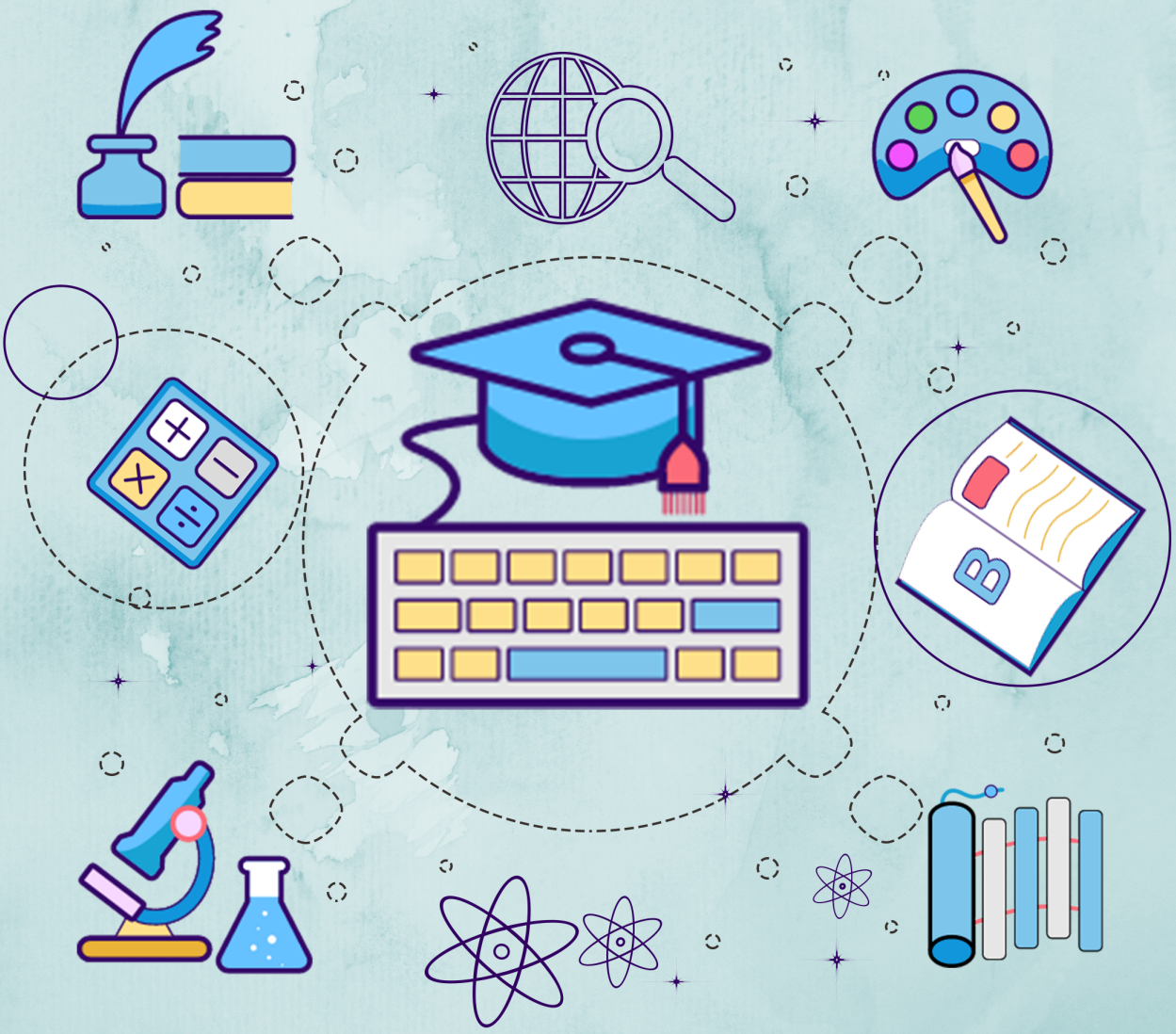
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KeralaNotes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

KTU S4 CSE NOTES

OPERATING SYSTEM (CST200)

Module 4

Related Link :

- KTU S4 CSE NOTES 2019 SCHEME
- KTU S4 SYLLABUS CSE COMPUTER SCIENCE
- KTU PREVIOUS QUESTION BANK S4 CSE SOLVED
- KTU CSE TEXTBOOKS S4 B.TECH PDF DOWNLOAD
- KTU S4 CSE NOTES | SYLLABUS | QBANK | TEXTBOOKS DOWNLOAD

MODULE 4
MEMORY MANAGEMENT

Memory Management: *Main Memory – Swapping- Contiguous Memory Allocation- Segmentation- Paging- Demand Paging*

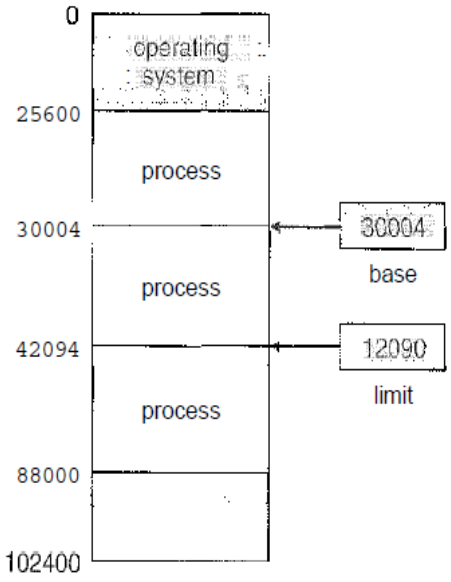
1. MAIN MEMORY

Memory is central to the operation of a modern computer system,. Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses. 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. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore *how* a program generates a memory address.

- **Basic Hardware**

Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. 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.

Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Memory access may take many cycles of the CPU clock to complete, in which case the processor normally needs to **stall,** since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast



A base and a limit register define a logical address space

For More Study Materials : https://www.keralanotes.com/

memory between the CPU and main memory. A memory buffer used to accommodate a speed differential, called a **cache.**

Not only are we concerned with the relative speed of accessing physical memory, but we also must ensure correct operation, has to protect the operating system from access by user processes and, in addition, to protect user processes from one another. This protection must be provided by the hardware.

We first need to make sure that each process has a separate memory space. To do this, 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, as illustrated in Figure 8.1. 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 limit register is 120900, then the program can legally access all addresses from 300040 through 420940 (inclusive).

