UNIT-IV
Memory Management

Memory is central to the operation of a modern computer system. Memory consists of a large array of bytes, each with its own address.

A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory.

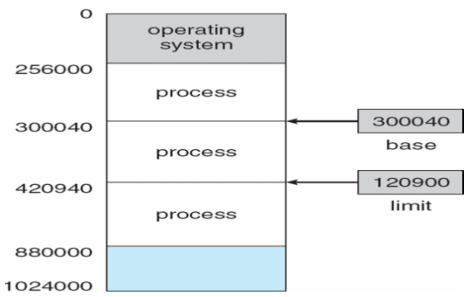
1. Basic Hardware

Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.

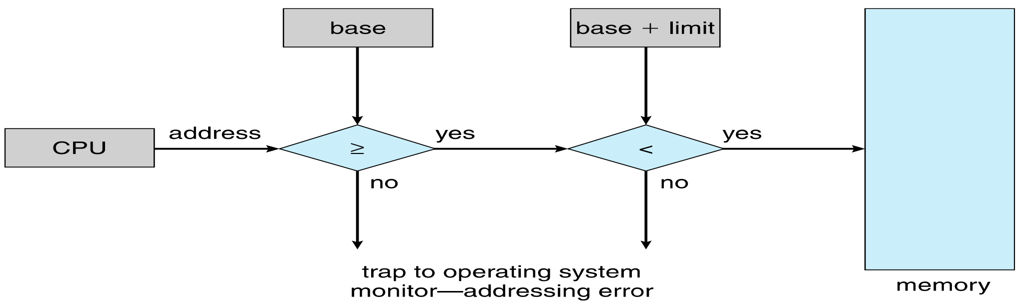
Protecting user processes from one another:

We first need to make sure that each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit

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 300040 and the limit register is 120900, and then the program can legally access all addresses from 300040 through 420939 (inclusive).



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.



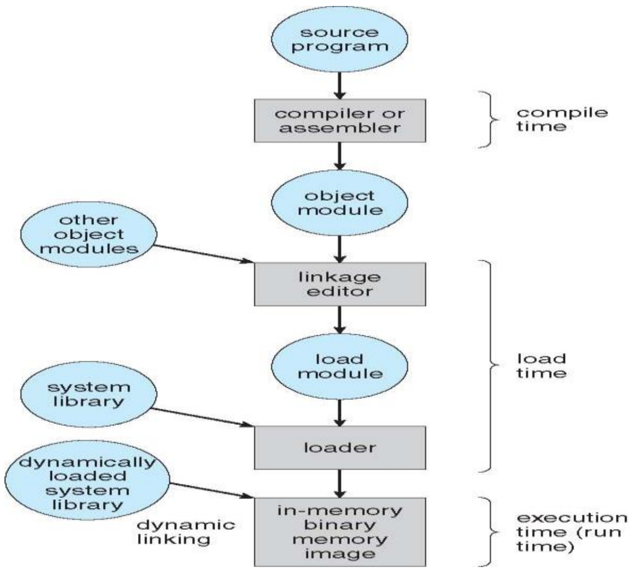
The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers.

2. Address Binding

Usually, 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. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the **input queue**.

Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

- **Compile time**. If you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know that a user process will reside starting at location *R*, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are bound at compile time.
- **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. If the starting addresses changes, we need only reload the user code to incorporate this changed value.
- **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. Most general-purpose operating systems use this method.



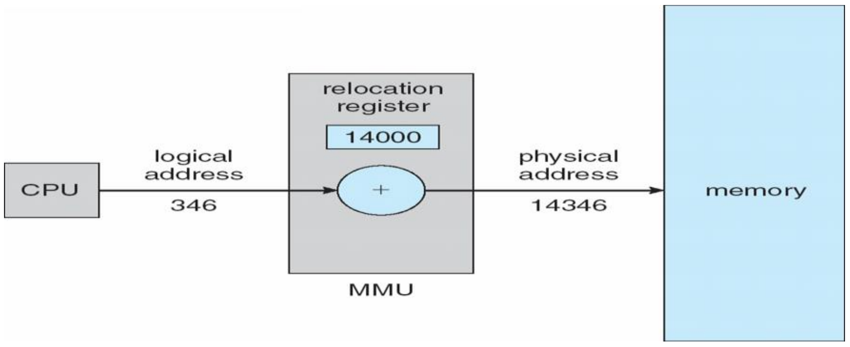
3. Logical Versus Physical Address Space

- An address generated by the CPU is commonly referred to as a **logical address or virtual address**.
- An address seen by the memory unit—that is, the one loaded into the **memory-address register** of the memory—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.**

Memory-Management Unit (MMU)

- The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**.
- 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 the address 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 location14000; an access to location346 is mapped to location 14346.



Swapping

What is Swapping?

A process must be in memory to be executed. A process, however, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution. Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

1. Standard Swapping

Standard swapping involves moving processes between main memory and a backing store. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.

Ready Queue: The system maintains a **ready queue** consisting of all processes whose memory images are on the backing store or in memory and are ready to run.

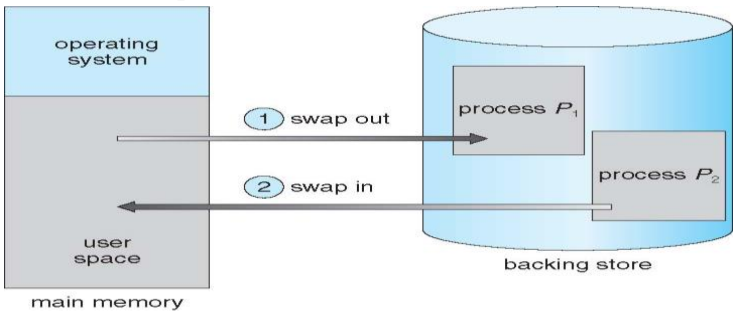
Dispatcher: Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.

Factors

- The context-switch time in such a swapping system is fairly high.
- The total transfer time is directly proportional to the amount of memory swapped.
- If we want to swap a process, we must be sure that it is completely idle.

Standard swapping in modern operating systems

- Standard swapping is not used in modern operating systems. It requires too much swapping time and provides too little execution time to be a reasonable memory-management solution.
- Modified versions of swapping, however, are found on many systems, including UNIX, Linux, and Windows.
- In one common variation, swapping is normally disabled but will start if the amount of free memory falls below a threshold amount. Swapping is halted when the amount of free memory increases.
- Another variation involves swapping portions of processes—rather than entire processes—to decrease swap time.



