The operating system must balance the needs of the various processes with the availability of the different types of memory, moving data in blocks (called pages) between available memory as the schedule of processes dictates.

Systems for managing memory can be divided into two categories: the system of moving processes back and forth between main memory and disk during execution (known as swapping and paging) and the process that does not do so (that is, no swapping and ping).

3.2.1 Requirements

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 memory address is generated by a program. We are interested in only the sequence of memory addresses generated by the running program.

Binding of Instructions and Data to Memory

Usually, a program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The input queue is formed by the collection of processes on the disk that is waiting to be brought into memory for execution.

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.

Address binding of instructions and data to memory addresses can happen at three different stages.

1. *Compile time:* If you know at compile time where the process will reside in memory, then absolute code can be generated. If sometime later, the starting location changes, then it will be necessary to recompile this code.

2. *Load time:* If it is not known at compile time where the process will reside in memory, then the compiler must generate re locatable code.

3. *Execution time:* If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

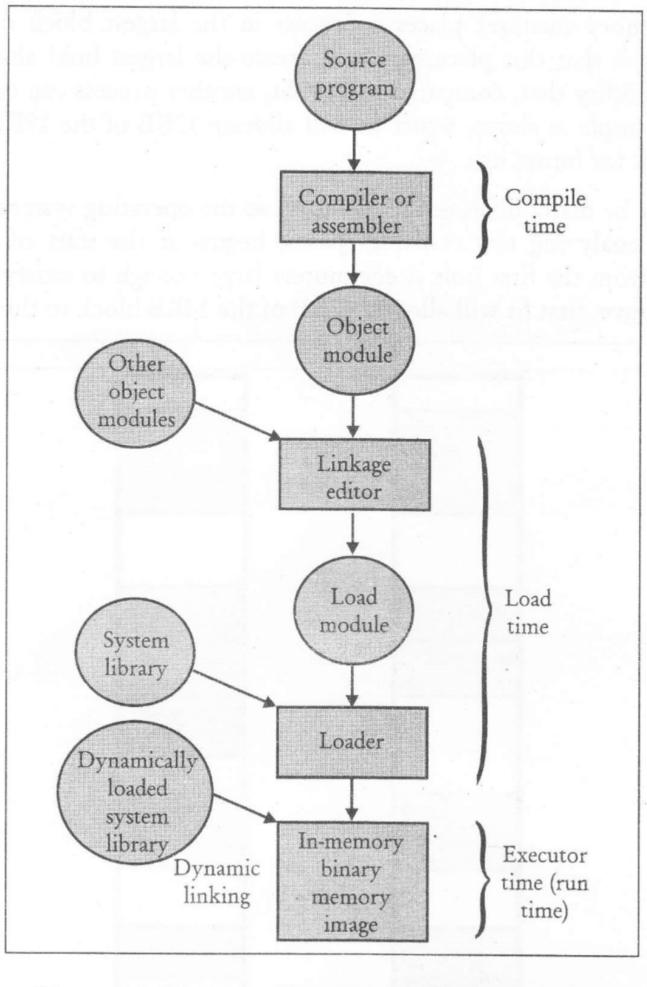

Figure 3.2: Multi-step Processing of a User Program

3.3 CONTIGUOUS MEMORY MANAGEMENT

The real challenge of efficiently managing memory is seen in the case of a system which has multiple processes running at the same time. Since primary memory can be space-multiplexed, the memory manager can allocate a portion of primary memory to each process for its own use. However, the memory manager must keep track of which processes are running in which memory locations, and it must also determine how to allocate and deallocate available memory when new processes are created and when old processes complete execution. While various different strategies are used to allocate space to processes competing for memory, three of the most popular are Best fit, Worst fit, and First fit. Each of these strategies is described below:

1. *Best fit:* The allocator places a process in the smallest block of unallocated memory in which it will fit. For example, suppose a process requests 12KB of memory and the memory manager currently has a list of unallocated blocks of 6KB, 14KB, 19KB, 11KB, and 13KB blocks. The best-fit strategy will allocate 12KB of the 13KB block to the process.

2. *Worst fit:* The memory manager places a process in the largest block of unallocated memory available. The idea is that this placement will create the largest hold after the allocations, thus increasing the possibility that, compared to best fit, another process can use the remaining space. Using the same example as above, worst fit will allocate 12KB of the 19KB block to the process, leaving a 7KB block for future use.

3. *First fit:* There may be many holes in the memory, so the operating system, to reduce the amount of time it spends analyzing the available spaces, begins at the start of primary memory and allocates memory from the first hole it encounters large enough to satisfy the request. Using the same example as above, first fit will allocate 12KB of the 14KB block to the process.

