



Page table when some pages are not in main memory



Steps in handling a page fault.

The procedure for handling this page fault is

1. 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, terminate the process. If it was valid, but have not yet

brought in that page, page it in.

3. Find a free frame.
4. Schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. Restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

In the extreme case, we can start executing a process with no pages in memory. When the operating system sets the instruction pointer to the first 404 Chapter 9 Virtual Memory 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

➢ Theoretically, some programs could access several new pages of memory with each instruction execution, possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance. Locality of reference can be used for such situations.

➢ Hardware support needed for demand paging

• **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

Performance of Demand Paging

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

➢ the effective access time is

Effective access time= (1 - p) x ma + p x page fault time.

➢ where,

- p be the probability of a page fault (0 <=p <=1).
- ma be the memory-access time

Page Replacement

- Page replacement occurs when:
  - A page fault occurs and we need to bring the desired page into memory
  - There are NO free frames.
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – decide which frame to free
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Basic Page Replacement

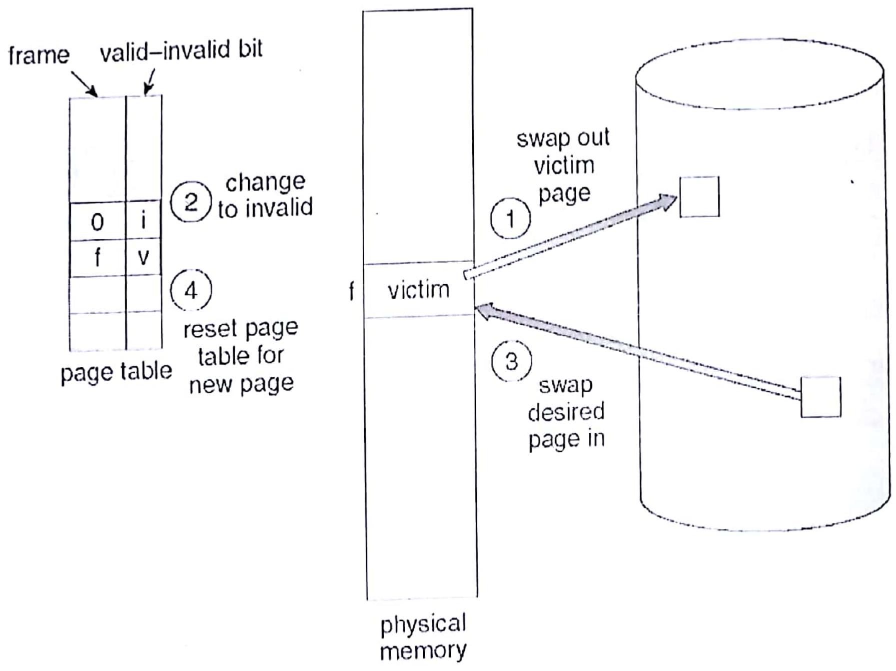
Page replacement takes the following approach.

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.