2. **Swapping on Mobile Systems**

   Mobile systems typically do not support swapping in any form.

Reasons

- Mobile devices generally use flash memory rather than hard disks. The resulting space constraints avoid swapping.
- The limited number of writes that flash memory can tolerate before it becomes unreliable
- The poor throughput between main memory and flash memory in these devices.

Mechanisms instead of Swapping

- **Apple's iOS** *asks* applications to voluntarily relinquish allocated memory. Any applications that fail to free up sufficient memory may be terminated by the operating system.
- **Android** does not support swapping and adopts a strategy similar to that used by iOS. It may terminate a process if insufficient free memory is available. However, before terminating a process, Android writes its **application state** to flash memory so that it can be quickly restarted.

Contiguous Memory Allocation

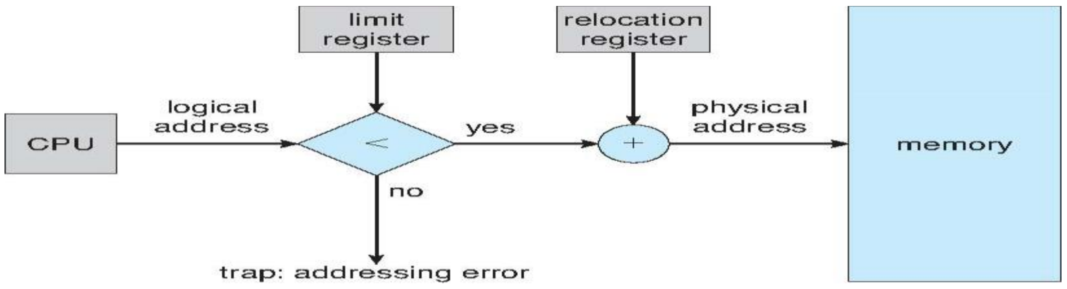
We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process.

1. Memory Protection

We can prevent a process from accessing memory it does not own by combining two ideas. If we have a system with a relocation register, together with a limit register, we accomplish our goal.

Process

- The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600).
- Each logical address must fall within the range specified by 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.
- 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.



2. Memory Allocation

Methods for Memory Allocation

a. Fixed-Sized Partitions

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized **partitions**.
- Each partition may contain exactly one process.
- In this **multiple partition 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.
- This method was originally used by the IBM OS/360 operating system (called MFT) but is no longer in use.

b. Variable Sized -Partition

- In the **variable-partition** scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**.
- 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 (Memory Allocation Techniques)

This concerns how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem. The **first-fit**, **best-fit**, and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

- **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. Again, we must 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.

Comparison:

- First fit and Best fit are better than Worst fit in terms of decreasing time and storage utilization.
- Neither first fit nor Best fit is clearly better than the other in terms of storage utilization, but First fit is generally faster.

3. Fragmentation

Memory fragmentation can be internal as well as external.

a. Internal Fragmentation

- The overhead to keep track of this hole will be substantially larger than the hole itself. 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.
- With this approach, the memory allocated to a process may be slightly larger than the requested memory.
- The difference between these two numbers is **internal fragmentation**—unused memory that is internal to a partition.

b. External Fragmentation

- 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.

- 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.

50-percent rule: Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. 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**.

Solution to External Fragmentation

a. Compaction

- The goal is to shuffle the memory contents so as to place all free memory together in one large block.
- Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time.
- The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.

b. Noncontiguous logical address space

- This permits the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available.
- Two complementary techniques achieve this solution: segmentation and paging.

Segmentation

Dealing with memory in terms of its physical properties is inconvenient to both the operating system and the programmer. What if the hardware could provide a memory mechanism that mapped the programmer's view to the actual physical memory? The system would have more freedom to manage memory, while the programmer would have a more natural programming environment. Segmentation provides such a mechanism.

1. Basic Method

Segmentation is a memory-management scheme that supports the programmer view of memory. A logical address space is a collection of variable sized segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The programmer therefore specifies each address by two quantities: a segment name and an offset.
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>.

Example of Segments

When a program is compiled, the compiler automatically constructs segments reflecting the input program.

A C compiler might create separate segments for the following:

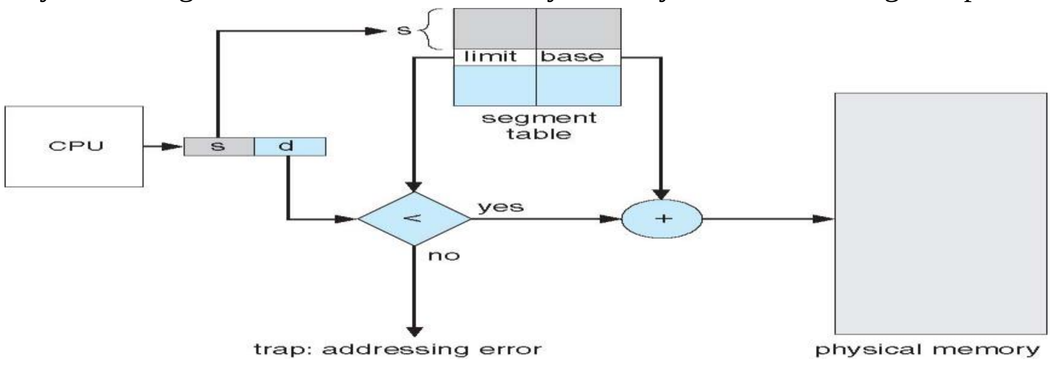
1. The code
2. Global variables
3. The heap, from which memory is allocated
4. The stacks used by each thread
5. The standard C library\

2. Segmentation Hardware

Although the programmer can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one dimensional sequence of bytes. Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses. This mapping is affected by a **segment table**. Each entry in the segment table has a **segment base** and a **segment limit**.

- **Segment base:** The segment base contains the starting physical address where the segment resides in memory.
- **Segment limit**: The segment limit specifies the length of the segment.