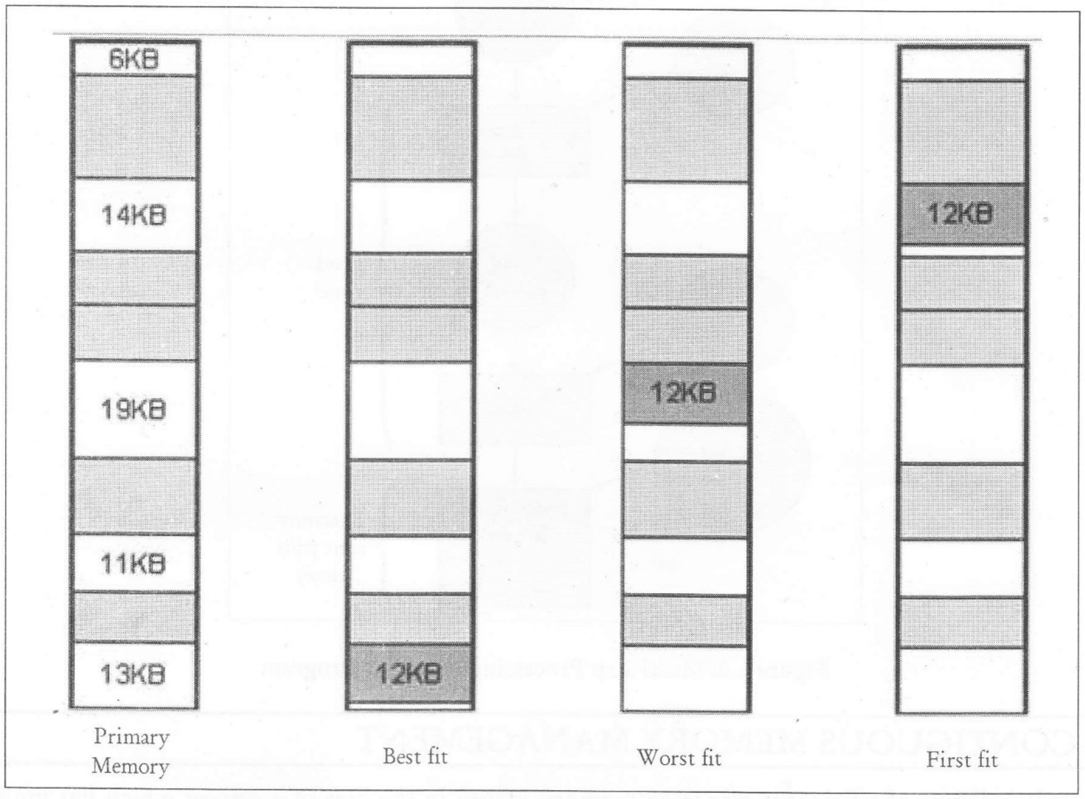

Figure 3.3: Diagram of Best Fit, Worst Fit and First Fit Memory Allocation Method

Notice in the Figure above that the Best fit and first fit strategies both leave a tiny segment of memory unallocated just beyond the new process. Since the amount of memory is small, it is not likely that any new processes can be loaded here. This condition of splitting primary memory into segments as the memory is allocated and deallocated is known as fragmentation. The Worst fit strategy attempts to reduce the problem of fragmentation by allocating the largest fragments to new processes. Thus, a larger amount of space will be left as seen in the Figure above.

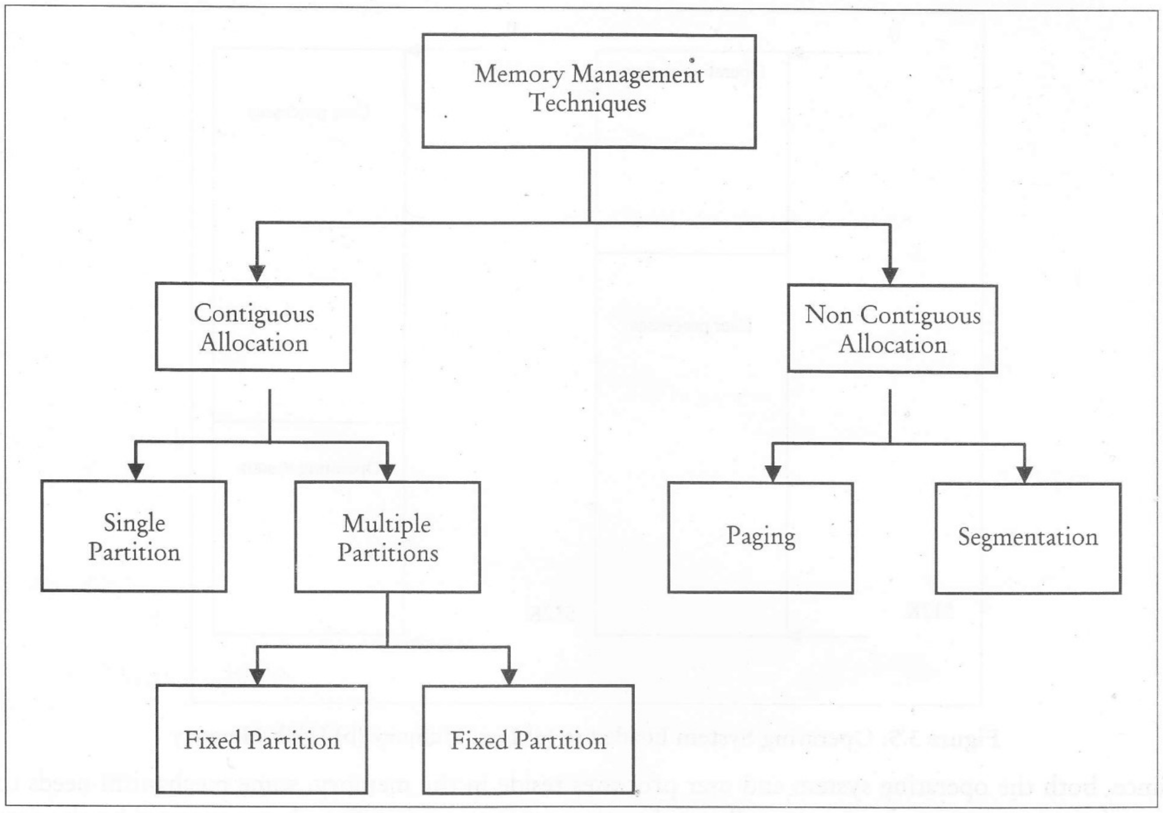

Figure 3.4: Memory Management Techniques

3.3.1 Single Partition Memory Allocation

The memory is divided into two partitions, one for the resident operating system, and one for the user process. Since operating system is also a program, it must also be loaded into the memory for its operation. For this purpose, memory is divided into two partitions, one for the resident operating system and the other for the user processes. The operating system may be either low memory (memory having low addresses) or into the high memory Figure 3.5.

