
CODE DIGITAL
HARDWARE DESIGN
INTERNET
SCREEN
DATA
SERVICE
GRAPHIC
INPUT
APPLICATION
OS
APP
WIRELESS
USER
NETWORK
DEVELOPMENT
INFORMATIONCONCEPT
COMPUTING
CONCEPTUAL
MEMORY
BACKGROUND
DESKTOP
PROGRAMMING
OPERATING
SYSTEM
TECHNOLOGY
ALLOCATION
PROGRAM
DEVICE
COMPUTER
SOFTWARE
STORAGE
MOBILE
BUSINESS
WEB
COMMUNICATION
SYMBOL
OUTPUT

Jump2Learn
The Online Learning Place

OPERATING SYSTEM - II

CHAPTER 4 : MEMORY MANAGEMENT

AUTHORS

Mr. Yatin Solanki
M.C.A., Ph.D.(Pursuing)

DOLAT USHA INSTITUTE OF APPLIED SCIENCES, VALSAD

Dr. Jaimin Shukla
M.C.A., M. Phil., Ph.D.(Pursuing)

SUTEX BANK COLLEGE OF COMPUTER APPLICATIONS & SCIENCE, AMROLI

Website : www.jump2learn.com | **Email :** info@jump2learn.com | **Instagram :** www.instagram.com/jump2learn
Facebook : www.facebook.com/Jump2Learn | **Whatsapp :** +91-909-999-0960 | **YouTube :** Jump2Learn

MEMORY MANAGEMENT

In a perfect world ...

Memory is *large, fast, non-volatile*

In real world ...

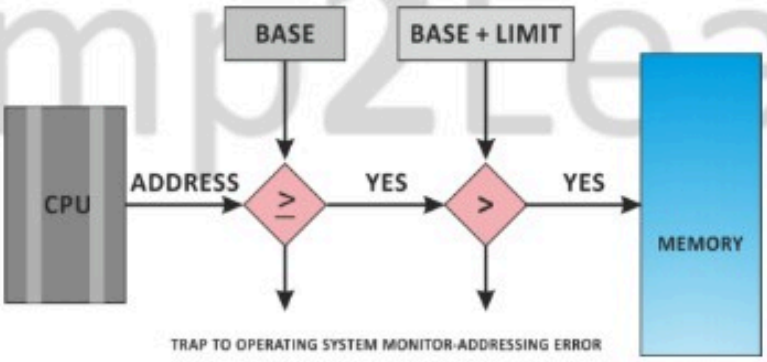
4.1 FUNCTION OF MEMORY MANAGEMENT :

- → Main memory is a **large array of words or bytes**. Each word or byte has its **own address**. For a program to be executed, **it must be mapped to absolute addresses** and loaded into memory. The operating system is responsible for the following activities in connection with memory management.
- → **Keep track of which parts of memory are currently being used by whom.**
- → Decide which **process are to be loaded into memory**, when **memory space becomes available**.
- → **Allocate** and **de – allocate memory** space as needed.

MEMORY MANAGEMENT REQUIREMENTS :

1) **Protection**
2) **Relocation**
3) **Sharing**

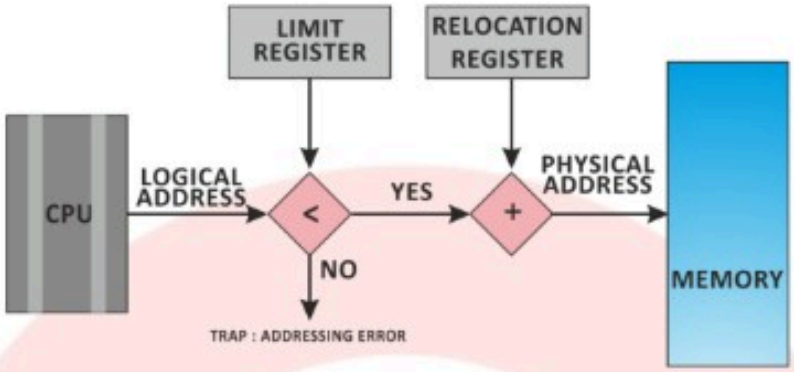
1) **Protection :** Look this figure for the **hardware, mechanism with base and limit register**. If the **logical address is greater than** the **contents of base register** it is the authorized access otherwise it is trap to operating system. The **physical address**

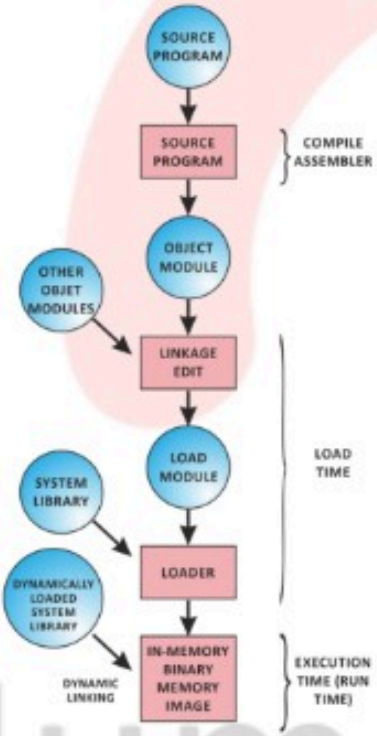


(Logical + Base) is less than the base limit it causes no problems. Otherwise it is trap to the operating system.