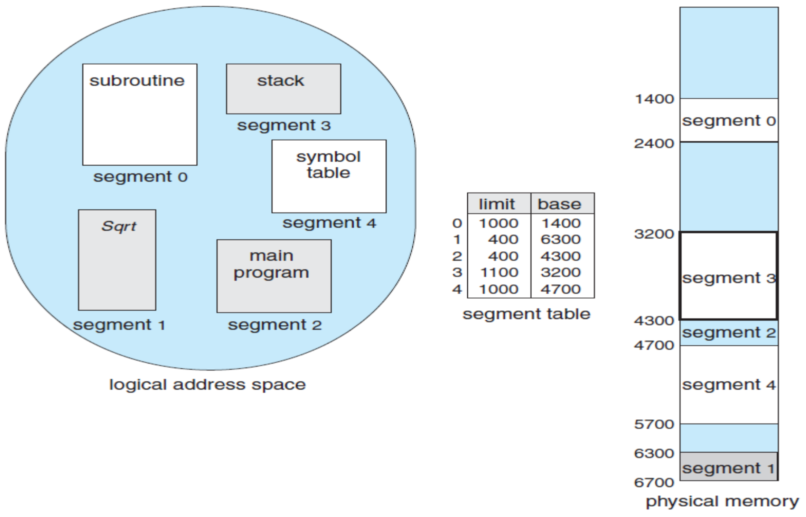
A logical address consists of two parts: a segment number, *s*, and an offset into that segment, *d*. The segment number are used as an index to the segment table. The offset *d* of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). 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–limit register pairs.



Example:

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).

Consider, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.



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

Paging

Paging is another memory-management scheme that offers physical address space of a process to be non-contiguous. Paging also avoids external fragmentation and the need for compaction, whereas segmentation does not. Because of its advantages, paging in its various forms is used in most operating systems, from mainframes to smart phones.

1. Basic Method

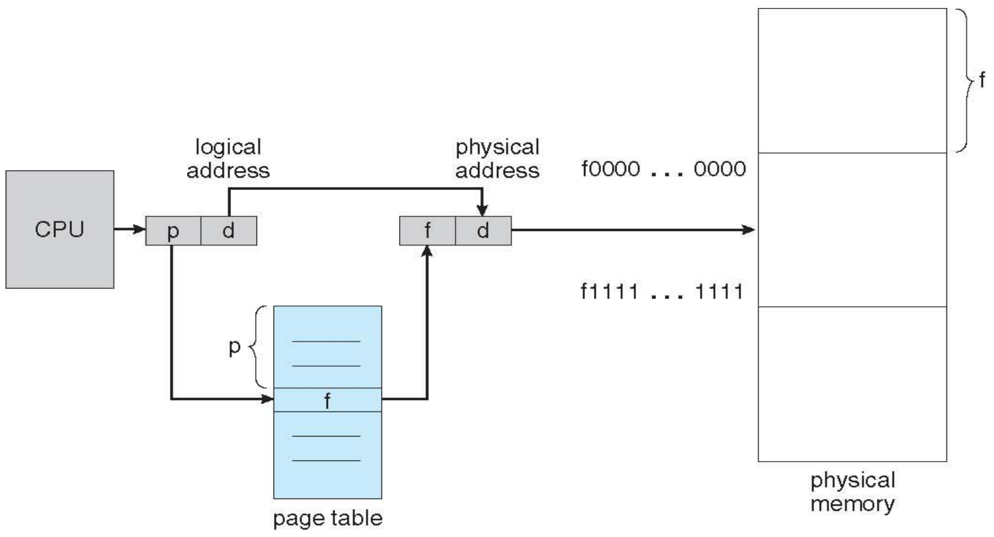
- **Frames**: Paging involves breaking physical memory into fixed-sized blocks called **frames.**
- **Pages:** Breaking logical memory into blocks of the same size called **pages**.

When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store).

Hardware Support for Paging

Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d)**.

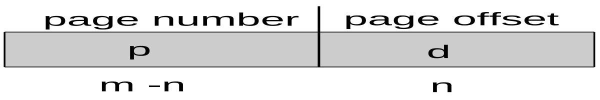
- **Page Table:** The page number is used as an index into a **page table**. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.
- **Frame Table:** Since the 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.



Defining of Page Size

The page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, varying between 512 bytes and 1 GB 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.

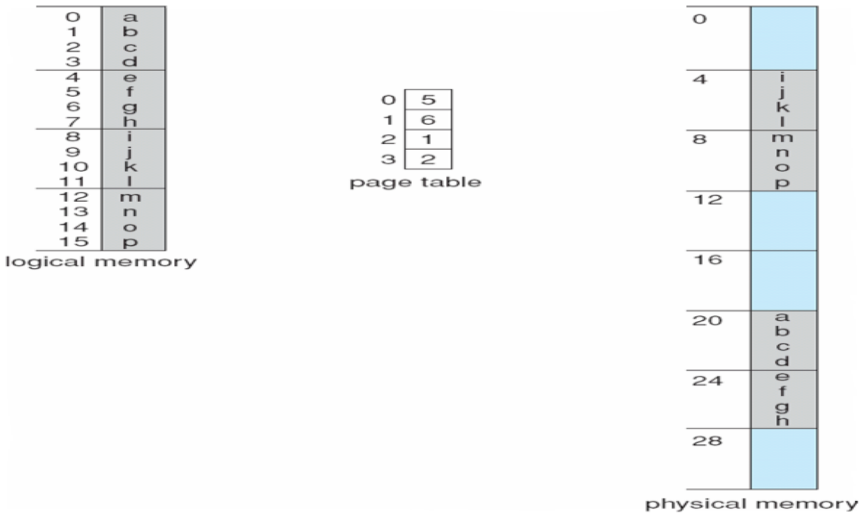
If the size of the logical address space is 2^m, and a page size is 2^n bytes, then the high-order $m-n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:



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

Example

Here, in the logical address, *n*= 2 and *m* = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages).Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0]. Logical address 13 maps to physical address 9.



2. Hardware Support

Methods for storing page table: Each operating system has its own methods for storing page tables.

a) Some allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table.

b) Other operating systems provide one or at most a few page tables, which decreases the overhead involved when processes are context-switched.

Hardware Implementation of the Page Table

Registers

- In the simplest case, the page table is implemented as a set of dedicated **registers**. These registers should be built with very high-speed logic to make the paging-address translation efficient.
- Every access to memory must go through the paging map, so efficiency is a major consideration.
- The CPU dispatcher reloads these registers, just as it reloads the other registers. Instructions to load or modify the page-table registers are, of course, privileged, so that only the operating system can change the memory map. The use of registers for the page table is satisfactory if the page table is reasonably small.
- The DEC PDP-11 is an example.