Protection of memory space is accomplished by having the CPU hardware compare *even/* address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error (Figure 8.2). This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users. The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents. The operating system, executing in kernel mode, is given

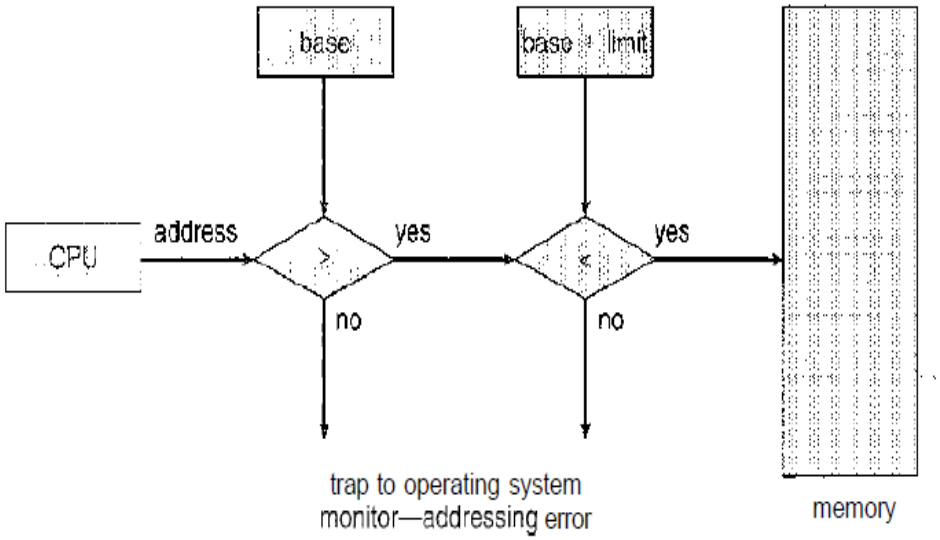


Figure 8.2 Hardware address protection with base and limit registers.

For More Study Materials : https://www.keralanotes.com/

unrestricted access to both operating system and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, and so on.

- **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. The normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available. Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer starts at 00000, the first address of the user process need not be 00000. This approach affects the addresses that the user program can use. In most cases, a user program will go through several steps—some of which maybe optional-—before being executed (Figure 8.3). Addresses may be represented in different ways during these steps.

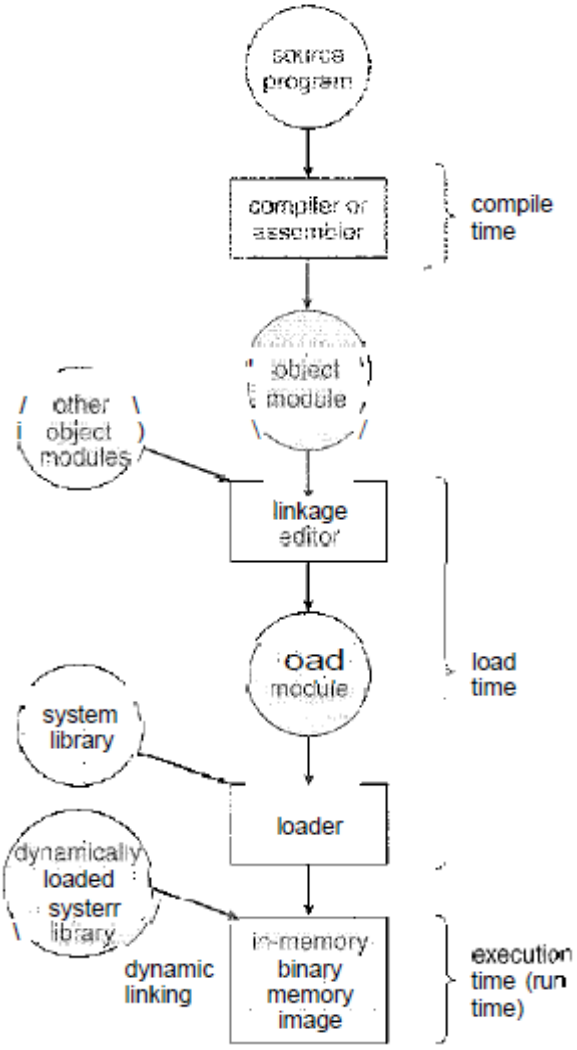


Figure 8.3 Multistep processing of a user program.

Addresses in the source program are generally symbolic (such as *count). A* compiler will typically **bind** these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader will in turn

For More Study Materials : https://www.keralanotes.com/

bind the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another. 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.

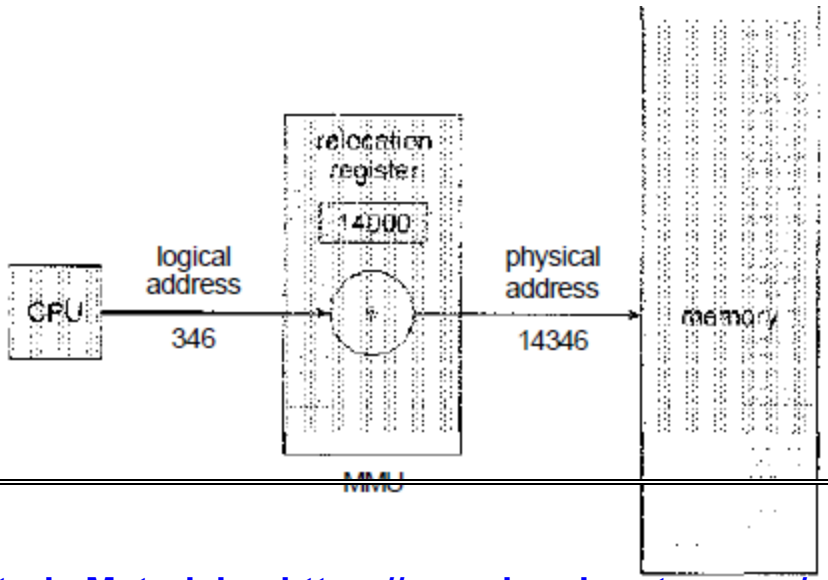
• **Load time.** If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code.** In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.

• **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. Special hardware must be available for this scheme to work. Most general-purpose operating systems use this method.

- **Logical Versus Physical Address Space**

An address generated by the CPU is commonly referred to as a **logical address,** whereas 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 compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address.**

We use *logical address* and *virtual address* interchangeably in this text. 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.** Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ. The run-time mapping from virtual to physical addresses is done by a hardware



For More Study Materials : https://www.keralanotes.com/

Figure 8.4 Dynamic relocation using a relocation register.

device called the **memory-management unit (MMU).** We can choose from many different methods to accomplish such mapping.