2) Relocation : Relocation is a mechanism to convert the logical address in to physical address. **An address is generated by CPU is said to be logical address. An address is generated by memory manager is said to be physical address.**

PHYSICAL ADDRESS = CONTENTS OF RELOCATION REGISTER + LOGICAL ADDRESS.





Relocation is necessary at the time of swap in process from one backing store to main memory.

3) Sharing : Generally protection mechanism are required at the time of access the same potion of main memory by several process. Accessing the same portion of main memory by number of process is said to be "Sharing".

Logical Address Space

Base register holds the smallest legal physical memory address.

Limit register contains the size of the range.

Static: before program start running it is Compile time and Load time.

Dynamic: as program runs it is Execution time.

→ Address Binding

Who assigns memory to segments?

Static-binding: before program start running

Compile time: Compiler and assembler generate an object file for each source file

Load time: Linker **combines all the object files into a single executable** object file.

Loader (part of OS) loads an executable object file into memory at location(s) determined by the OS

DYNAMIC-BINDING: AS PROGRAM RUNS

Execution time:

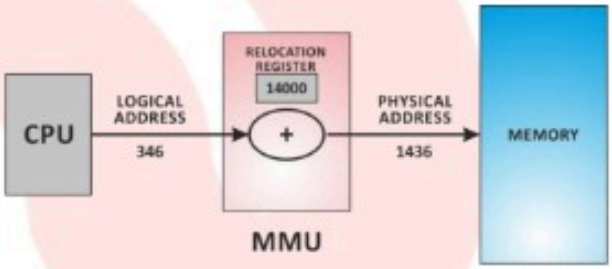
uses ***new*** **and** ***malloc*** **to dynamically** allocate memory. gets **space on stack during** function calls.

Logical vs. Physical Address Space

- Mapping logical address space to physical address space is central to Memory Management.
- **Logical address: generated by the CPU**; also referred to as **virtual address.**
- **Physical address:** address **seen by the memory unit**
- Logical and physical addresses are the **same in compile-time and load-time address-binding** schemes; **logical (virtual) and physical addresses differ in execution-time address-binding** scheme.

MEMORY MANAGEMENT UNIT (MMU)

- Hardware device that **maps virtual to physical address**
- In MMU scheme, the value in the **relocation register is added to every address generated by a user** process at the time it is sent to memory
- The user program deals with ***logical*** addresses; it never sees the **real physical** addresses



DYNAMIC RELOCATION USING A RELOCATION REGISTER :

Physical Address = logical address + content of relocation register.

Static 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.

DYNAMIC LOADING

- Routine is not loaded until it is called.
- Only the main program is loaded into memory and is executed.
- Unused routine is never loaded.
- Better memory-space utilization.
- Useful when large amounts of code are needed to handle infrequently occurring cases.
- No special support from the operating system is required implemented through program design.

ADVANTAGE: BETTER MEMORY UTILIZATION

```
#include<stdio.h>

Main()

{

    Printf("hello");
}
```

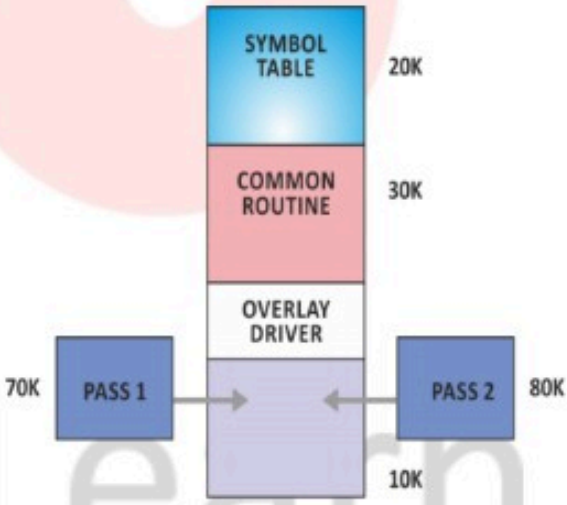
Example : Suppose the program occupies 1KB in secondary storage, But it occupies 100's of KBs in main memory because some routines and header file (stdio.h) should be loaded in main memory to execute that program. Generally this loading is **done at the time of execution this scenario is called as dynamic loading or "with dynamic loading a routine is not loaded until it is called".**

DYNAMIC LINKING

- **Linking postponed until execution time.**
- Small piece of code, ***stub*, used to locate the appropriate memory-resident library routine.**
- **Stub replaces itself with the address of the routine, and executes the routine.**
- Operating system needed to **check if routine is in processes memory address.**
- Dynamic linking is **particularly useful for libraries.**