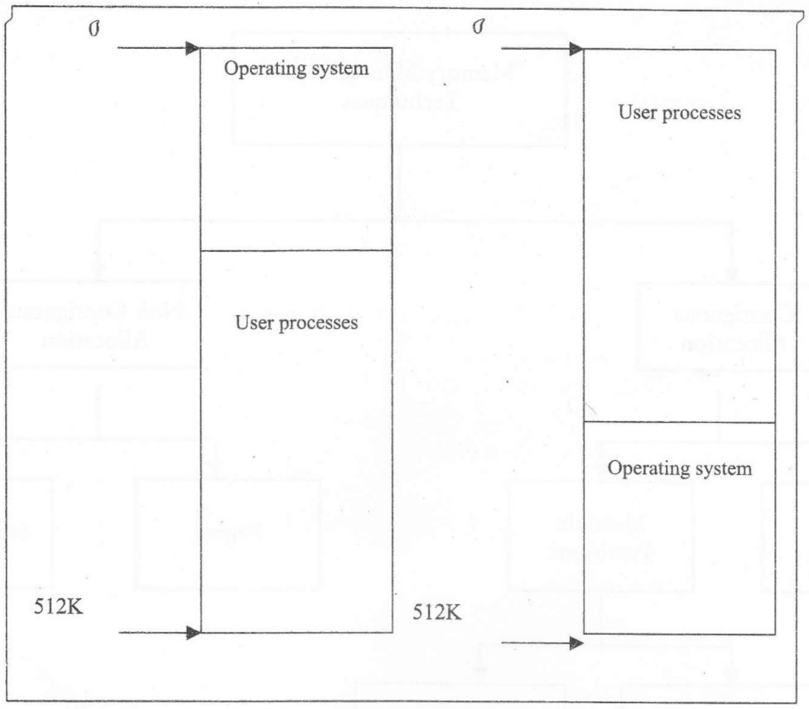

Figure 3.5: Operating System Loaded in (a) Low Memory (b) High Memory

Since, both the operating system and user processes reside in the memory, some mechanism needs to be enforced to protect the memory allocated to the operating system from being accessed by the user process. This protection may be enforced with the help of two registers – relocation register and limit register. The relocation register contains the value of the smallest physical address while limit register contains the range of the allowable logical address. The hardware support for this type of partitioning is shown in Figure 3.6.

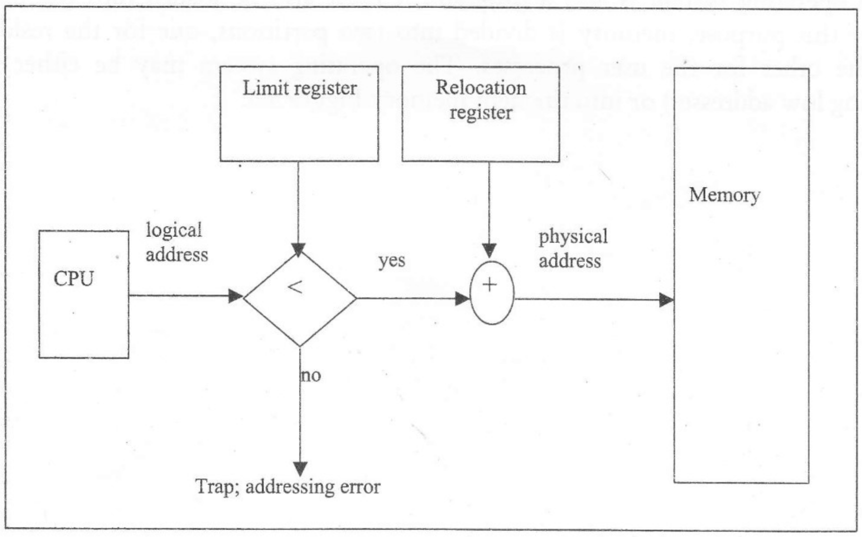

Figure 3.6: Hardware Support for Relocation and Limit Registers

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 then sent to memory.

In a multi-programming environment, it is desirable that more than one user processes may be brought into the memory. For this purpose, memory may be divided into multiple partitions of fixed size. Each of these partitions may house one user process. Thus, in this scheme, the number of processes residing in the memory is limited by the number of partitions in the memory. 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 other processes.

The operating system keeps a table indicating which partitions of the memory are free and which are occupied. In the beginning all the memory is available to the user process. When a process arrives and needs memory, we search for a chunk of memory (also called hole) large enough to accommodate the process. If we find one, we allocate only as much memory as is needed.

For example, let us take one situation in which we have 2560K memories available and the resident operating system occupies 400K. This leaves 2160K for the user processes. The input queue has five processes as shown below. Also assume that processes have been scheduled on the basis of FCFS.

3.3.2 Multiple Partitions Memory Allocation

Fixed-Partition Memory Allocation

Memory is divided into a no. of fixed sized partitions. Each partition may contain exactly one process. When a partition is free, a process is selected from the input Queue and is loaded into a free partition. Some features of fixed partition memory allocation:

- very easy to implement
- common in old batch processing systems
- well suited to well-known job mix
- presumes largest possible process size
- must reconfigure system for larger processes
- likely to use memory inefficiently
- large internal fragmentation losses
- swapping results in convoys on partitions

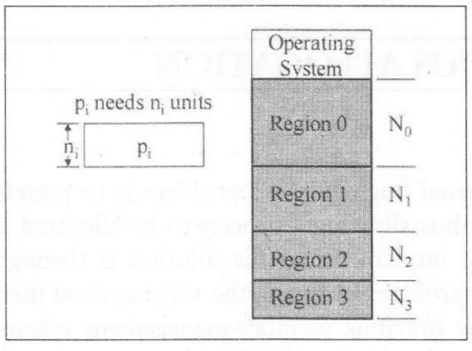

Figure 3.7: Fixed Partition Memory Allocation

Variable Memory Allocation

Initially all memory is available for user processes, and is considered as large block of available memory. When a process arrives and needs memory, we search for a whole large enough for this process and allocate only as much. Some features of variable memory allocation

- start with one large "heap" of memory
- when a process requests more memory
- find a large enough chunk of memory
- carve off a piece of the requested size
- put the remainder back on the free list
- when a process frees memory
- put it back on the free list
- eliminates internal fragmentation losses