Page-Table Base Register (PTBR)

Most contemporary computers, allow the page table to be very large (for example, 1 million entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a **page-table base register (PTBR)** points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

Problem

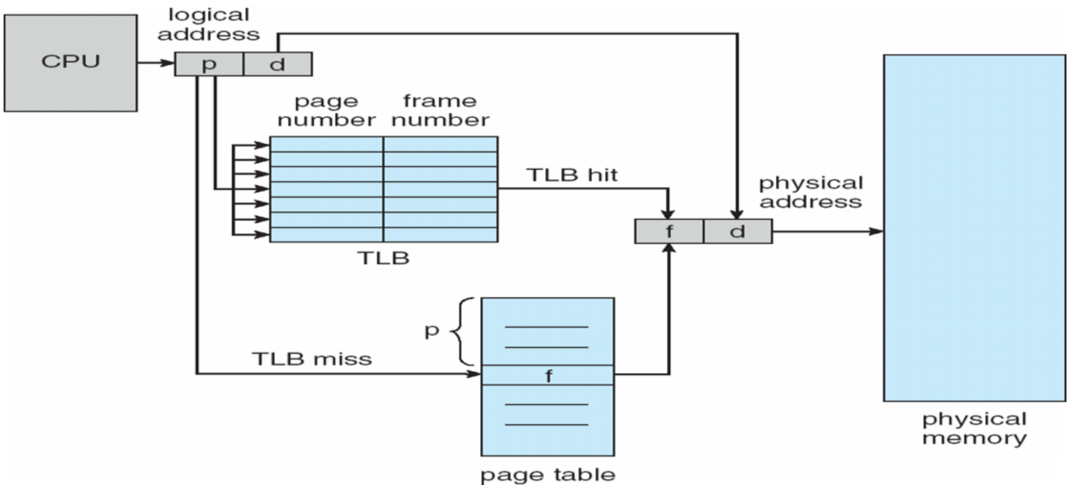
The problem with this approach is the time required to access a user memory location. If we want to access location *i*, we must first index into the page table, using the value in the PTBR offset by the page number for *i*. This task requires a memory access. It provides us with the frame number, which 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.

Solution: Translation Look-Aside Buffer (TLB).

The standard solution to this problem is to use a special, small, fast lookup hardware cache called a **translation look-aside buffer (TLB)**. The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value.

Working of translation look-aside buffer (TLB):

- The TLB is used with page tables in the following way. 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, its frame number is immediately available and is used to access memory.
- If the page number is not in the TLB (known as a **TLB miss**), a memory reference to the page table must be made.
- Depending on the CPU, this may be done automatically in hardware or via an interrupt to the operating system.
- When the frame number is obtained, we can use it to access memory. 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, an existing entry must be selected for replacement.



Address-Space Identifiers (ASIDs)

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process.
- When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match the attempt is treated as a TLB miss.

Hit ratio: The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**

3. Protection

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

Read–Write or Read-Only Bit

- One bit can define a page to be read–write or read-only. Every reference to memory goes through the page table to find the correct frame number.
- At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page.

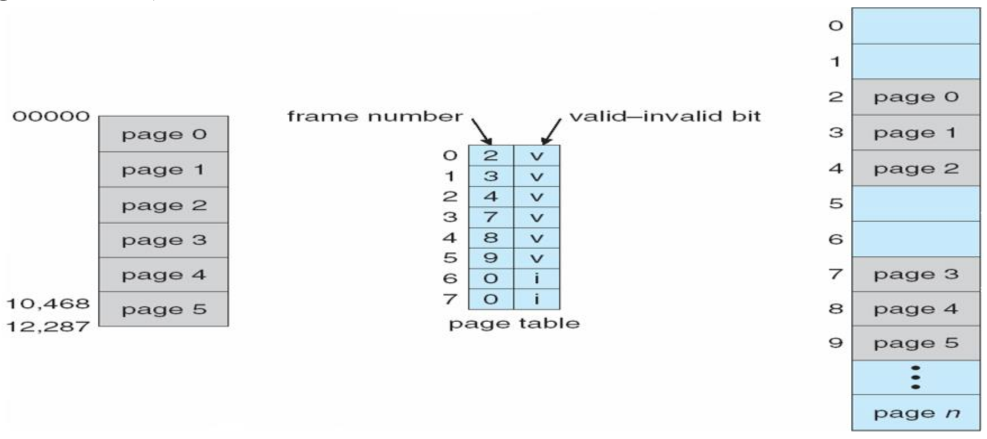
- An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).
- We can easily expand this approach to provide a finer level of protection.
- We can create hardware to provide read-only, read–write, or execute-only protection; or, by providing separate protection bits for each kind of access, we can allow any combination of these accesses. Illegal attempts will be trapped to the operating system.

Valid–Invalid Bit

- 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 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. 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.

Example

- Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468.
- Given a page size of 2 KB, 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).



Hardware for Protection: Page-Table Length Register (PTLR)

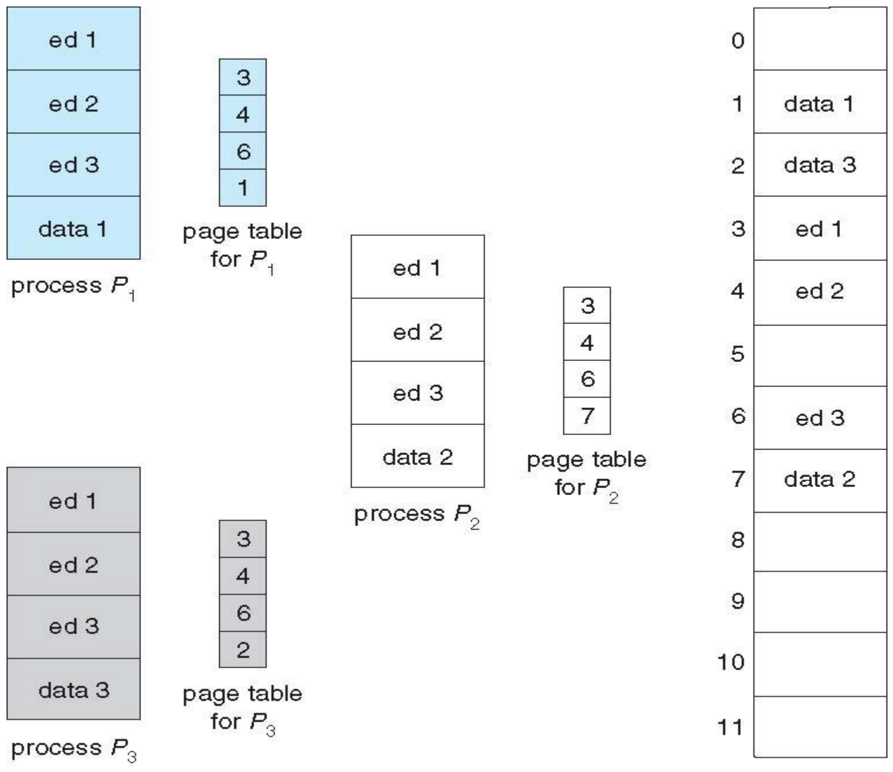
Some systems provide hardware, in the form of a **page-table length register (PTLR)**, to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the operating system.