OVERLAYS

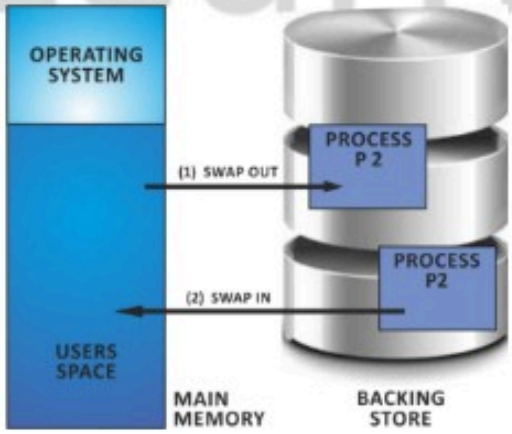
- **Keep in memory only those instructions and data that are needed** at any given time.
- Needed when process is larger than amount of memory allocated to it.
- **Implemented by user, no special support** needed from **operating system, programming design of overlay structure is complex.**



OVERLAYS FOR A TWO-PASS ASSEMBLER →

SWAPPING

- **A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.**
- **Backing store – fast disk large enough to accommodate copies of all memory images for all users**; must **provide direct access** to these memory images.



- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of **swap time is transfer time**; total transfer time is **directly proportional to the amount of memory swapped**.
- **Modified versions of swapping** are found on many systems (i.e., UNIX, Linux, and Windows).

DISPATCHER :

→ The dispatcher is the module that gives control of the CPU to the process selected by the short term scheduler which is used to selects processes from the pool and loads them into memory for execution. This function involves :

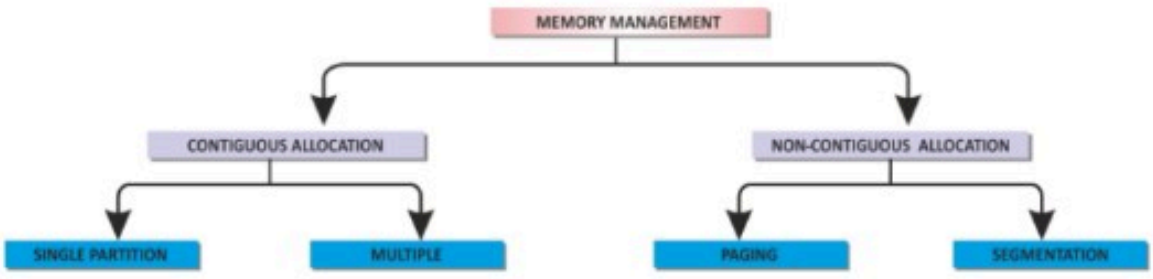
- Switching context.
- Switching to user mode.
- Jumping to the proper location in the user program to restart that program.

→ It should be as fast as possible, given that it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the dispatcher latency.

ALGORITHM TO LOAD A JOB IN A SINGLE USER SYSTEM :

1. Store 1st memory location of program into base register (for memory protection)
2. Set program counter (it keeps track of memory space used by the program) equals to address memory location.
3. Read 1st instruction of Program.
4. Increment program counter by number of bytes in instruction.
5. Has the last instruction been reached?
   If YES, then stop loading program ------→ If NO, then continue with step 6
6. Is program counter greater than memory size?
   If YES, then stop loading program ------→ If NO, then continue with step 7
7. Load instruction in memory.
8. Read next instruction of Program.
9. Go To step 4.
10.

MEMORY MANAGEMENT METHODS OR SCHEMES OR TECHNIQUES



4.2 CONTIGUOUS MEMORY ALLOCATION

- Contiguous memory allocation is a classical memory allocation model that assigns a process consecutive memory blocks (that is, memory blocks having consecutive addresses).
- Contiguous memory allocation is one of the oldest memory allocation schemes.
- When a process needs to execute, memory is requested by the process.
- The size of the process is compared with the amount of contiguous main memory available to execute the process.
- If sufficient contiguous memory is found, the process is allocated memory to start its execution. Otherwise, it is added to a queue of waiting processes until sufficient free contiguous memory is available.
- The contiguous memory allocation scheme can be implemented in operating systems with the help of two registers, known as the base and limit registers.
- When a process is executing in main memory, its base register contains the starting address of the memory location where the process is executing, while the amount of bytes consumed by the process is stored in the limit register.
- A process does not directly refer to the actual address for a corresponding memory location. Instead, it uses a relative address with respect to its base register.
- All addresses referred by a program are considered as virtual addresses.
- The CPU generates the logical or virtual address, which is converted into an actual address with the help of the memory management unit (MMU).
- The base address register is used for address translation by the MMU. Thus, a physical address is calculated as follows:
    - Physical Address = Base register address + Logical address/Virtual address
- The address of any memory location referenced by a process is checked to ensure that it does not refer to an address of a neighbouring process. This processing security is handled by the operating system.
- One disadvantage of contiguous memory allocation is that the degree of multiprogramming is reduced due to processes waiting for free memory.

4.2.1 PARTITIONED MEMORY

SINGLE-PARTITION ALLOCATION

- Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data
- Relocation register contains value of smallest physical address; limit register contains range of logical addresses each logical address must be less than the limit register.