The base register is now called a **relocation register.** The value in the relocation register is *added* to every address generated by a user process at the time it is sent to memory (see Figure 8.4). For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346. The MS-DOS operating system running on the Intel 80x86 family of processors uses four relocation registers when loading and running processes. The user program never sees the *real* physical addresses.

The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses—all as the number 346. Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register. The user program deals with *logical* addresses. The memory-mapping hardware converts logical addresses into physical addresses.

The final location of a referenced memory address is not determined until the reference is made. We now have two different types of addresses: logical addresses (in the range 0 to *max)* and physical addresses (in the range *R* + 0 to *R* + *max* for a base value *R).* The user generates only logical addresses and thinks that the process runs in locations 0 to *max.* The user program supplies logical addresses; these logical addresses must be mapped to physical addresses before they are used. The concept of a *logical address space* that is bound to a separate *physical address space* is central to proper memory management.

- **Dynamic Loading**

In our discussion so far, the entire program and all data of a process must be in physical memory for the process to execute. The size of a process is thus limited to the size of physical memory. To obtain better memory-space utilization, we can use **dynamic loading.** With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed.

When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

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

For More Study Materials : https://www.keralanotes.com/

Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

- **Dynamic Linking and Shared Libraries**

Figure 8.3 also shows **dynamically linked libraries.** Some operating systems support only **static linking,** in which system language libraries are treated like any other object module and are combined by the loader into the binary program image. The concept of dynamic linking is similar to that of dynamic loading. Here, though, linking, rather than loading, is postponed until execution time. This feature is usually used with system libraries, such as language subroutine libraries. Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image. This requirement wastes both disk space and main memory.

With dynamic linking, a *stub* is included in the image for each library routine reference. The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present. When the stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory.

Either way, the stub replaces itself with the address of the routine and executes the routine. Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking. Under this scheme, all processes that use a language library execute only one copy of the library code. This feature can be extended to library updates (such as bug fixes).

A library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be relinked to gain access to the new library. So that programs will not accidentally execute new, incompatible versions of libraries, version information is included in both the program and the library.

More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. Minor changes retain the same version number, whereas major changes increment the version number. Thus, only programs that are compiled with the new library version are affected by the incompatible changes incorporated in it.

Other programs linked before the new library was installed will continue using the older library. This system is also known as **shared libraries.** Unlike dynamic loading, dynamic linking generally requires help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity

For More Study Materials : https://www.keralanotes.com/

that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses.

2. 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 into memory for continued execution. For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed (Figure 4.2). In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. In addition, the quantum must be large enough to allow reasonable amounts of computing to be done between swaps.

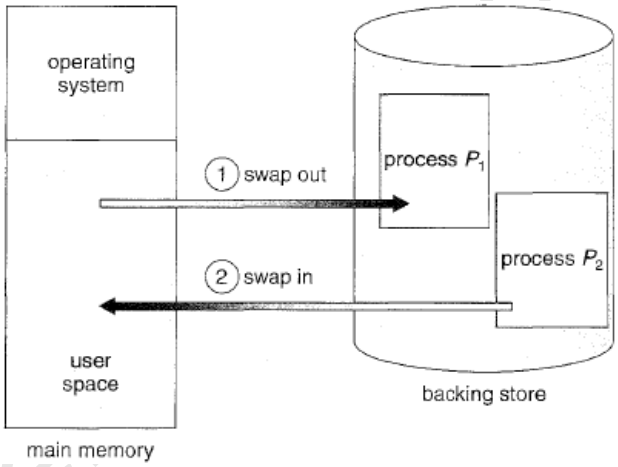


Fig:4.2 Swapping of two processes using a disk as a backing store.

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

Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously. This restriction is dictated by the method of address binding. **If binding is done at assembly or load time, then the process cannot be easily moved to a different location. If execution-time binding is being**

For More Study Materials : https://www.keralanotes.com/

used, however, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

Swapping requires a backing store. The backing store is commonly a fastdisk. It must be large enough to accommodate copies of all memory imagesfor all users, and it must provide direct access to these memory images. Thesystem maintains a **ready queue**consisting of all processes whose memoryimages are on the backing store or in memory and are ready to run. Wheneverthe CPU scheduler decides to execute a process, it calls the dispatcher. Thedispatcher 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 aprocess currently in memory and swaps in the desired process. It then reloadsregisters and transfers control to the selected process.

The context-switch time in such a swapping system is fairly high.

Swapping is constrained by other factors as well. **If we want to swap a process, we must be sure that it is completely idle.** Of particular concern is any pending I/0. A process may be waiting for an I/0 operation when
we want to swap that process to free up memory. However, if the I/0 is asynchronously accessing the user memory for I/0 buffers, then the process cannot be swapped.**Never swap a process with pending I/0.**

3. CONTIGUOUS MEMORY ALLOCATION

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

The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.

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 contiguous section of memory.**

- **Memory Mapping and Protection**

Memory mapping and protection is done with the help of **relocation register, together with a limit register**. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses.

For More Study Materials : https://www.keralanotes.com/

With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address *dynamically* by adding the value in the relocation register. This mapped address is sent *to* memory (Figure 4.3).

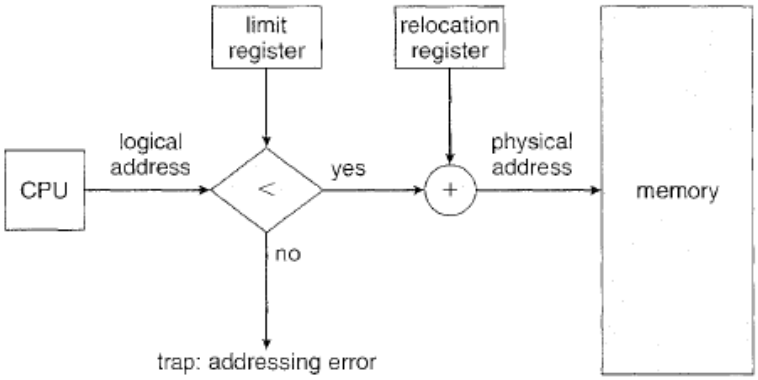


Fig:4.3Hardware supportfor relocation and limit registers.

- **Memory Allocation**

One of the simplest methods for allocating memory is to divide memory into several **fixed-sizedpartitions.** Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. **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.

In the **variable partition scheme**, the operating system keeps a tableindicating which parts of memory are available and which are occupied.Initially, all memory is available for user processes and is considered onelarge block of available memory a **hole**. Eventually as you will see, memorycontains a set of holes of various sizes.