4. Shared Pages

An advantage of paging is the possibility of ***sharing*** common code. This consideration is particularly important in a time-sharing environment.

Example:

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. If the code is **reentrant code** or **pure code (**Reentrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time. However, it can be shared.



Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different. 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. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB—a significant savings.

Other heavily used programs can also be shared—compilers, window systems, run-time libraries, database systems, and so on.

Segmentation with Paging

Pure segmentation is not very popular and not being used in many of the operating systems. However, Segmentation can be combined with Paging to get the best features out of both the techniques.

In Segmented Paging, the main memory is divided into variable size segments which are further divided into fixed size pages. Pages are smaller than segments. Each Segment has a page table which means every program has multiple page tables.

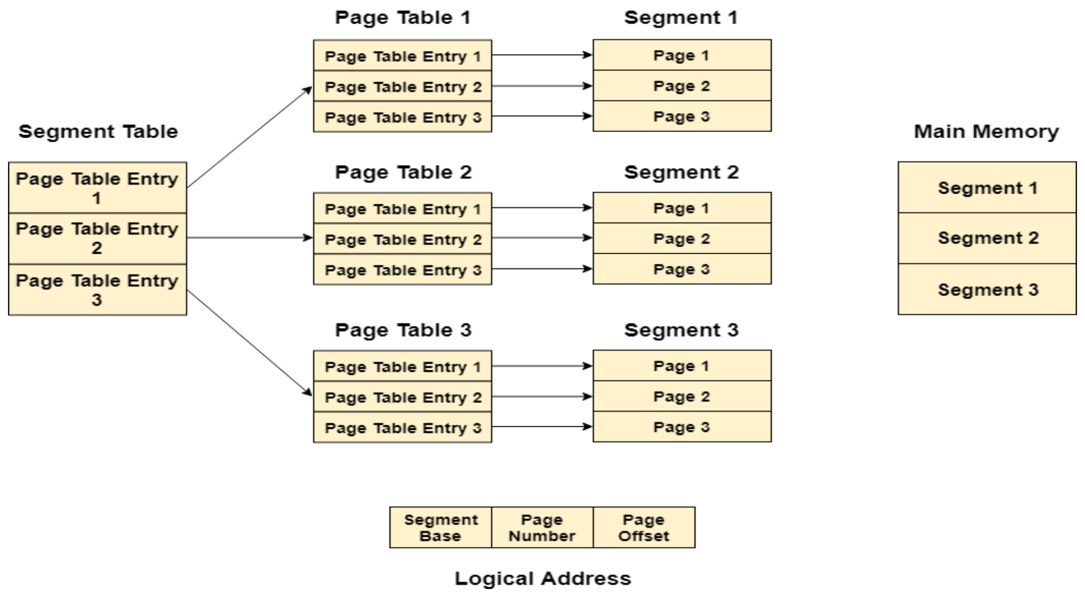
The logical address is represented as Segment Number (base address), Page number and page offset.

Segment Number → It points to the appropriate Segment Number.

Page Number → It Points to the exact page within the segment

Page Offset → Used as an offset within the page frame

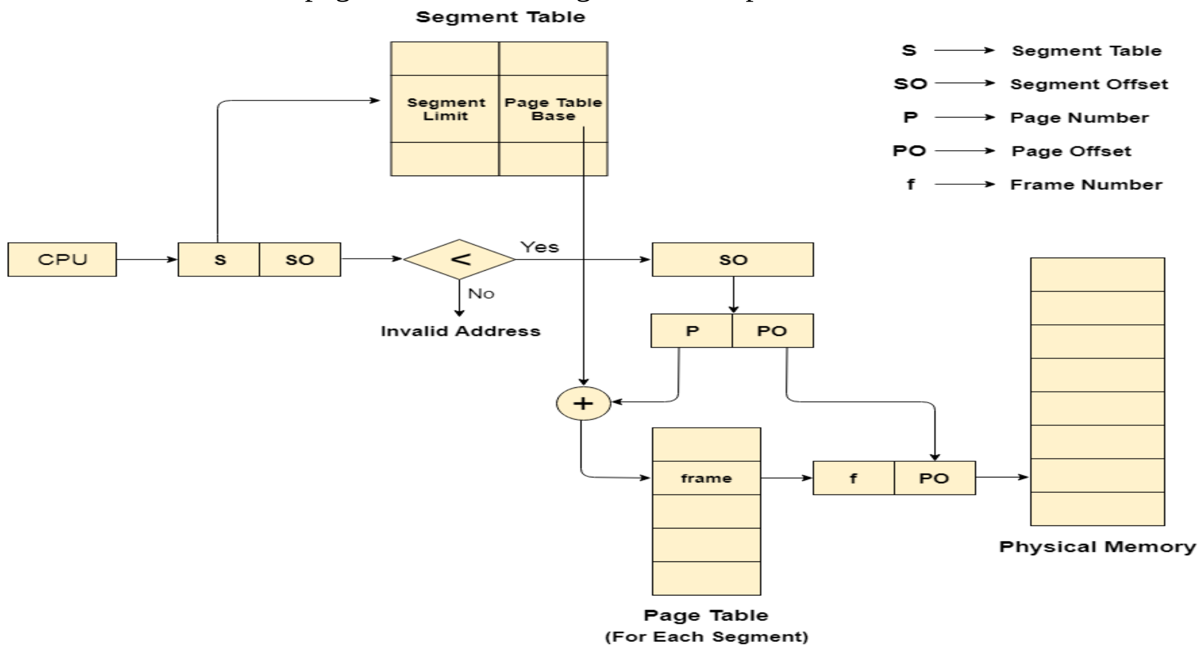
Each Page table contains the various information about every page of the segment. The Segment Table contains the information about every segment. Each segment table entry points to a page table entry and every page table entry is mapped to one of the page within a segment.