MULTIPLE-PARTITION ALLOCATION

- Hole – block of available memory; holes of various size are scattered throughout memory.
- → When a process arrives it is allocated memory from a process arrives, it is allocated memory from a
- Hole large enough to accommodate it.
- Operating system maintains information about: a) allocated partitions b) free partitions (hole)

4.2.2 STATIC AND DYNAMIC MEMORY ALLOCATION

- Static Allocation (fixed in size)
  - Sometimes we create data structures that are "fixed" and don't need to grow or shrink
  - Done at compile time.
  - Global variables: variables declared "ahead of time" such as fixed arrays.
  - Lifetime: entire runtime of program

ADVANTAGE:

- Efficient execution time.

Disadvantage:

- If we declare more static data space than we need, we waste space.
- If we declare less static space than we need, shortage of space introduce later on.

- Dynamic Allocation (change in size)
  - At other times, we want to increase and decrease the size of our data structures to accommodate changing needs.
  - Often, real world problems mean that we don't know how much space to declare, as the number needed will change over time.
  - Done at run time.
  - Data structures can grow and shrink to fit changing data requirements
  - We can allocate (create) additional storage whenever we need them.
  - We can de-allocate (free/delete) dynamic space whenever we are done with .

➢ Advantage: we can always have exactly the amount of space required - no more, no less.

➢ For example, with references to connect them, we can use dynamic data structures to create a chain of data structures called a linked list.

Advantage: Efficient storage use.

DYNAMIC STORAGE ALLOCATION PROBLEM

→ When a process arrives and needs memory, the system searches this set for a hole that is large enough for this process. If the hole is too large, it is split into two: 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.

→ This procedure is a particular instance of the general **dynamic storage allocation problem**, which is how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem. The set of holes is searched to determine which hole is best to allocate. The first-fit, best-fit, and worst-fit strategies are the most common ones 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 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 kept 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.

→ Simulations have shown that both **first fit and best fit are better than worst fit in terms of decreasing both time and storage utilization.** Neither first fit nor best fit is clearly better in terms of storage utilization, but **first fit is generally faster**.

→ These algorithms, however, 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 enough total memory space** exists to satisfy a request, but **it is not contiguous**; storage is fragmented into a large number of small holes. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all this memory were in one big free block, we might be able to run several more processes.

External Fragmentation: It happens when a dynamic memory allocation algorithm allocates some memory and a small piece is left over that cant a effectively used. If too much external fragmentation occurs, the amount of usable memory is drastically reduced. Total memory space exits to satisfy a request, but it is not contiguous.

Internal Fragmentation: Internal fragmentation is the space wasted inside of allocated memory blocks because of restriction on the allowed sizes of allocated blocks. Allocated

memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

4.3 NON-CONTIGUOUS MEMORY ALLOCATION

Contiguous allocation allocates one single contiguous block of memory to the process whereas, the non-contiguous allocation divides the process into several blocks and place them in the different address space of the memory.

The Non-contiguous memory allocation allows a process to acquire the several memory blocks at the different location in the memory according to its requirement.

The non-contiguous memory allocation also reduces the memory wastage caused due to internal and external fragmentation.

Paging and segmentation are the two ways which allow a process's physical address space to be non-contiguous.

In non-contiguous memory allocation, the process is divided into blocks (pages or segments) which are placed into the different area of memory space according to the availability of the memory.

The non-contiguous memory allocation has an advantage of reducing memory wastage but, it increases the overheads of address translation.

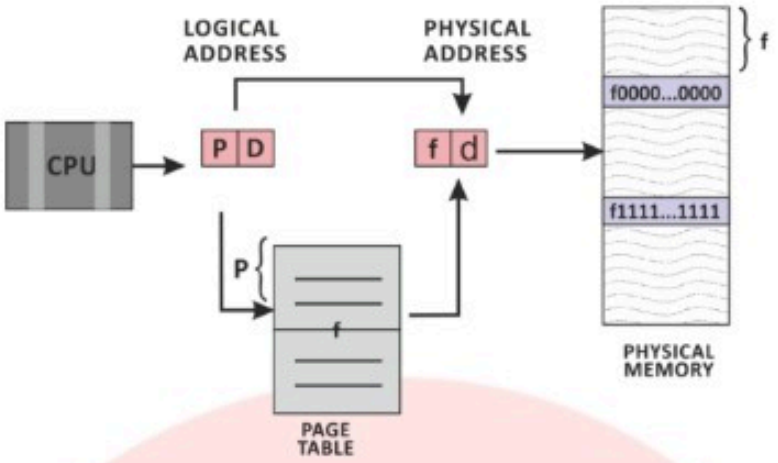
As the parts of the process are placed in a different location in memory, it slows the execution of the memory because time is consumed in address translation.

In contiguous memory allocation, operating system has to maintain a table which indicates which partition is available for the process and which is occupied by the process. In non-contiguous memory allocation, a table is maintained for each process which indicates the base address of each block of the process placed in the memory space.

VIRTUAL MEMORY

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard disk that's set up to emulate the computer's RAM

Virtual memory is a memory management capability of an operating system (OS) that uses hardware and software to allow a computer to compensate for physical memory shortages by temporarily transferring data from random access memory (RAM) to disk storage.