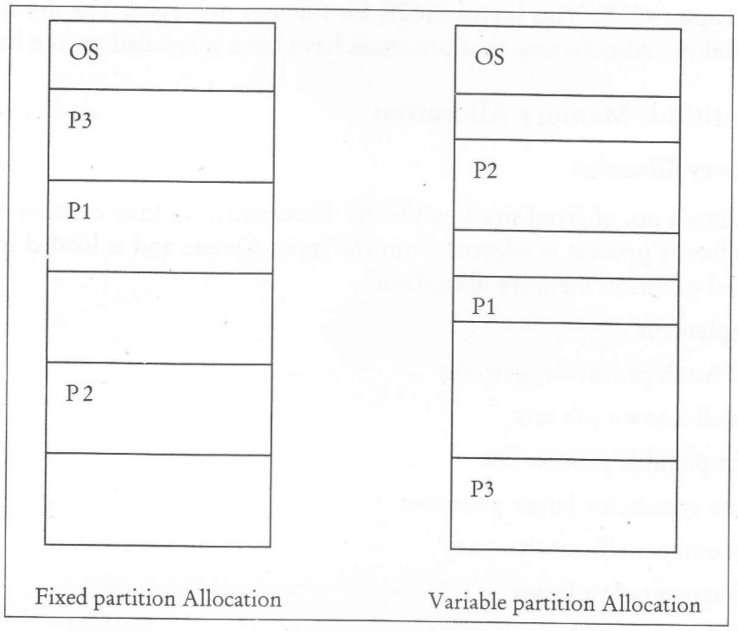

Figure 3.8: Fixed and Variable Partition Allocation

3.4 NON-CONTIGUOUS ALLOCATION

3.4.1 Paging

A possible solution to the external fragmentation problem is to permit the logical address space of a process to be non-contiguous, thus allowing a process to be allocated physical memory wherever the latter is available. One way of implementing this solution is through the use of a paging scheme. 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. The fragmentation problems discussed in connection with main memory are also prevalent with backing store, except that access is much slower, so compaction is impossible. Because of its advantages over the previous methods, paging in its various forms is commonly used in many operating systems.

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.

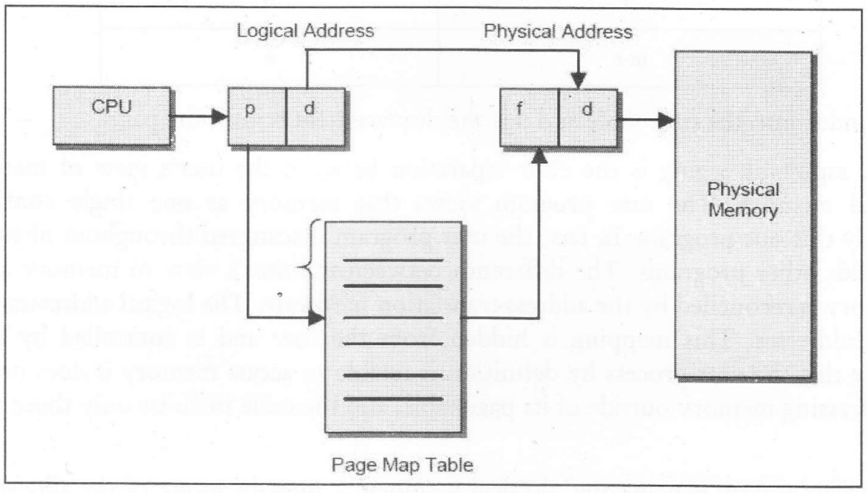

Figure 3.9: Paging Hardware

The hardware support for paging is illustrated in Figure 3.9. 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 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 3.10.

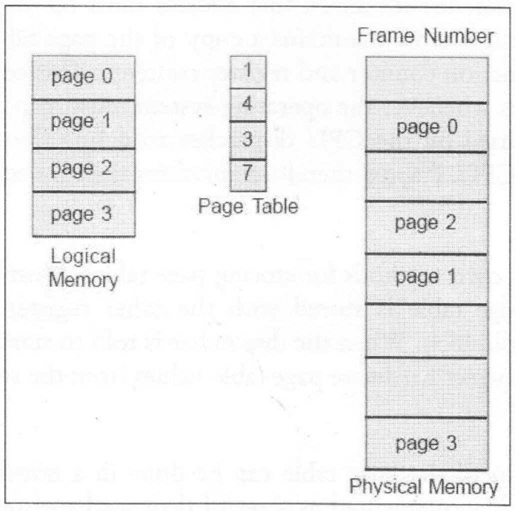

Figure 3.10: Paging Model of Logical and Physical Memory

The page size (like the frame size) is defined by the hardware. The size of page is typically a power of 2 varying between 512 bytes and 8,192 bytes per age, depending on the computer architecture. The selection of a power of 2 s 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 2", and a page size is 2" addressing units (bytes or words), then the high-order m - n its 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:

Page Number	Page Offset
p	d
m-n	n

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

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. 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 use; 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. In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses. If a user makes a system call (to do I/O, for example) and provides an address as a parameter (a buffer, for instance), 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. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging, therefore, increases the context-switch time.

Structure of Page Table

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. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table.

Hardware Support

The hardware implementation of the page table can be done in a number of different 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 8K. The page table, thus, consists of eight entries that are kept in fast registers.

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 for 1. 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. We might as well resort to swapping. The standard solution to this problem is to use a special, small, fast-lookup hardware cache, variously called associative registers or translation look-aside buffers (TLBs). A set of associative registers is built of especially high-speed memory. Each register consists of two parts: a key and a value. When the associative registers are presented with an item, it is compared with all keys simultaneously. If the item is found, the corresponding value field is output. The search is fast; the hardware, however, is expensive. Typically, the number of entries in a TLB varies between 8 and 2048.

Associative registers are used with page tables in the following way. The associative registers contain only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to a set of associative registers that contain page numbers and their corresponding frame numbers. If the page number is found in the associative registers, its frame number is immediately available and is used to access memory. The whole task may take less than 10% longer than it would were an unmapped memory reference used.

If the page number is not in the associative registers, a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory. In addition, we add the page number and frame number to the associative registers, 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. Unfortunately, every time a new page table is selected (for instance, each context switch), the TLB must be flushed (erased) to ensure that the next executing process does not use the wrong translation information. Otherwise, there could be old entries in the TLB that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

The percentage of times that a page number is found in the associative registers is called the hit ratio. An 80% hit ratio means that we find the dared page number in the associative registers 80% of the time. If it tabs 20 nanoseconds to search the associative registers, and 100 nanoseconds to acres memory, then a mapped memory access takes 120 nanoseconds when the page number is in the associative registers. If we fail to find the page number in associative registers (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:

$$\begin{aligned}\text{Effective access time} &= 0.80 \times 120 + 0.20 \times 220\\ &= 140 \text{ nanoseconds.}\end{aligned}$$

In this example, we suffer a 40 per cent slowdown in memory access time (from 100 to 140 nanoseconds). This increased hit rate produces only a 22% slowdown in memory also time.

The hit ratio is clearly related to the number of associative registers. With the number of associative registers ranging between 16 and 512, a hit ratio of 80% to 98% can be obtained. The Motorola 68030 processor (used in AE: Macintosh systems) has a 22-entry TLB. The Intel 80486 CPU (found in some) has 32 registers, and claims a 98% hit ratio.