Translation of logical address to physical address

The CPU generates a logical address which is divided into two parts: Segment Number and Segment Offset. The Segment Offset must be less than the segment limit. Offset is further divided into Page number and Page Offset. To map the exact page number in the page table, the page number is added into the page table base.

The actual frame number with the page offset is mapped to the main memory to get the desired word in the page of the certain segment of the process.



Advantages of Segmented Paging

1. It reduces memory usage.
2. Page table size is limited by the segment size.

3. Segment table has only one entry corresponding to one actual segment.
4. External Fragmentation is not there.
5. It simplifies memory allocation.

Disadvantages of Segmented Paging

1. Internal Fragmentation will be there.
2. The complexity level will be much higher as compare to paging.
3. Page Tables need to be contiguously stored in the memory.

Demand Paging

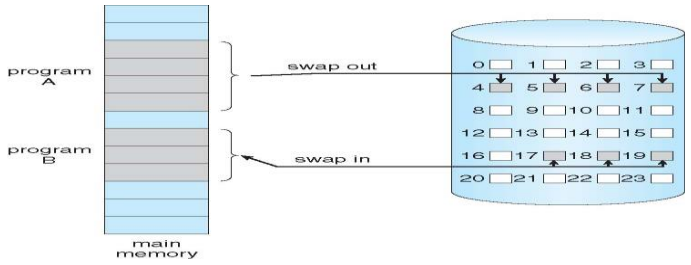
Definition

Loading the entire program into memory results in loading the executable code for ***all*** options, regardless of whether or not an option is ultimately selected by the user. An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems.

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

Lazy Swapper

A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, though, we use a **lazy swapper**. A lazy swapper never swaps a page into memory unless that page will be needed. In the context of a demand-paging system, use of the term “swapper” is technically incorrect. We thus use “pager,” rather than “swapper,” in connection with demand paging.



Transfer of a paged memory to contiguous disk space.

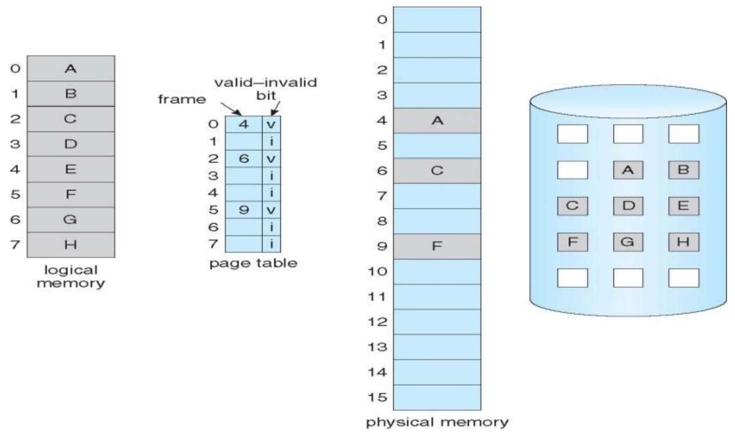
1. **Basic Concepts**

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

Valid–Invalid Bit

- We need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. when this bit is set to “valid,” the associated page is both legal and in memory.
- If the bit is set to “invalid,” the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that

is not currently in memory is either simply marked invalid or contains the address of the page on disk.

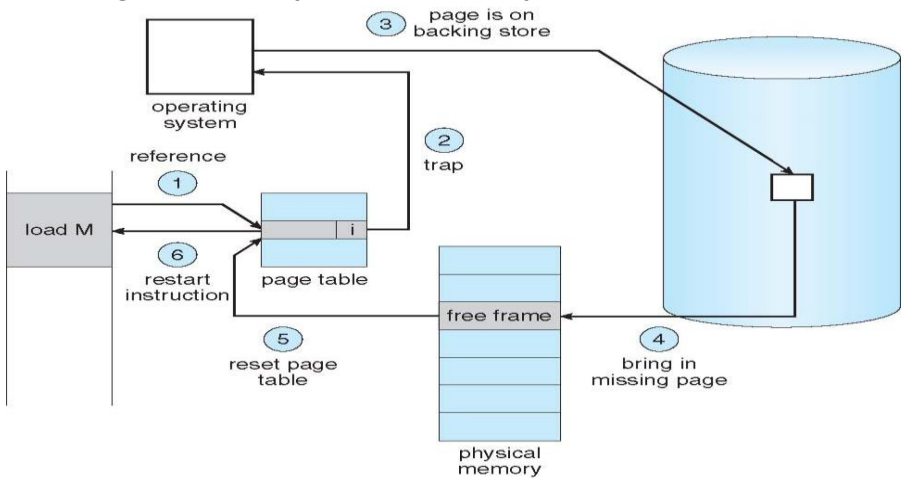


Page Fault

Access to a page marked invalid causes a **page fault**. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory.

The procedure for handling this page fault is straightforward

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



Pure Demand Paging

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 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.

Hardware to Support 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**.

A crucial requirement for demand paging is the ability to restart any instruction after a page fault. Because we save the state (registers, condition code, and instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in ***exactly*** the same place and state, except that the desired page is now in memory and is accessible. In most cases, this requirement is easy to meet.

A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.

2. Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system let's compute the **effective access time** for a demand-paged memory. For most computer systems, the memory-access time, denoted *ma*, ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from disk and then access the desired word.

Let p be the probability of a page fault ($0 \leq p \leq 1$). We would expect p to be close to zero—that is, we would expect to have only a few page faults.

The **effective access time** is then

$$\textbf{Effective Access Time} = (1 - p) \times ma + p \times \textbf{page fault time}$$

To compute the effective access time, we must know how much time is needed to service a page fault.

Example:

With an average page-fault service time of 8 milliseconds and a memory access time of 200 nanoseconds, the effective access time in nanoseconds is

$$\begin{aligned}\text{Effective Access Time} &= (1 - p) \times (200) + p\ (8 \text{ milliseconds}) \\ &= (1 - p) \times 200 + p \times 8{,}000{,}000 \\ &= 200 + 7{,}999{,}800 \times p.\end{aligned}$$

We see, then, that the effective access time is directly proportional to the **page-fault rate**.

An additional aspect of demand paging is the handling and overall use of swap space. Disk I/O to swap space is generally faster than that to the file system. It is a faster file system because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used. However, swap space must still be used for pages not associated with a file (known as **anonymous memory**).

Mobile operating systems typically do not support swapping. Instead, these systems demand-page from the file system and reclaim read-only pages (such as code) from applications if memory becomes constrained. Such data can be demand-paged from the file system if it is later needed. Under iOS, anonymous memory pages are never reclaimed from an application unless the application is terminated or explicitly releases the memory.