4.3.1 PAGING:

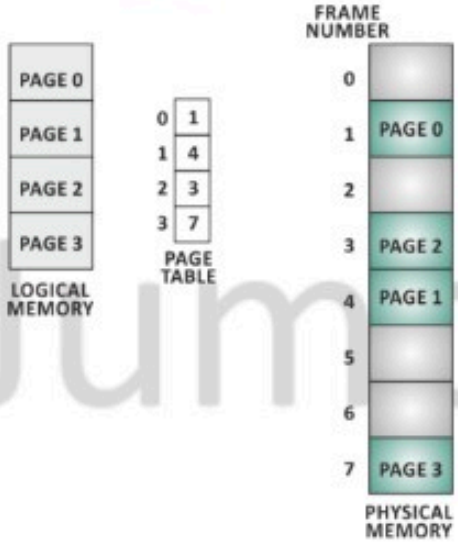
→ Paging is a memory-management scheme that permits the **physical-address space of a process to be noncontiguous**

. → Paging avoids the considerable problem of fitting the varying-sized memory chunks onto the backing store, from which most of the previous memory-management schemes suffered.

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

→ **Basic Method:**

→ Physical memory is broken into fixed-sized blocks called **frames**. Logical memory is also broken 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 the backing store. 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 9.6. Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d).**

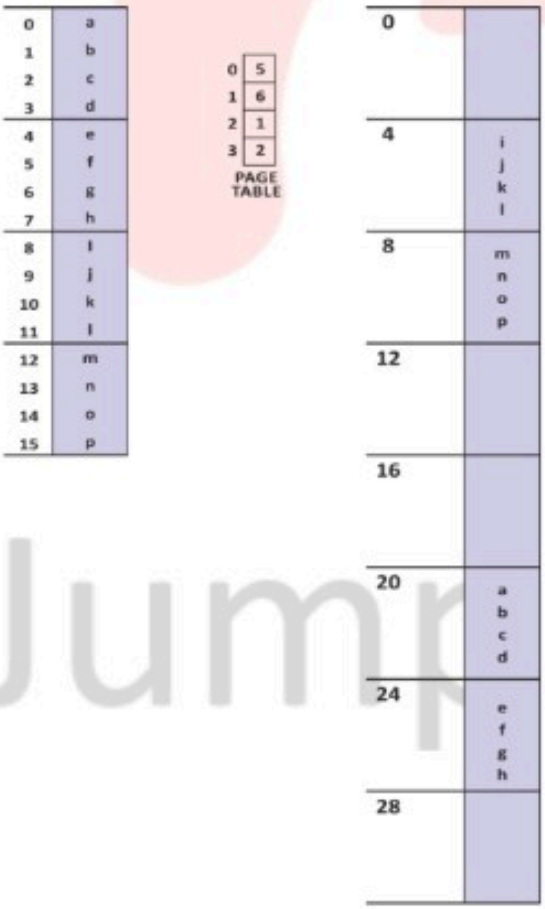


→ The page number is used as an index into a **page table**. The page table contains the **base address** of each page in physical memory. This base address is combined with the **page offset to define the physical memory address** that is sent to the memory unit. The paging model of memory is shown in Figure 8.8.

→ 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 logical-address space is 2m, and a page size is *2n* addressing 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.

example, consider the memory in Figure 8.9 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 (= (5 x 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 x 4) + 0). Logical address 13 maps to physical address 9.



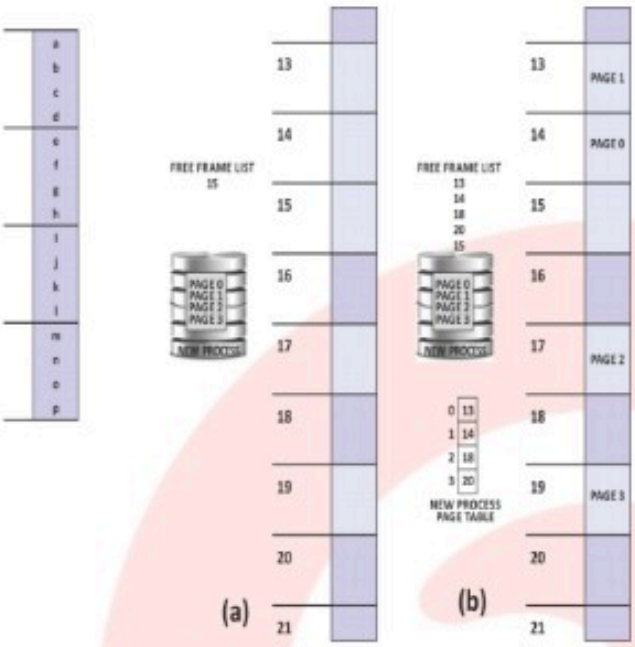
→ You may have noticed that paging itself is a form of dynamic relocation.

→ Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

→ When we use a paging scheme, we have no external fragmentation: Any free frame can be allocated to a process that needs it.

→ However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to fall on page boundaries, the last frame allocated may not be completely full. For example, if pages are 2,048 bytes, a process of 72,766 bytes would need 35 pages plus 1,086 bytes. It would be allocated 36 frames, resulting in an internal fragmentation of 2048 - 1086 = 962 bytes. In the worst case, a process would need n pages plus one byte. It would be allocated n + 1 frames, resulting in an internal fragmentation of almost an entire frame.