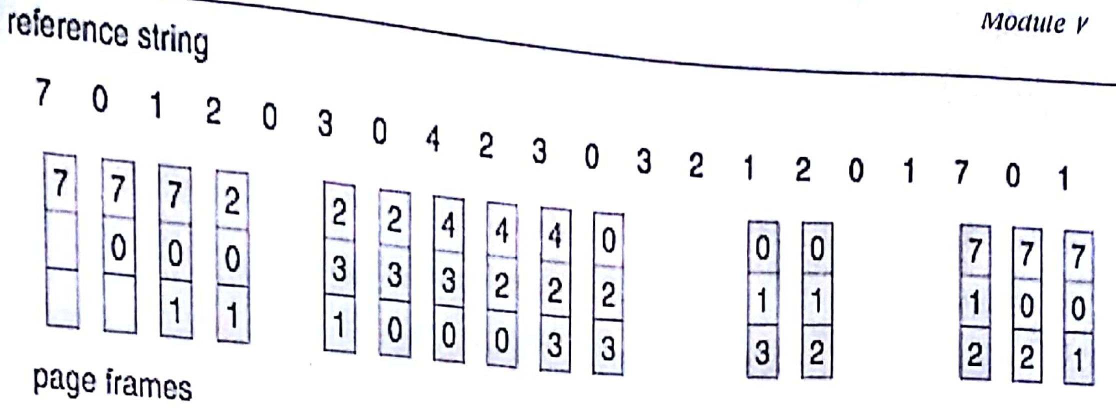


4. Restart the user process.

- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory
- large virtual memory can be provided on a smaller physical memory.
- two major problems to implement demand paging
  - Frame allocation algorithm
  - page replacement algorithm
- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available

1. First-In-First-Out (FIFO) Algorithm

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



➢ **Belady's Anomaly**-for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases.

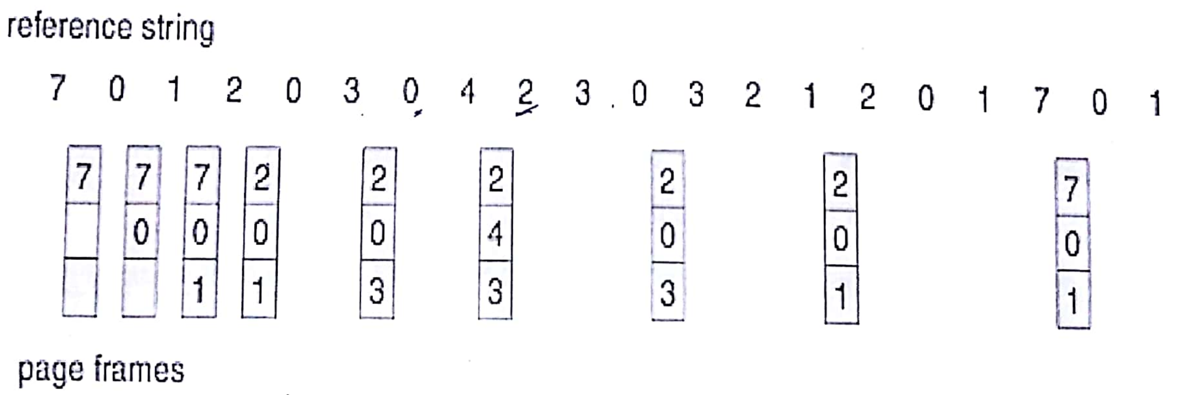
- Example:For the reference string 1,2,3,4,1,2,5,1,2,3,4,5

  page fault is 9 with frames 3 and page fault is 10 with frames 4.

2. Optimal Page Replacement

➢ Replace page that will not be used for longest period of time

➢ **It has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly-hence known as OPT or MIN.**



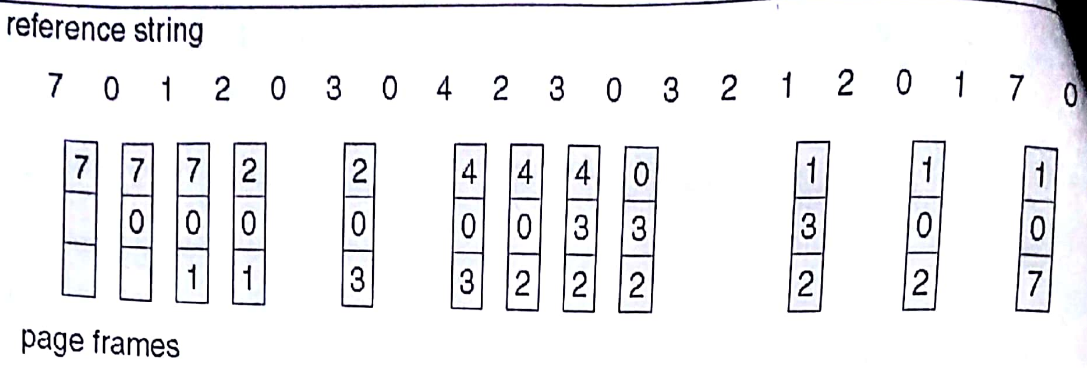
➢ It is difficult to implement, because it requires future knowledge of the reference string.