Page Replacement

In Demand Paging, pages are only brought into memory only when needed. This has two benefits,

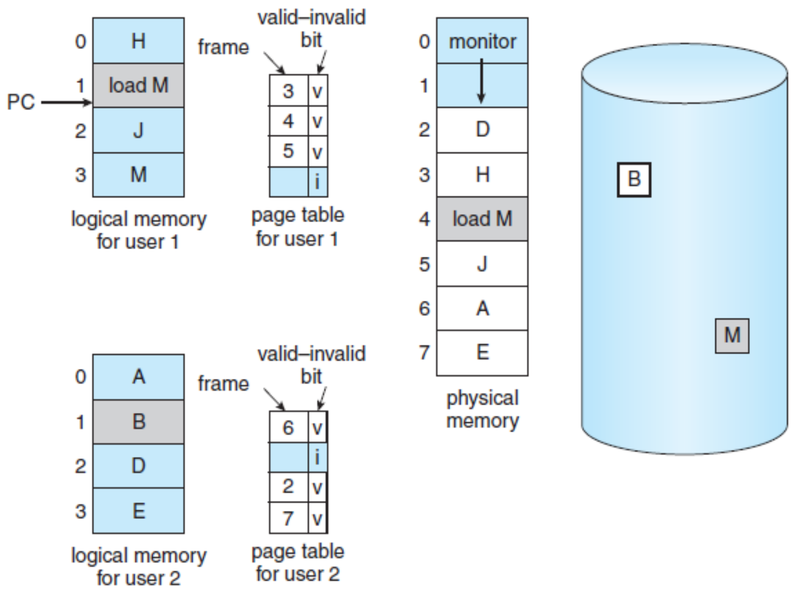
1. Saves I/O necessary to load unused pages.
2. Increases the degree of multiprogramming.

But increasing degree of multi programming may arise new problem called "Over allocating of memory".

Over-Allocating Memory

For example, there are 10 processes and each has 10 pages out of which only 5 may be used. If there are 50 frames then we can allocate only 5 processes if all the 10 pages are loaded. But by using demand paging (we load only used or demanded pages) we can accommodate 10 processes as only 5 pages are in demand. Problem arises when suddenly a process needs all 10 pages but no frames are free.

Over-allocation of memory manifests itself as follows. While a user process is executing, a page fault occurs. The operating system determines where the desired page is residing on the disk but then finds that there are ***no*** free frames on the free-frame list; all memory is in use. The operating system has several options at this point. It could terminate the user process. This option is not the best choice. The operating system could instead swap out a process, freeing all its frames and reducing the level of multiprogramming. This option is a good one but requires page replacement.

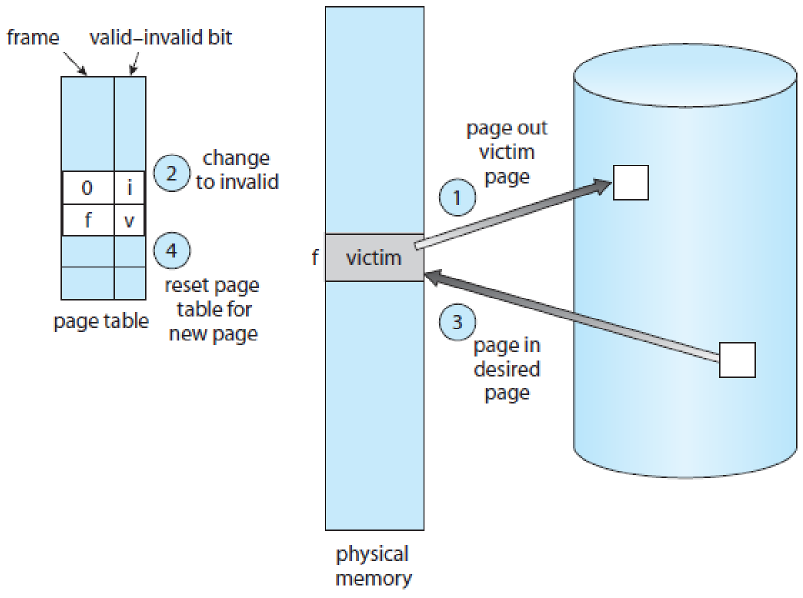


Need for page replacement.

1. 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. Continue the user process from where the page fault occurred.



Page replacement.

Modify Bit (or **Dirty Bit**).

- If no frames are free, ***two*** page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly. We can reduce this overhead by using a **modify bit** (or **dirty bit**).
- When this scheme is used, each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified.
- When we select a page for replacement, we examine it’s modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk. If the modify bit is not set, however, the page has ***not*** been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there.

Major Problems to Implement Demand Paging

We must solve two major problems to implement demand paging: we must develop a **frame-allocation algorithm** and a **page-replacement algorithm**.

That is, if we have multiple processes in memory, we must decide how many frames to allocate to each process; and when page replacement is required, we must select the frames that are to be replaced.

Reference String

There are many different page-replacement algorithms. We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string**.

We can generate reference strings

- Artificially (by using a random-number generator, for example).
- We can trace a given system and record the address of each memory reference. But this produces large amount of data.

To reduce this, we use two facts

a. First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than the entire address.

b. Second, if we have a reference to a page *p*, then any references to page *p* that ***immediately*** follow will never cause a page fault.

Example

If we trace a particular process, we might record the following address sequence:
0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

At 100 bytes per page, this sequence is reduced to the following reference string:

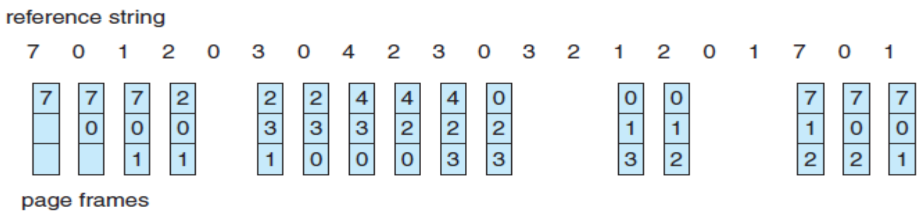
1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Page Replacement Algorithms

FIFO Page Replacement

- The simplest page-replacement algorithm is a 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.
- We can create a FIFO queue to hold all pages in memory. 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.
- The FIFO page-replacement algorithm is easy to understand and program.
- However, its performance is not always good. a bad replacement choice increases the page-fault rate and slows process execution. If we place an active page, some other page should be replaced to bring it back.