→ Today pages typically are between 4 KB and 8 KB, and some systems support even larger page sizes. Some CPUs and kernels even support multiple page sizes. For instance, Solaris uses 8 KB and 4 MB page sizes, depending on the data stored by the pages.



→ When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires *n* pages, at least *n* frames must be available in memory. If *n* frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, and its frame number is put into the page table, and so on (Figure 8.10).

→ An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. The user program views that memory as one single contiguous space, containing only this one program.

Advantages and Disadvantages of Paging:

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

PAGING OF HARDWARE WITH PTB

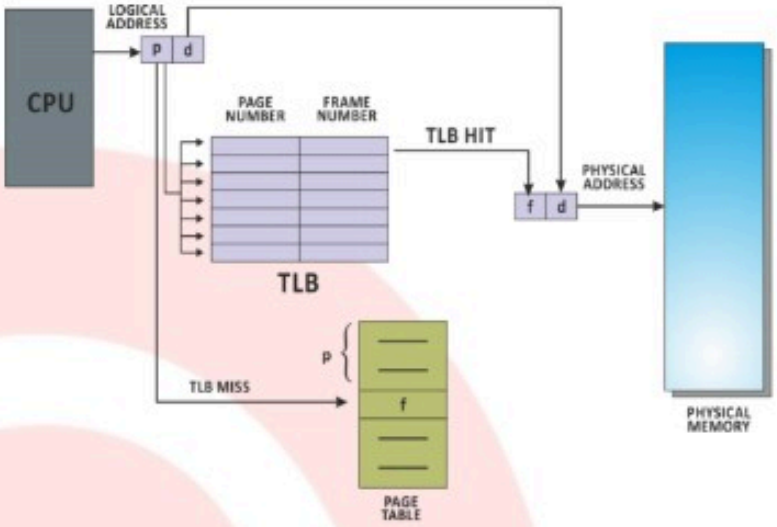
→ The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated registers. These registers should be built with very high-speed logic to make the paging-address translation efficient.

→ 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 DEC PDP-11 is an example of such an architecture. The address consists of 16 bits, and the page size is 8 KB.

→ 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. 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).

→ The standard solution to this problem is to use a special, small, fast lookup hardware cache, called 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. When the associative memory is presented with an item, it is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; the hardware, however, is expensive. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.



→ 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. When the frame number is obtained, we can use it to access memory (Figure 8.11).

In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from **least recently used (LRU)** to random. Furthermore, some TLBs allow entries to be wired down, meaning that they cannot be removed from the 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 must weigh 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

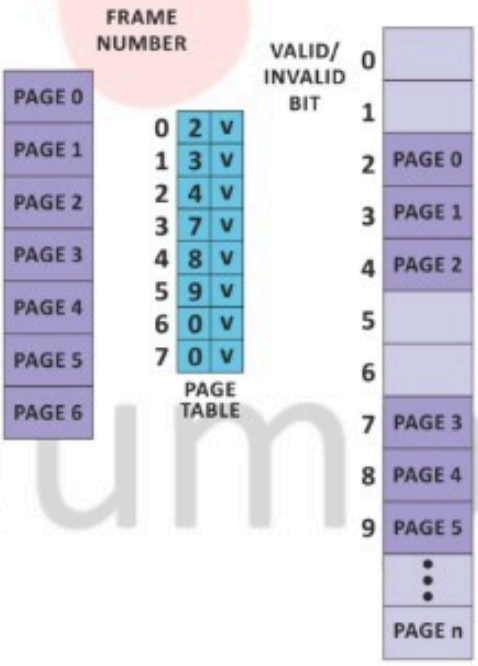
access time = 0.98 x 120 + 0.02 x 220 = 122 nanoseconds.

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

PROTECTION ON PAGE TABLE

→ One more bit is generally attached to each entry in the page table: a **valid-invalid** bit. When this bit is set to "valid," this value indicates that the associated page is in the process' logical-address space, and is thus a legal (or valid) page.

→ If the bit is set to "invalid," this value indicates that the page is not in the process' logical-address space. Illegal addresses are trapped by using the valid-invalid bit. The operating system sets this bit for each page to allow or disallow accesses to that page.



→ For example, in a system with a 14-bit address space (0 to 16383), we may have a program that should use only addresses 0 to 10468.

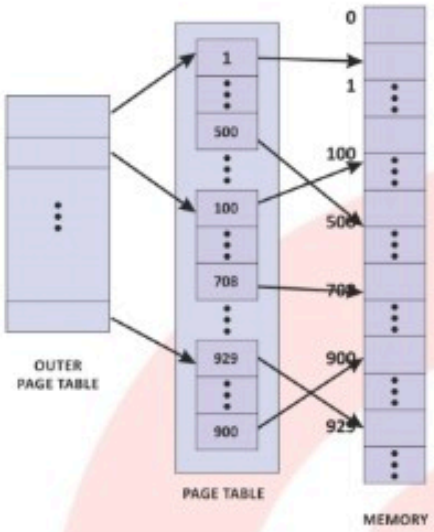
Given a page size of 2 KB, we get the situation shown in Figure 9.11. Addresses in pages 0, 1,2,3,4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, finds that the valid-invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

→ Because the program extends to only address 10468, any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid. This problem is a result of the 2 KB page size and reflects the internal fragmentation of paging.