Protection

Memory protection in a paged environment is accomplished by protection bits that are associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read and write or read-only. Ever 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 the operating system (memory-protection violation). This approach to protection can be expanded easily to provide a finer level of protection. We can create hardware to provide read-only, read-write, or execute-only protection. Or, by providing separate protection bits for each kind of access, any combination of these accesses can be allowed, and illegal attempts will be trapped to the operating system.

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's logical address space, and is thus a legal (valid) page. If the bit is set to "invalid," this value indicates that the page is not in the process's 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.

3.4.2 Segmentation

An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory and the actual physical memory. The user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory. The mapping allows differentiation between logical memory and physical memory.

What is the user's view of memory? Does the user think of memory as a linear array of bytes, some containing instructions and others containing data, or is there some other preferred memory view? There is general agreement that the user or programmer of a system does not think of memory as a linear array of bytes. Rather, the user prefers to view memory as a collection of variable-sized segments, with no necessary ordering among segments (Figure 3.11).

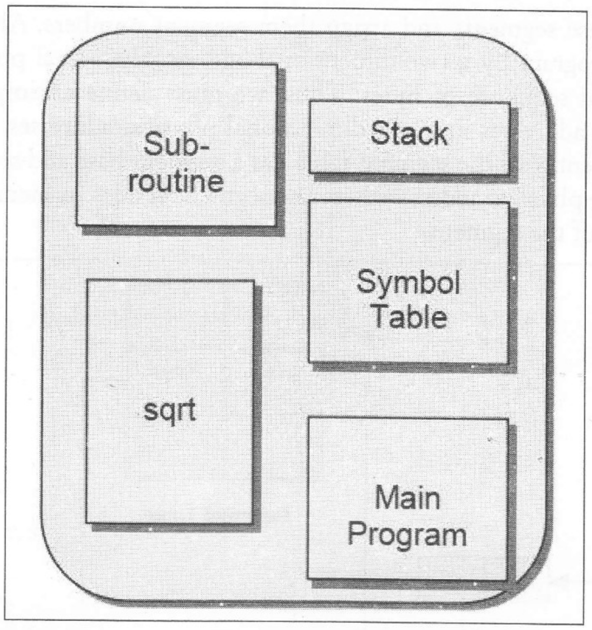

Figure 3.11: User's View of a Program

Consider how you think of a program when you are writing it. You think of it as a main program with a set of subroutines, procedures, functions or modules. There may also be various data structures: tables, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by Mine. You talk about "the symbol table," "function Sqrt," "the main program," without caring what addresses in memory these elements occupy. You are not concerned with whether the symbol table is stored before or after the Sqrt faction. Each of these segments is of variable length; the length is intrinsically defined by the purpose of the segment in the program. Elements within an augment are identified by their offset from the beginning of the segment: the first statement of the program, the seventeenth entry in the symbol table, the fifth instruction of the Sqrt functions, and so on.

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 logical address space is a collection or segments, each segment a name and a length. The addresses specify both the segment name and 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, where the user specified only a single address, which was 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:

<center>< segment-number, offset ></center>

Normally the user program is compiled, and the compiler automatically constructs reflecting the input program. A Pascal compiler might create separate segments for (1) the global variables; (2) the procedure call stack, to store parameters and return addresses; (3) the code portion of each procedure or function; and (4) the local variables of each procedure and function. A FORTRAN compiler might create a separate segment for each common block. Arrays might be assigned separate segments. The loader would take all these segments and assign them segment numbers. Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, we must define an implementation to map two dimensional user-defined addresses into one-dimensional physical addresses. This mapping is affected by a segment table. Each entry of the segment table has a segment base and segment limit. The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.

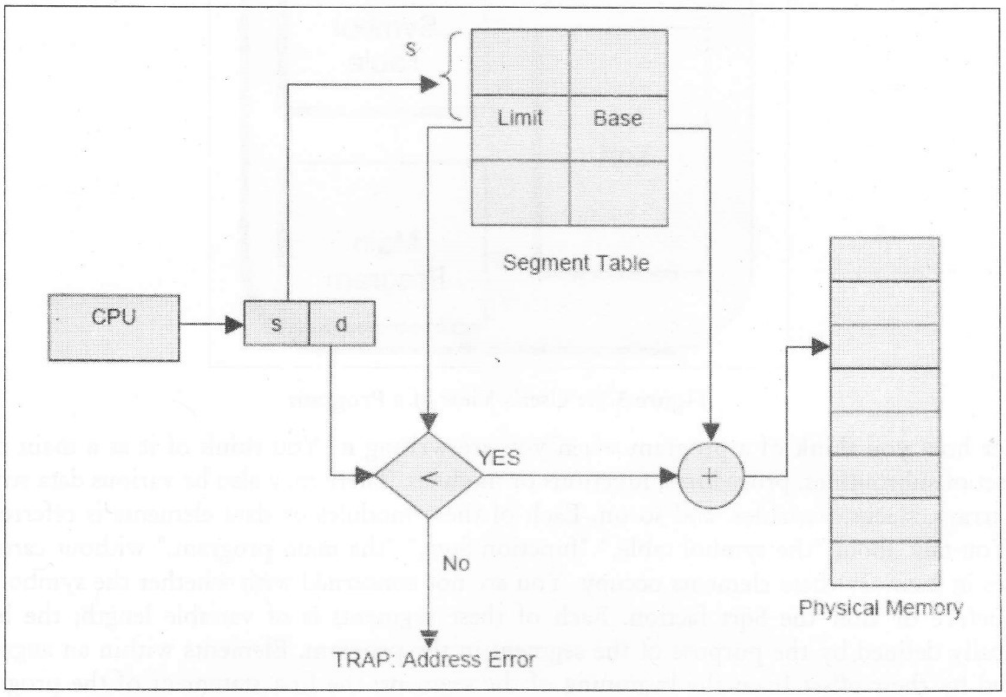

Figure 3.12: Segmentation Hardware

The use of a segment table is illustrated in Figure (3.13). 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 into the segment table. The offset do 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). If this 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.

As an example, consider the situation shown in Figure 3.15. We have 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 (the base) and the length of that segment (the 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 on bytes long.