As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory which the operating system may then fill with another process from the input queue.

At any given time, we have a list of available block sizes and the input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied—that is, no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough blocks is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

For More Study Materials : https://www.keralanotes.com/

In general, at any given time we have a *set* of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

This procedure is a particular instance of the general **dynamic storage allocation problem,** which 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.

- **Memory allocation strategies:**
- **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.
- **Next Fit:**It works same as first fit, but after allotting a memory to a process, start your search for the next allocation from the last block allotted rather than from the starting block. This saves from looking at blocks you already allotted at the beginning and thus you are able to do a faster job at allotting blocks to various processes.
- **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.

Simulations have shown that next fit, first fit and best fit are better than wors tfit in terms of decreasing time and storage utilization. But worst fit is better than the others in terms of storage utilization,.

- **Fragmentation**

Memory fragmentation can be internal as well as external.

Internal Fragmentation:The general approach 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

For More Study Materials : https://www.keralanotes.com/

memory. The difference between these two numbers is **internal fragmentation-** unused memory that is internal to a partition.

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. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Statistical analysis of first fit, for instance, reveals that, even with some optimization, given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the **50-percent rule.**

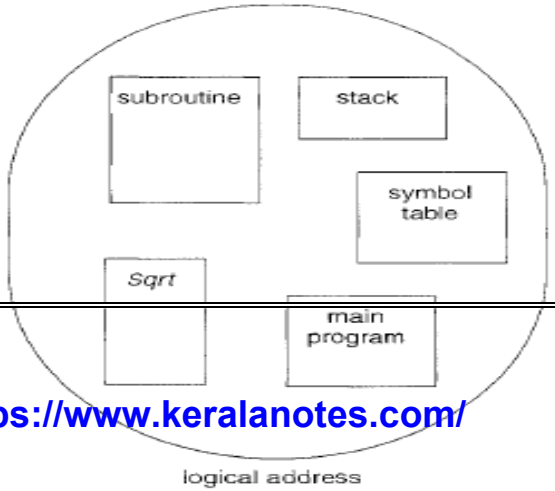
One solution to the problem of external fragmentation is **Compaction**.Thegoal is to shuffle the memory contents so as to place all free memory together in one large block. **Compaction is not always possible, If relocation is static and is done at assembly or load time, compaction cannot be done; compaction is possible *only* if relocation is dynamic and is done at execution time**. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost. **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.

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

4. SEGMENTATION

- **Basic Method**

Segmentation is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify



For More Study Materials : https://www.keralanotes.com/

both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: **a segment name and an offset.** (Contrast this scheme with the paging scheme, in which the user specifies only a single address, which is partitioned by the hardware into a page number and an offset, all invisible to the programmer.) For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two tuple:*

<segment-number, offset>.

Normally, the user program is compiled, and the compiler automatically constructs segments reflecting the input program.

A C compiler might create separate segments for the following:

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

Fig:4.12 User's view of a program.

- **Hardware**

Thus, we must define an implementation to map two dimensional user-defined addresses into one-dimensional physical addresses. This mapping is effected by a Each entry in the segment table has a ***segment base* and a *segment limit*.** The segment base contains the **starting physical address** where the segment resides in memory, and the segment limit specifies the **length of the segment**. The use of a segment table is illustrated in Figure 4.13.

A logical address consists of two parts: **a segment number, s, and an offset into that segment, *d*.** The segment number is used as an index to the segment table. The offset *d* of the logical address must be between 0 and the segment limit. If it is not, 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.

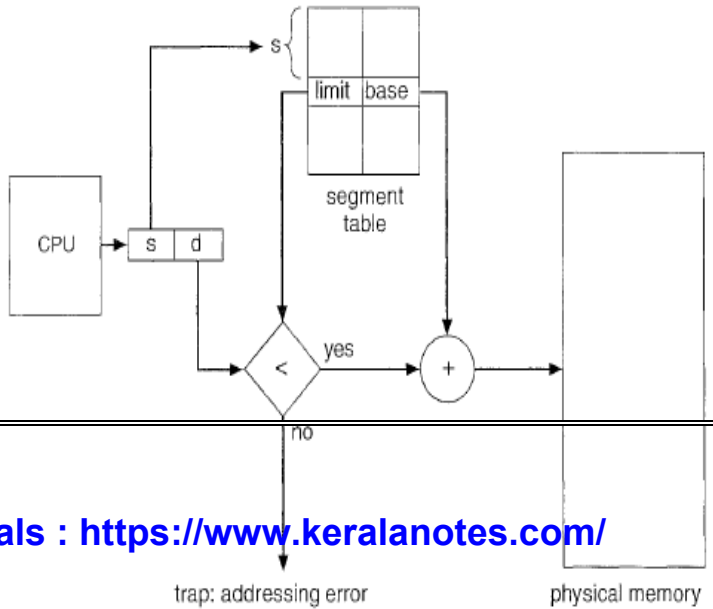


Fig:4.13 Segmentation hardware.

keralanotes.com

As an example, consider the situation shown in Figure 8.20. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 +53= 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1000 bytes long.