→ 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.

STRUCTURE OF THE PAGE TABLE



1) Hierarchical Paging :

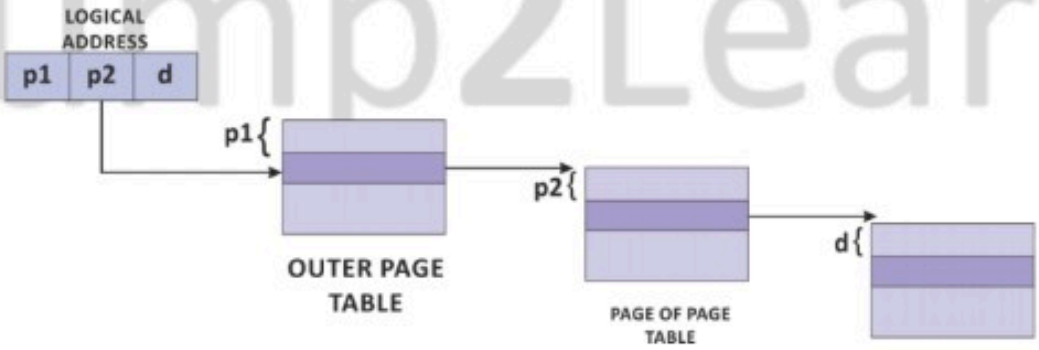
→ page table may consist of up to 1 million entries ($2^{32}/2^{12}$). Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical-address space for the page table alone.

→ If we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces. There are several ways to accomplish this division.

→ One way is to use a two-level paging algorithm, in which the page table itself is also paged (Figure 9.12). Remember our example to our 32-bit machine with a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number

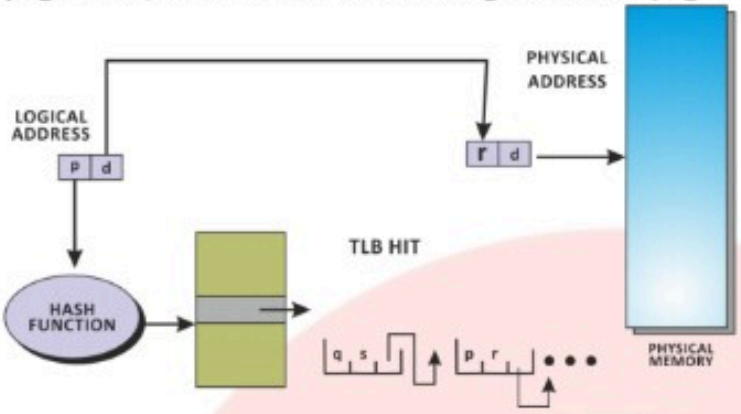
and a 10-bit page offset. Thus, logical address is as follows: where pl is an index into the outer page table and p2 is the displacement within the page of the outer page table. The

address-translation method for this architecture is shown in Figure 9.13. Because address translation works from the outer page table inwards, this scheme is also known as a **forward-mapped page table.** The Pentium-I1 uses this architecture.



2) HASHED PAGE TABLES :

→ A common approach for handling address spaces larger than 32 bits is to use a **hashed page table,** with the hash value being the virtual-page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (a) **the virtual page number, (b) the value of the mapped page frame, and (c) a pointer to the next element in the linked list.**

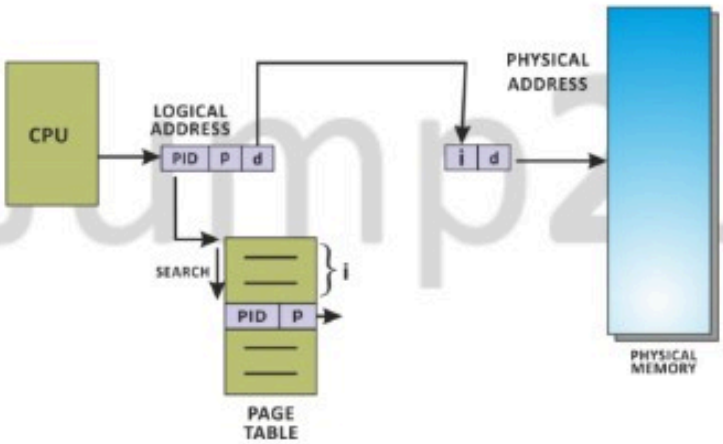


→ The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared to field (a) in the first element in the linked list. If there is a match, the corresponding page frame (field (b)) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure 9.14.

→ **A** variation to this scheme that is favorable for 64-bit address spaces has been proposed. **Clustered page tables** are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables are particularly useful for **sparse** address spaces where memory references are noncontiguous and scattered throughout the address space.

3) INVERTED PAGE TABLE :

→ each page table may consist of millions of entries. These tables may consume large amounts of physical memory, which is required just to keep track of how the other physical memory is being used.