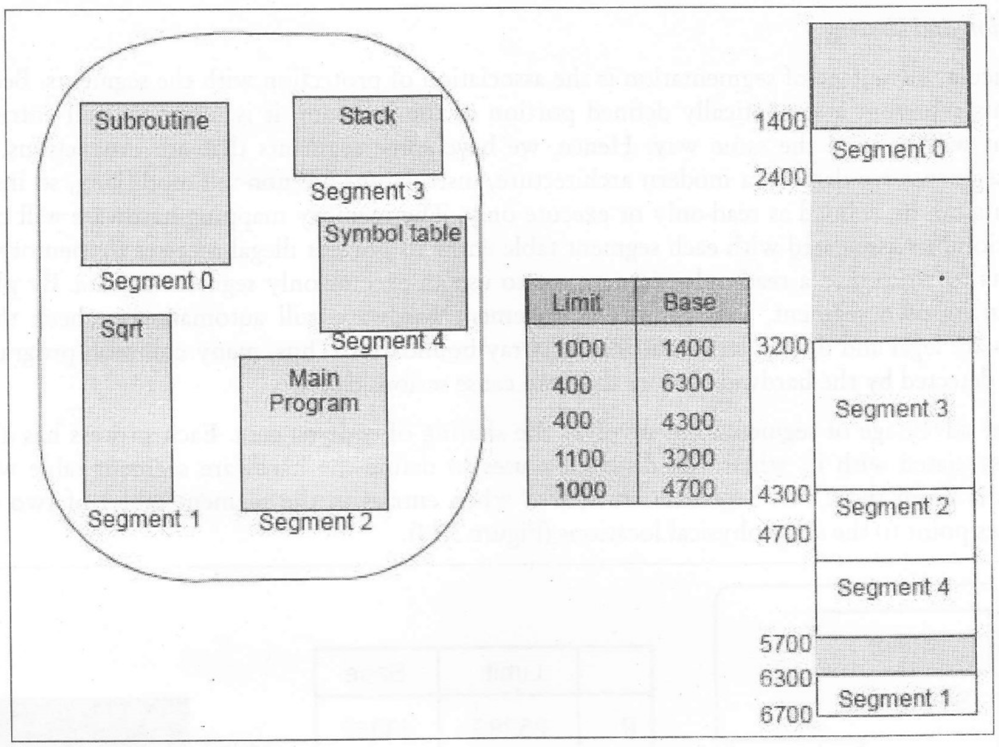

Figure 3.13: Examples of Segmentation

Implementation of Segment Tables

Segmentation is closely related to the partition models of memory management presented earlier, the main difference being that one program may consist of segments. Segmentation is a more complex concept, however, which is why we are describing it after discussing paging. Like the page table, the segment table can be put either in fast registers or in memory. A segment table in registers can be referenced quickly; the addition to the base and comparison with the limit can be done simultaneously to save time.

In the case where a program may consist of a large number of segments, it is not feasible to keep the segment table in registers, so we must keep it in memory. Segment-table bases register (STBR) points to the segment table. Also, because the number of segment used by a program may vary widely, a segment-table length register (STLR) is used. For a logical address (s, d.), we first check that the segment number s is legal (that is, s < STLR). Then, we add the segment number to the STBR, resulting in the address (STBR + s) in memory of the segment-table entry. The entry is read from memory and we proceed as before: Check the offset against the segment length and compute the physical address of the desired byte as the sum of the segment base and offset.

As occurs with paging, this mapping requires two memory reference per logical address, effectively slowing the computer system by a factor of 2 unless something is done. The normal solution is to use a set of associative registers to hold the most recently used segment-table entries. Again, a small set of associative registers can generally reduce the time required for memory accesses to no more than 10 or 15% slower than unmapped memory accesses.

Protection and Sharing

A particular advantage of segmentation is the association of protection with the segments. Because the segments represent a semantically defined portion of the program, it is likely that all entries in the segment will be used the same way. Hence, we have some segments that are instructions, whereas other segments are data. In a modern architecture, instructions are non-self-modifying, so instruction segments can be defined as read-only or execute only. The memory mapping hardware will check the protection bits associated with each segment table entry to prevent illegal accesses to memory, such as attempts to write into a read-only segment, or to use an execute-only segment as data. By placing an array in its own segment, the memory-management hardware will automatically check that array indexes are legal and do not stray outside the array boundaries. Thus, many common program errors will be detected by the hardware before they can cause serious damage.

Another advantage of segmentation involves the sharing of code or data. Each process has a segment table associated with it, which the dispatcher uses to define the hardware segment table when this process is given the CPU. Segments are shared when entries in the segment tables of two different processes point to the same physical locations (Figure 3.14).