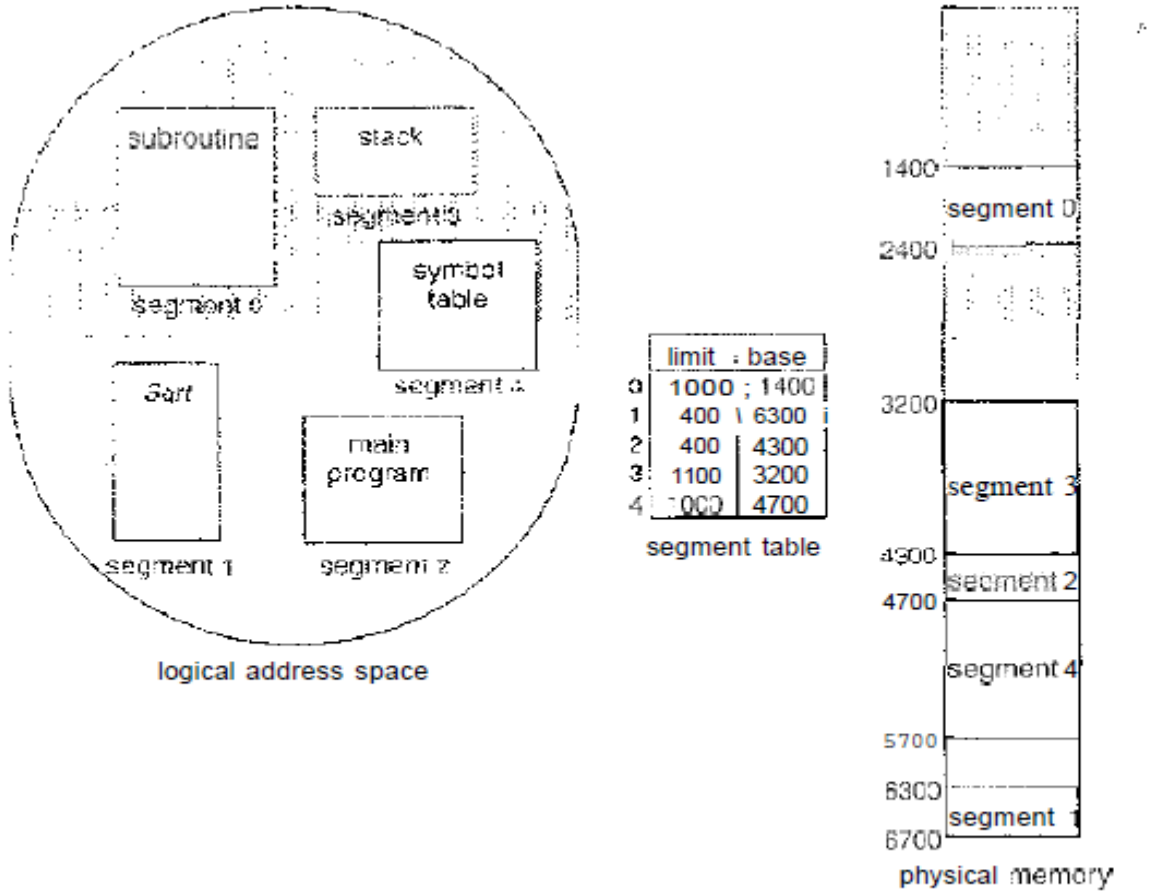


Figure 8.20 Example of segmentation.

5. PAGING

Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous. Paging avoids external fragmentation and the need for compaction. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store. Most memory-management schemes used before the introduction of paging suffered from this problem. The problem arises because, when

For More Study Materials : https://www.keralanotes.com/

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

- **Basic Method**

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and 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). The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

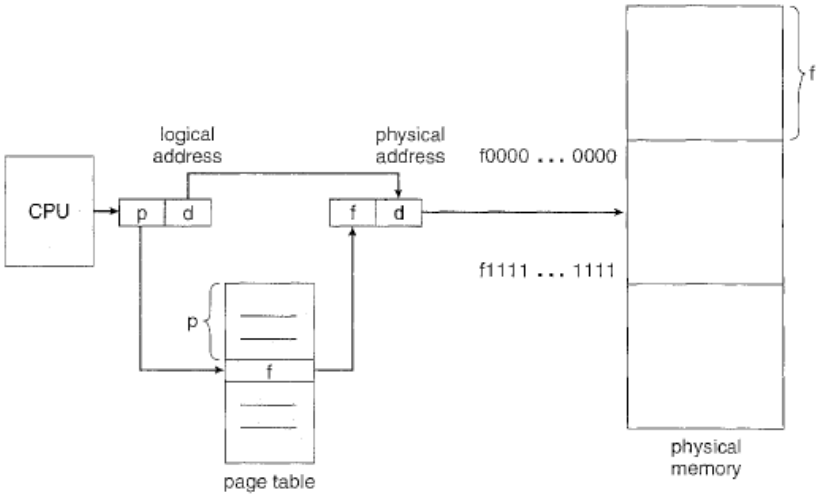


Fig:4.4 Paging Hardware

The hardware support for paging is illustrated in Figure 4.4. Every address generated the CPU is divided into two parts: **a page number(p) and a offset(d).** . The page number is used as an index into a The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in Figure 4.5

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.

For More Study Materials : https://www.keralanotes.com/

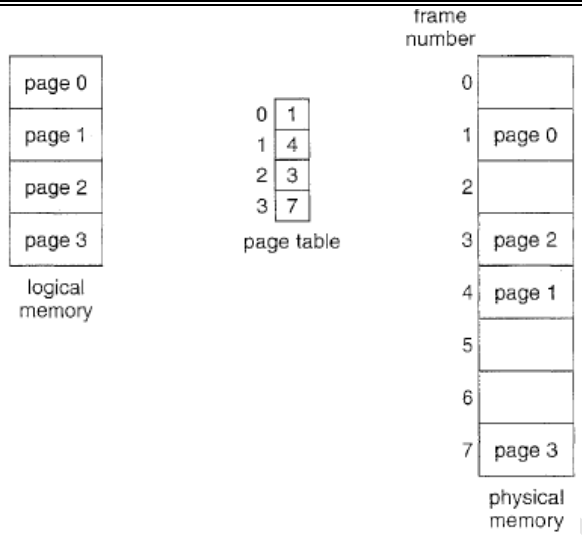
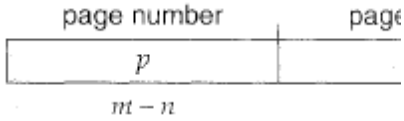


Fig:4.5 Paging model of logical and physical memory.

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



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

For 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 {- (5x4) + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4

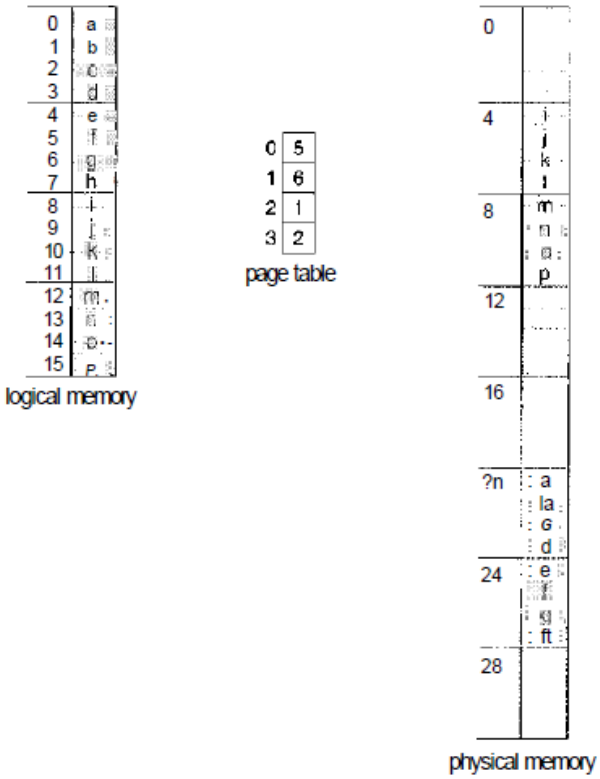


Figure 8.9 Paging example for a 32-byte memory with 4-byte pages.