Example:



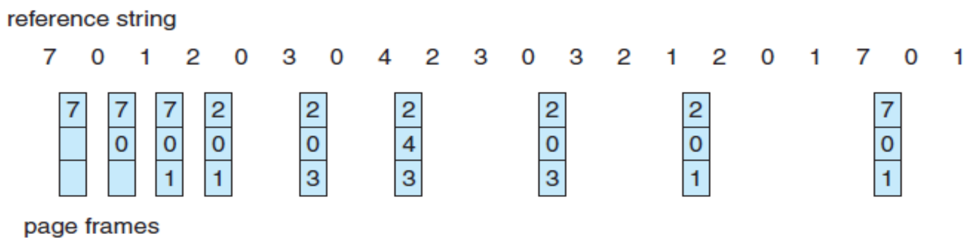
FIFO page-replacement algorithm.

Belady's anomaly: For some page-replacement algorithms, the page-fault rate may ***increase*** as the number of allocated frames increases.
Consider the following reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Number of faults for four frames (ten) is ***greater*** than the number of faults for three frames (nine)

Optimal Page Replacement

- It says that**,** Replace the page that will not be used for the longest period of time.
- It has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.
- As a result, the optimal algorithm is used mainly for comparison studies.

Example:

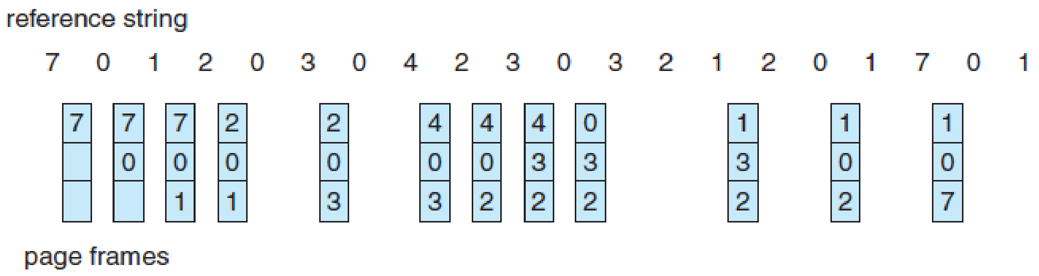


Optimal page-replacement algorithm.

LRU Page Replacement

- LRU replacement associates with each page the time of that page's last use.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.
- We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward.
- Like optimal replacement, LRU replacement does not suffer from Belady's anomaly. Both belong to a class of page-replacement algorithms, called **stack algorithms**.

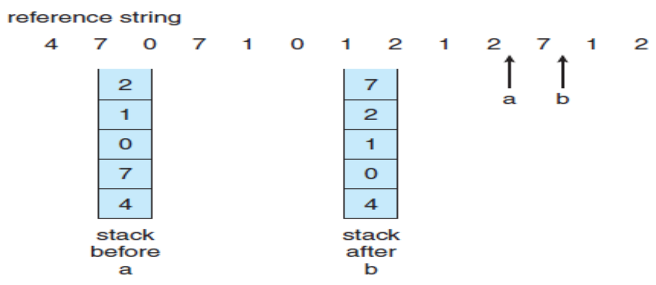
Example:



LRU page-replacement algorithm.

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

- **Counters**. In the simplest case, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page and a write to memory (to the time-of-use field in the page table) for each memory access.
- **Stack**. Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom. Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a tail pointer.



Use of a stack to record the most recent page references

LRU-Approximation Page Replacement

- Few computer systems provide sufficient hardware support for true LRU page replacement. In fact, some systems provide no hardware support, and other page-replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a **reference bit**.
- The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.
- Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the ***order*** of use. This information is the basis for many page-replacement algorithms that approximate LRU replacement.

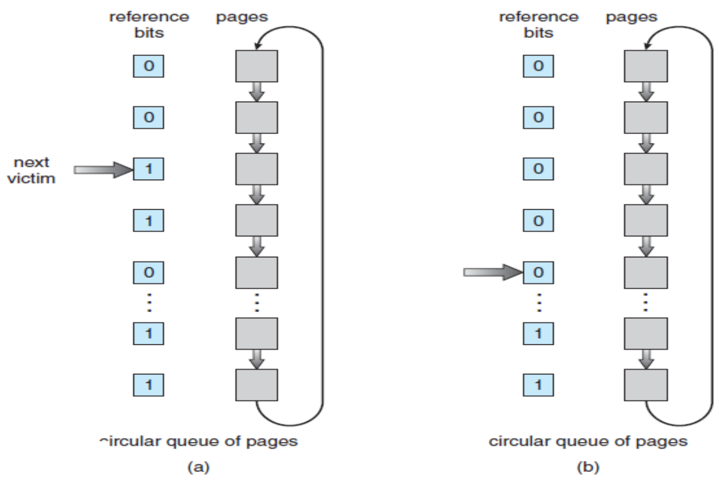
Additional-Reference-Bits Algorithm

- We can gain additional ordering information by recording the reference bits at regular intervals.
- We can keep an 8-bit byte for each page in a table in memory.
- At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system.

- The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit. These 8-bit shift registers contain the history of page use for the last eight time periods.
- If the shift register contains 00000000, for example, then the page has not been used for eight time periods.
- A page that is used at least once in each period has a shift register value of 11111111. A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.
- If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced. Notice that the numbers are not guaranteed to be unique, however. We can either replace (swap out) all pages with the smallest value or use the FIFO method to choose among them.

Second-Chance Algorithm OR clock algorithm

- The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, we inspect its reference bit.
- If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we 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 (or given second chances).
- In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.
- One way to implement the second-chance algorithm is as a circular queue. 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.



Second-chance (clock) page-replacement algorithm.

Enhanced Second-Chance Algorithm

- We can enhance the second-chance algorithm by considering the reference bit and the modify bit as an ordered pair. With these two bits, we have the following four possible classes:

- **(0, 0)** neither recently used nor modified—best page to replace.
- **(0, 1)** not recently used but 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.

Counting-Based Page Replacement

There are many other algorithms that can be used for page replacement. For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes,

Least Frequently Used (LFU)

- The **least frequently used (LFU)** page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.
- A problem arises, however, when a page is used heavily during the 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.
- One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

Most Frequently Used (MFU)

- The **most frequently used (MFU)** page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.