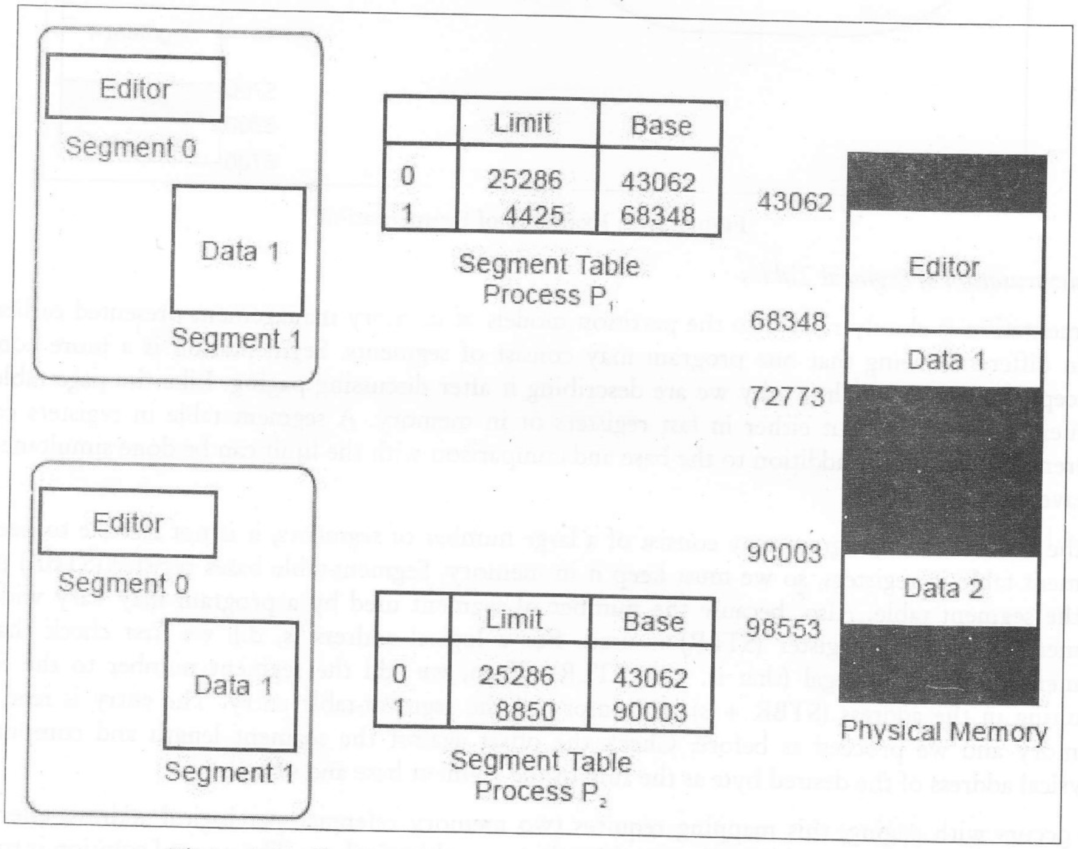

Figure 3.14: Sharing of Segments in a Segmented Memory System

The sharing occurs at the segment level. Thus, any information can be shared if it is defined to be a segment. Several segments can be shared, so program composed of several segments can be shared.

For example, consider the use of a text editor in a time-sharing system A complete editor might be quite large, composed of many segments. The segments can be shared among all users, limiting the physical memory needed to support editing tasks. Rather than n copies of the editor, we need only one copy. For each user, we still need separate, unique segments to store local variables. These segments, of course, would not be shared.

It is also possible to share only parts of programs. For example, common subroutine packages can be shared among many users if they are defined sharable, read-only segments. Two FORTRAN programs, for instance, may use the same Sqrt subroutine, but only one physical copy of the Sqrt routine would be needed.

Although this sharing appears simple, there are subtle considerations. Code segments typically contain references to themselves. For example, a conditional jump normally has a transfer address. The transfer address is a segment number and offset. The segment number of the transfer address will be the segment number of the code segment. If we try to share this segment, all sharing processes must define the shared code segment to have the same segment number.

For instance, if we want to share the Sqrt routine, and one process wants to make segment 4 and another wants to make it segment 17, how should the Sqrt routine refer to itself? Because there is only one physical copy of Sqrt, it must refer to itself in the same way for both users-it must have a unique segment number. As the number of users sharing the segment increases, so does the difficulty of finding an acceptable segment number. Read-only data segments that contain no physical pointers may be shown as different segment numbers, as may code segments that refer to them not directly, but rather only indirectly. For example, conditional branches that specify the branch address as an offset from the current program counter or relative to a register containing the current segment number would allow code to avoid direct reference to the current segment number.

Segmentation with Paging

Both paging and segmentation have their advantages and disadvantages. In fact, of the two most popular microprocessors now being used, the Motorola 68000 line is designed based on a flat address space, whereas the Intel 80x8 family is based on segmentation. Both are merging memory models towards a mixture of paging and segmentation. It is possible to combine these two schemes to improve on each. This combination is best illustrated by two different architectures - the innovative but not widely used MULTICS system and the Intel 386.

3.5 VIRTUAL MEMORY MANAGEMENT SYSTEMS

So far, we have discussed various memory-management strategies that have been used in computer systems. Each of these strategies has the same goal - to keep as many processes in memory as possible, simultaneously to allow multi-programming. However, these strategies tend to require that the entire process to be in memory before the process can execute. Obviously, because of the limited size of physical memory available in a system, these schemes are not very optimal. The largest process (program) that can be executed at one time is limited by the size of physical memory.

For instance, if 16MB RAM (physical memory) is available in the system. Assuming that the operating system itself occupies 4MB, then the no process larger than 12MB can be executed on this system. The total number of processes that can be executed simultaneously is also limited to total space of 12MB, in this case. This limitation may be overcome by another memory management technique - Virtual Memory System.

Virtual memory is a technique that permits the execution of processes that may not be completely resident in the memory. The main advantage of this scheme is that programs can be larger than physical memory. Further, it views main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. It, as a matter of fact augments the main memory with secondary storage devices (like hard disks). This technique frees programmers from concern over memory storage limitations. Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly. In this chapter, we discuss virtual memory in the form of demand paging, and examine its complexity and cost.

The memory-management algorithms of discussed so far, are necessary because of one basic requirement: the instructions being executed must be in physical memory. The first approach to meeting this requirement is to place the entire logical address space in physical memory. Overlays and dynamic loading can help ease this restriction, but they generally require special precautions and extra effort by the programmer. This seems both necessary and reasonable, but it is also unfortunate, since it limits the size of a program to the size of physical memory.

In fact, an examination of real programs shows us that, in many cases, the entire program is not needed for execution. For instance;

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements at execution time. An assembler symbol table may have room for 4000 symbols, although the average program has less than 300 symbols.
- Not all sub-routines are executed always. Certain options and features of a program may be used rarely. For instance, the routines on U.S. government computers that balance the budget have not been used in years.
- Even in those cases where the entire program may be needed at the same time (such is the case with overlays, for example).
- The ability to execute a program that is only partially in memory would have many benefits:
- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large virtual address space, simplifying the programming task. The users do not have to worry about the amount of memory available for the programs.
- Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and consequently, throughput, but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.

Thus, running a program that is not entirely in memory would benefit both the system, and the user. Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure 3.15). Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available, or about what code can be placed in overlays, but can concentrate instead on the problem to be programmed. On systems, which support virtual memory, overlays have virtually disappeared, therefore.

Virtual memory is commonly implemented by Demand Paging. It can also be implemented in a segmentation system. Several systems provide a paged segmentation scheme, where segments are broken into pages. Thus, the user view is segmentation, but the operating system can implement this view with demand paging. Demand segmentation can also be used to provide virtual memory. Burroughs' computer systems have used demand segmentation. The IBM OS/2 operating system also uses demand segmentation. However, segment-replacement algorithms are more complex than the page-replacement algorithms because the segments have variable sizes.