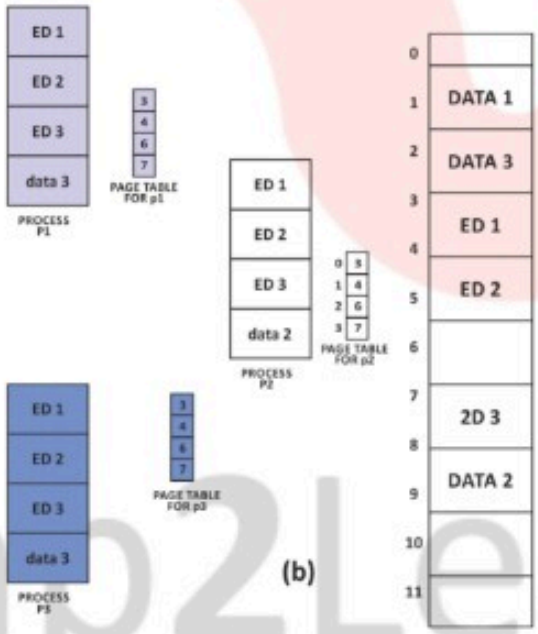
→ To solve this problem, we can use an **inverted page table.** An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.

→ Only one page table is in the system, and it has only one entry for each page of physical memory.

Figure 9.15 shows the operation of an inverted page table. Compare it to Figure 9.6, which depicts a standard page table in operation. Because only one page table is in the system yet there are usually several different address spaces mapping physical memory, inverted page tables often require an address-space identifier stored in each entry of the page table.

SHARED PAGES

- An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment.
- 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 pure code, however, it can be shared, as shown in the Figure on next slide.
- Here, we see three processes sharing a three-page editor—each page 50 KB in size.
- Each process has its own data page. Rentrant 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.



- 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.
- To be sharable, the code must be reentrant (pure code). The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property.
- The sharing of memory among processes on a system is similar to the sharing of the address space of a task by threads.
- Some operating systems implement shared memory using shared pages.
- Organizing memory according to pages provides numerous benefits in addition to allowing several processes to share the same physical pages.

4.3.2 SEGMENTATION

- Segmentation is one of the most common ways to achieve memory protection.
- Memory segmentation is the division of a computer's primary memory into segments or sections.
- In a computer system using segmentation, an instruction operand that refers to a memory location includes a value that identifies a segment and an offset within that segment.
- Segmentation is a memory management technique in which we divide the process into smaller segments.
- The process is segmented module by module.
- In main memory, we only store the segments of the process.
- Process segment table is used to keep the record of segments, its size and its memory address.
- A process is divided into Segments.
- The portions that a program is divided into which are not necessarily all of the same length is called segments.
- There are types of segmentation:

VIRTUAL MEMORY SEGMENTATION: –

Each process is divided into a number of segments, not all of which are resident at any one point in time.

SIMPLE SEGMENTATION: –

Each process is divided into a number of segments, all of which are loaded into memory at run time, though not necessarily contiguously.

- There is no simple relationship between logical addresses and physical addresses in segmentation.
- A table stores the information about all such segments and is called Segment Table.

Segment Table – It maps two dimensional Logical address into one dimensional Physical address.

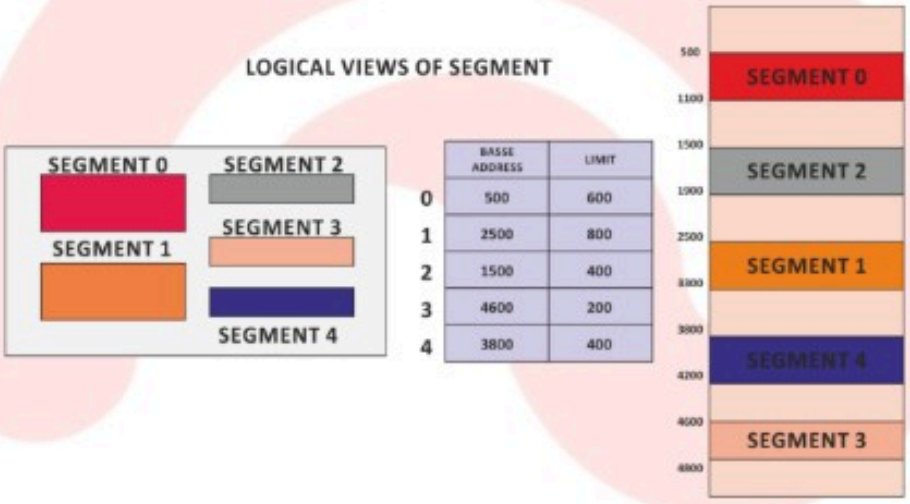
- Each table entry has:

Base Address:

It contains the starting physical address where the segments reside in memory.

Limit:

It specifies the length of the segment.



- Address generated by the CPU is divided into:
  Segment number (s):

Number of bits required to represent the segment.

Segment offset (d):

Number of bits required to represent the size of the segment.

ADVANTAGES OF SEGMENTATION:

- The segment table is used to keep the record of segments and segment table occupies less space as compared to the paging table.
- No internal fragmentation.

DISADVANTAGES OF SEGMENTATION :

Due to segments external fragmentation occurs and external fragmentation results in a lot of memory waste.