3. LRU Page Replacement

➢ Use past knowledge rather than future

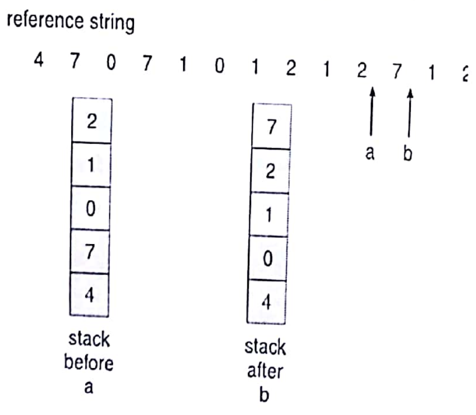
➢ Replace page that has not been used in the most amount of time

➢ Associate time of last use with each page



12 faults – better than FIFO but worse than OPT

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



Use of a stack to record the most recent page references.

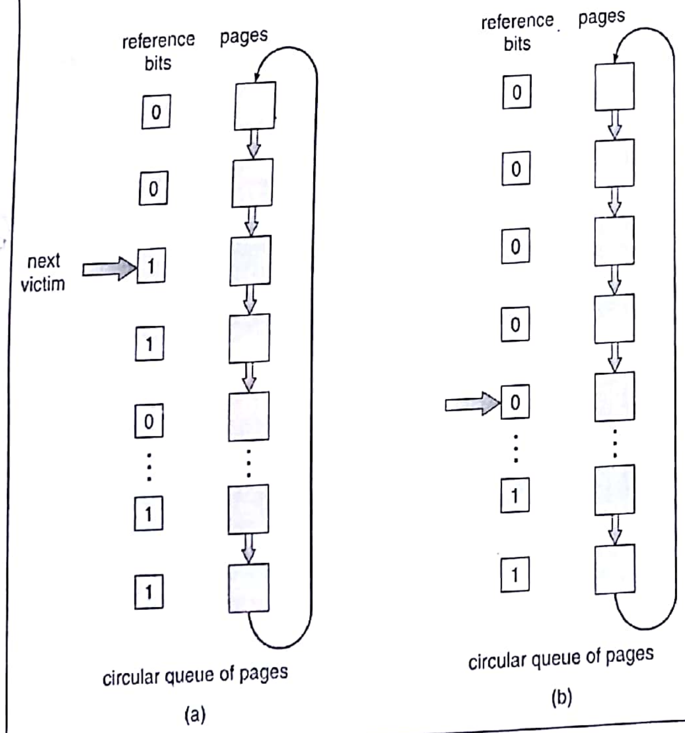
4. LRU-Approximation Page Replacement **Additional Reference-Bits Algorithm**

- With each page associate a bit, initially = 0
- When page is referenced bit set to 1
- Replace any with reference bit = 0 (if one exists)

Second-Chance Algorithm

- The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, inspect its reference bit.
- If the value is 0, proceed to replace this page; but if the reference bit is set to 1, give the page a second chance and move on to select the next FIFO page.
- When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced.
- One way to implement the second-chance algorithm (the clock algorithm) is as a circular queue. A pointer indicates which page is to be replaced next.

pointer advances until it finds a page with a 0 reference bit



Second-chance (clock) page-replacement algorithm

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)

1. (0, 0) neither recently used not modified – best page to replace
2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
3. (1, 0) recently used but clean – probably will be used again soon
4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
    - Might need to search circular queue several times

1. Difference b/w segmentation & Paging
2. thrashing
3. state the advantages of segmented paging over pure segmentation
4. when does a Belady's anomaly occur.
5. Pure demand paging