For More Study Materials : https://www.keralanotes.com/

maps to physical address 24 (= (6x4) + 0). Logical address 13 maps to physical address 9.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded inJo one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on (Figure 4.6).

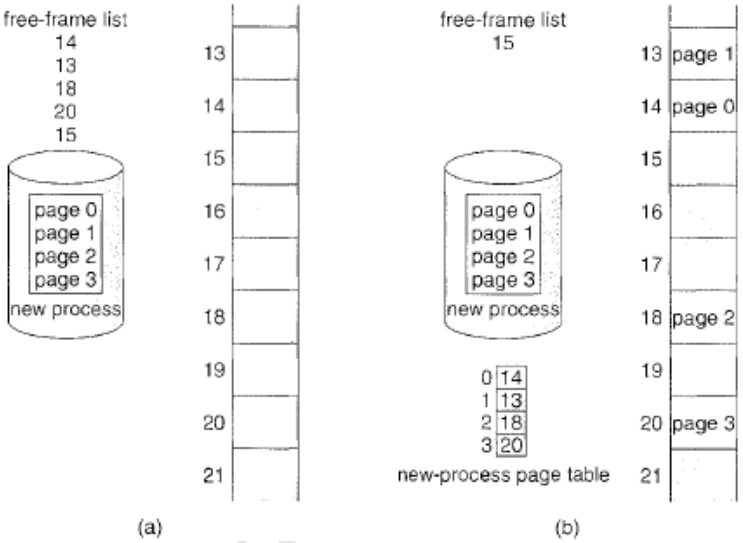


Fig:4.6 Free frames (a) before allocation and (b) after allocation

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 memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs. The difference between the user's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the user and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own. It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.

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.

For More Study Materials : https://www.keralanotes.com/

If a user makes a system call and provides an address as a parameter that address must be mapped to produce the correct physical address. The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents.This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually.

- **Hardware Support**

Each operating system has its own methods for storing page tables. Most allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block.The hardware implementation of the page table can be done in several 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.

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

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

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

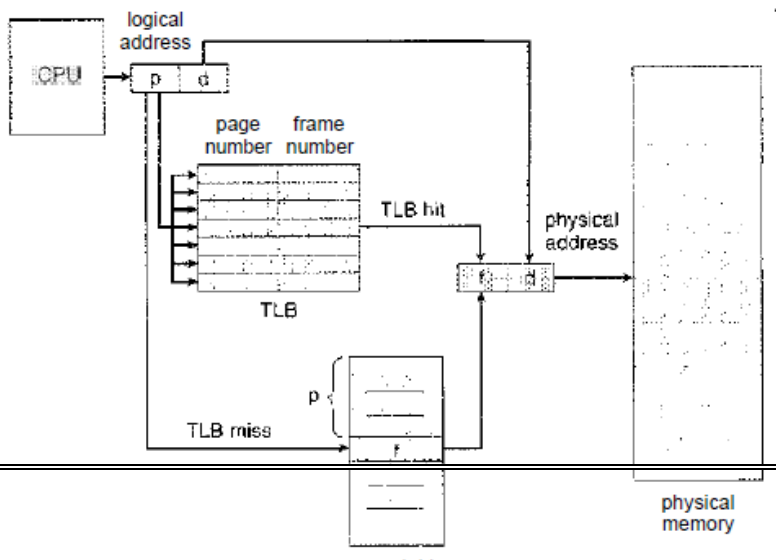


Figure 8.11 Paging hardware with TLB.

For More Study Materials : https://www.keralanotes.com/

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,(**TLB hit**) its frame number is immediately available and is used to access memory. The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.

If the page number is not in the TLB (**TLB miss)** a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory (Figure 8.11). 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 replacement**.

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. **In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously.**

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

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

To find the **effective memory-access time,** we weight each case by its probability: **effective access time = 0.80 x 120 + 0.20 x 220 = 140 nanoseconds.**
In this example, we suffer a 40-percent slowdown in memory-access time (from 100 to 140 nanoseconds).
For a 98-percent hit ratio, we have
effective access time = 0.98 x 120 + 0.02 x 220 = 122 nanoseconds.

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

- **Protection**

For More Study Materials : https://www.keralanotes.com/

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

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).

One additional bit is generally attached to each entry in the page table: a **valid-invalid bit**. When this bit is set to "valid," the associated page is in the process's logical address space or present in the main memory and is thus a legal (or valid) page. When the bit is set to "invalid," the page is not in the process's logical address space or not present in the main memory. Illegal addresses are trapped by use of the valid -invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.

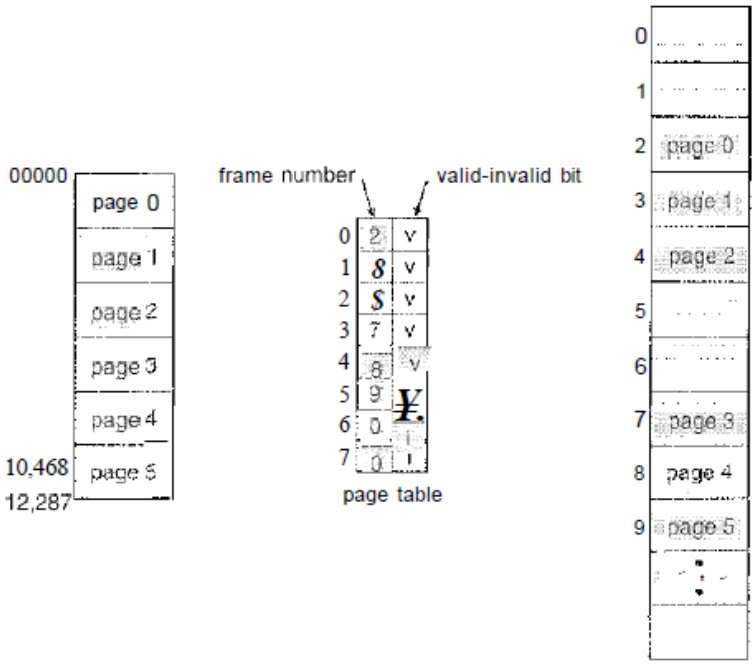


Figure 8.12 Valid (v) or invalid (i) bit in a page table.

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, we get the situation shown in Figure 8.12. Addresses in pages 0,1, 2,3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, will find that the valid-invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

- **Shared Pages**

An advantage of paging is the possibility of sharing common code. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. If the code is **reentrant (or pure code)** however, I tcan be shared, as shown in Figure

For More Study Materials : https://www.keralanotes.com/

4.7. Here we see a three-page editor-each page 50 KB in size (the large page size is used to simplify the figure)-being shared among three processes. Each process has its own data page.

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. 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.

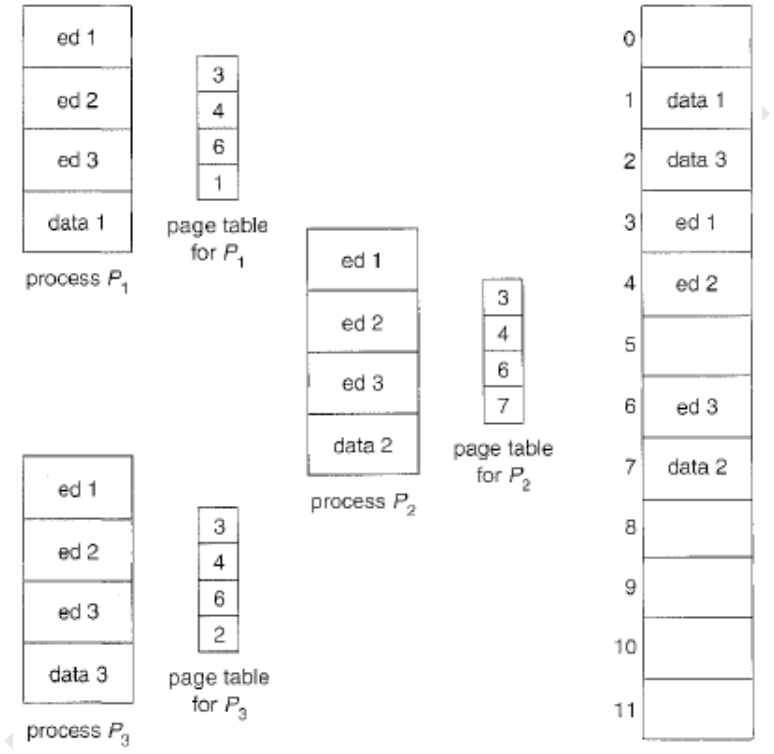


Fig:4.7 Sharing of code in a paging environment.

6. DEMAND PAGING

Virtual memory involves the separation of logical memory as perceived by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available. The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory. Physical memory may be organized in page frames and that the physical page frames assigned to a process may not be contiguous. It is up to the **memory management unit (MMU)** to map logical pages to physical page frames in memory.

For More Study Materials : https://www.keralanotes.com/

When an executable program is loaded from disk in to memory it is not necessary to load the entire program in physical memory at program execution time. We may not initially *need* the entire program in memory. Consider a program that starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for *all* options, regardless of whether an option is ultimately selected by the user or not. 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 only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory.

A demand-paging system is similar to a paging system with swapping (Fig 4.14) 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, however, we use a **lazy swapper**. A lazy swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, use of the term *swapper* is technically incorrect. A swapper manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process. We thus use *pager,* rather than *swapper,* in connection with demand paging.

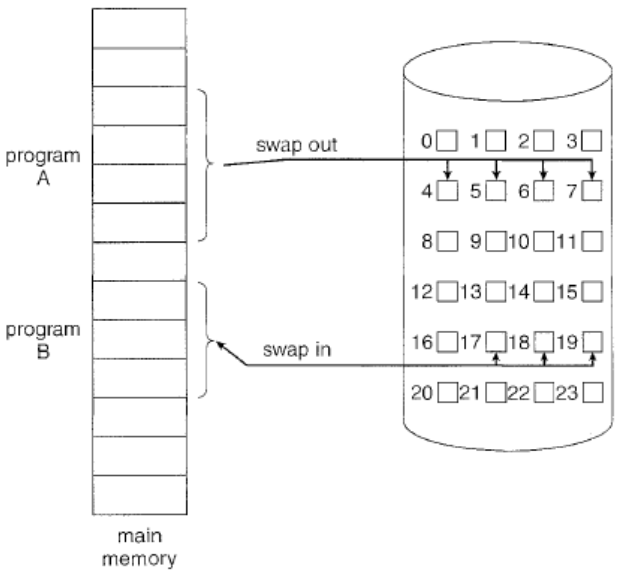


Fig 4.14:Transfer of a paged memory to contiguous disk space.

- **Basic Concepts**

With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. The **valid -invalid bit scheme** can be used for this purpose. This time, however, 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. This situation is depicted in Figure 4.15.

For More Study Materials : https://www.keralanotes.com/

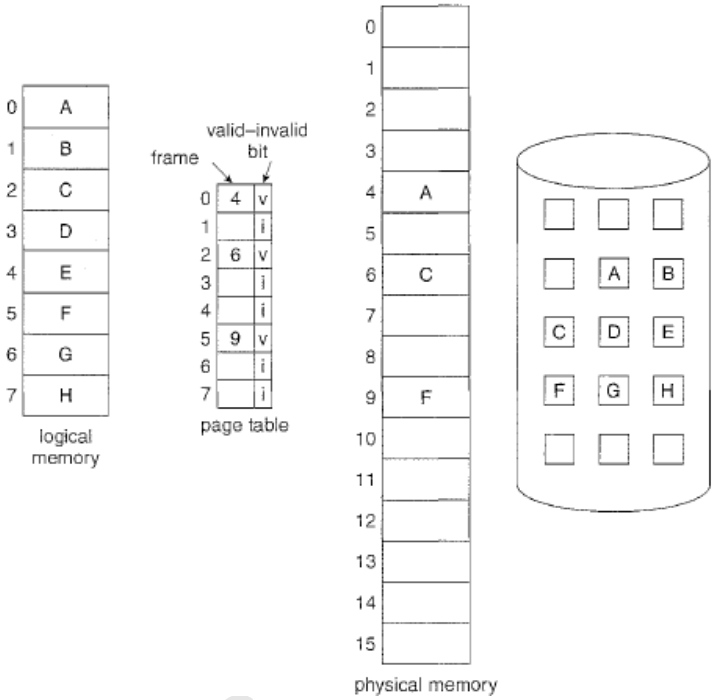


Fig 4.15:Page table when some pages are not in main memory.

If the process tries to access a page that was not brought into memory (access to a page marked as 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 (Figure 4.16):

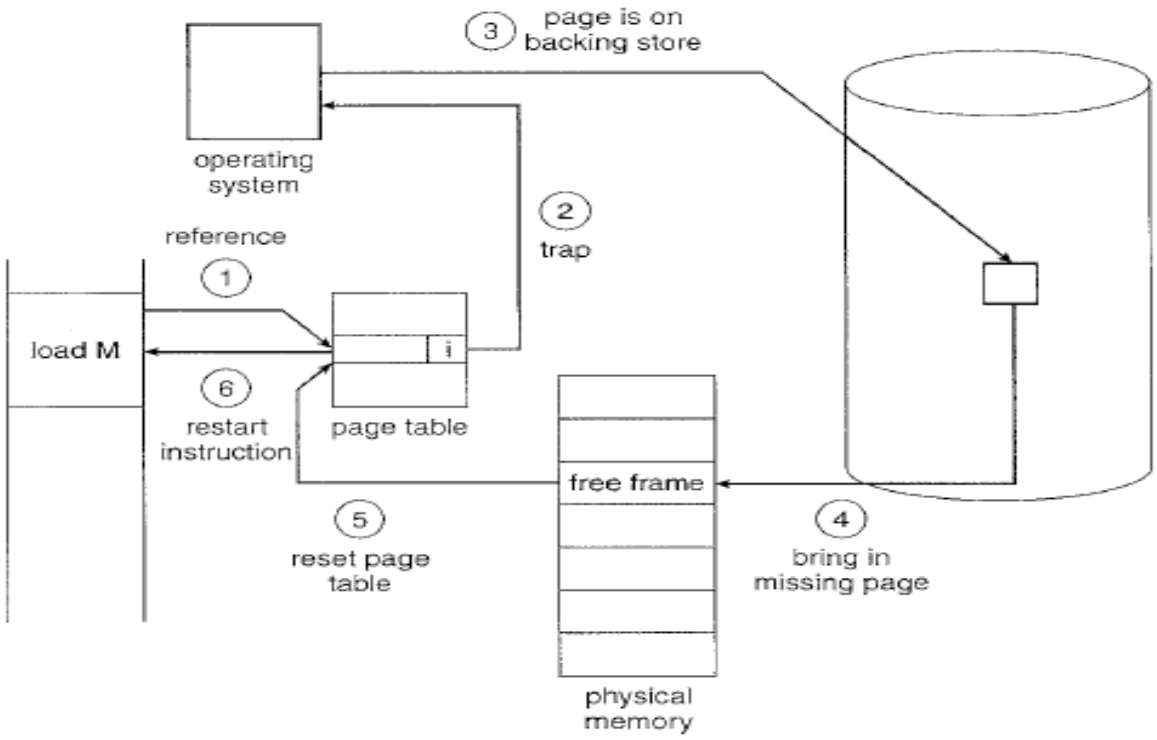


Fig 4.16: Steps in handling a page fault.

1. We check an internal table (usually kept with the process control block)for this process to determine whether the reference was a valid or aninvalid memory access.

For More Study Materials : https://www.keralanotes.com/

2. If the reference was invalid, we terminate the process. If it was valid, butwe 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 newlyallocated frame.
5. When the disk read is complete, we modify the internal table kept withthe 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 processcan now access the page as though it had always been in memory.

In the extreme case, we can start executing a process with *no* pages inmemory. When the operating system sets the instruction pointer to the firstinstruction of the process, which is on a non-memory-resident page, the processimmediately faults for the page. After this page is brought into memory, theprocess continues to execute, faulting as necessary until every page that itneeds is in memory. At that point it can execute with no more faults. This scheme is **pure demand paging**, that is never bring a page into memory until it isrequired.

The hardware to support demand paging is the same as the hardware forpaging and swapping:

- **Page table**: This table has the ability to mark an entry invalid through avalid - invalid bit or a special value of protection bits.
- **Secondary memory**: This memory holds those pages that are not presentin main memory. The secondary memory is usually a high-speed disk. It isknown as the swap device, and the section of disk used for this purpose isknown as**swap space**.

A crucial requirement for demand paging is the ability to restart anyinstruction after a page fault. Because we save the state (registers, conditioncode, instruction counter) of the interrupted process when the page faultoccurs, we must be able to restart the process in *exactly* the same place andstate, except that the desired page is now in memory and is accessible. In mostcases, this requirement is easy to meet. A page fault may occur at any memoryreference. If the page fault occurs on the instruction fetch, we can restart byfetching the instruction again. If a page fault occurs while we are fetching anoperand, we must fetch and decode the instruction again and then fetch the operand.

As a worst-case example, consider a three-address instruction such as ADD the content of A to B, placing the result in C. These are the steps to execute this instruction:

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.

For More Study Materials : https://www.keralanotes.com/

5. Store the sum in C.

If we fault when we try to store in C (because C is in a page not currently in memory), we will have to get the desired page, bring it in, correct the page table, and restart the instruction. The restart will require fetching the instruction again, decoding it again, fetching the two operands again, and then adding again. However, there is not much repeated work (less than one complete instruction), and the repetition is necessary only when a page fault occurs.

- **Performance of Demand Paging**

Demand paging can significantly affect the performance of a computer system.To see why, let's compute the effective access time for a demand-pagedmemory. For most computer systems, the memory-access time, denoted *ma,*ranges from 10 to 200 nanoseconds. As long as we have no page faults, theeffective access time is equal to the memory access time. If, howevera pagefault occurs, we must first read the relevant page from disk and then access thedesired word. Let *p* be the probability of a page fault (0≤*p* ≤1). We would expect *p* tobe close to zero-that is, we would expect to have only a few page faults. Theeffective access time is then

effective access time=[(1 - *p) *ma*]+ [*p* *page fault time]

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault. '
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:

a. Wait in a queue for this device until the read request is serviced.

b. Wait for the device seek and /or latency time.

c. Begin the transfer of the page to a free frame.

6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

Not all of these steps are necessary in every case. For example, we are assuming that, in step 6, the CPU is allocated to another process while the I/O occurs.

For More Study Materials : https://www.keralanotes.com/

This arrangement allows multiprogramming to maintain CPU utilization but requires additional time to resume the page-fault service routine when the I/O transfer is complete.

In any case, we are faced with three major components of the page-fault service time:

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

The first and third tasks can be reduced, with careful coding, to several hundred instructions.

keralanotes.com