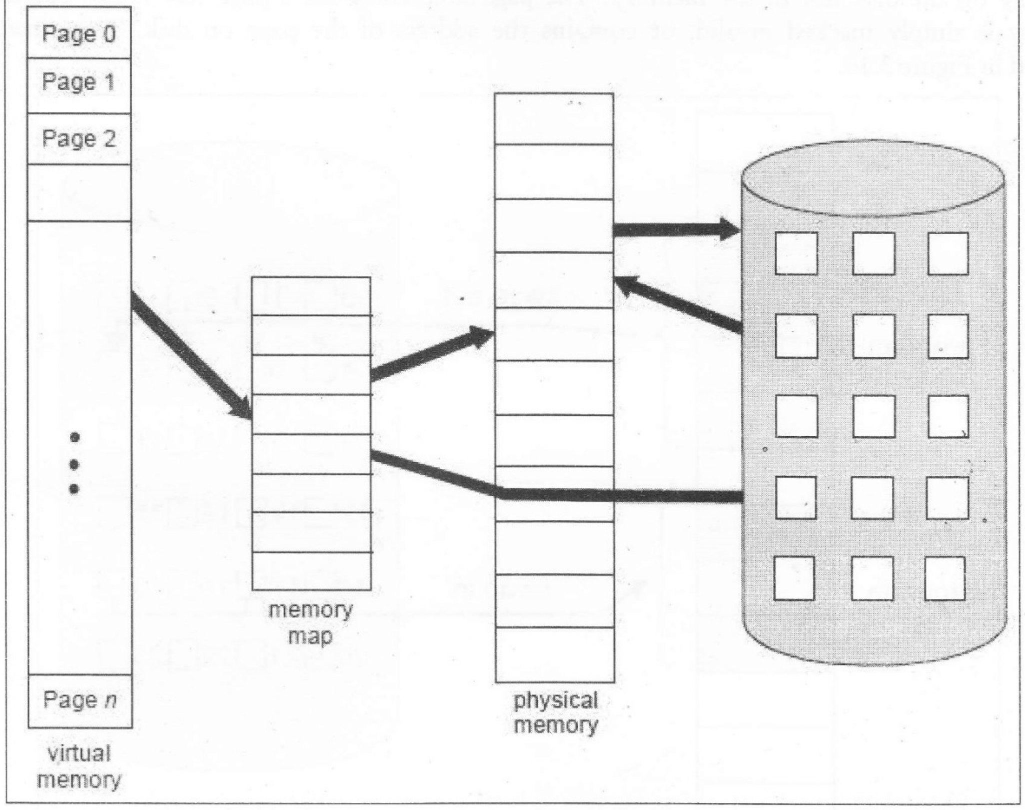

Figure 3.15: Diagram Showing Virtual Memory Larger than Physical Memory

3.5.1 Demand Paging

A demand-paging system is similar to a paging system, discussed earlier, with a little difference that it uses — swapping. Processes reside on secondary memory (which is 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*, which swaps a page into memory only when that page is needed. Since we are now viewing a process as a sequence of pages, rather than one large contiguous address space, the use of the term swap will not technically correct. A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. We shall thus use the term pager, rather than swapper, in connection with demand paging.

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

With this scheme, we need some form of hardware support to distinguish between those pages that are in memory and those pages that are on the disk. The valid-invalid bit scheme described earlier can be used for this purpose. This time, however, when this bit is set to "valid", this value indicates that the associated page is both legal and in memory. If the bit is set to "invalid," this value indicates that 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 not in the memory. The page-table entry for a page that is not currently in memory is simply marked invalid, or contains the address of the page on disk. This situation is depicted in Figure 3.16.

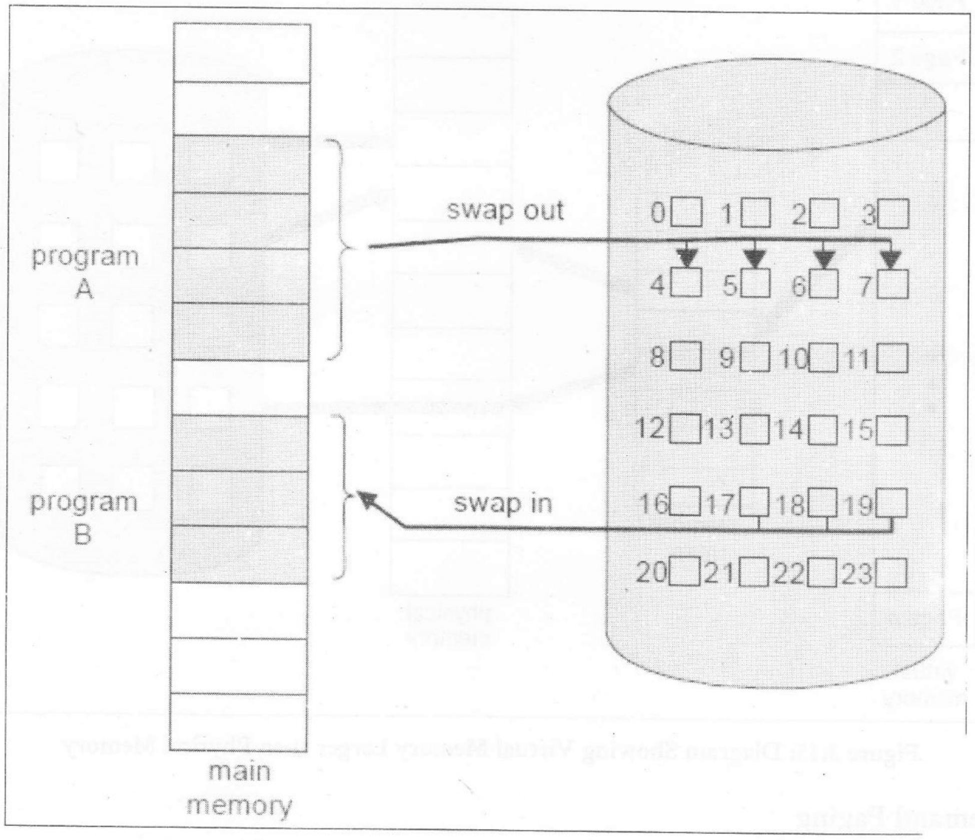

Figure 3.16: Transfer of a Paged Memory to Contiguous Disk Space

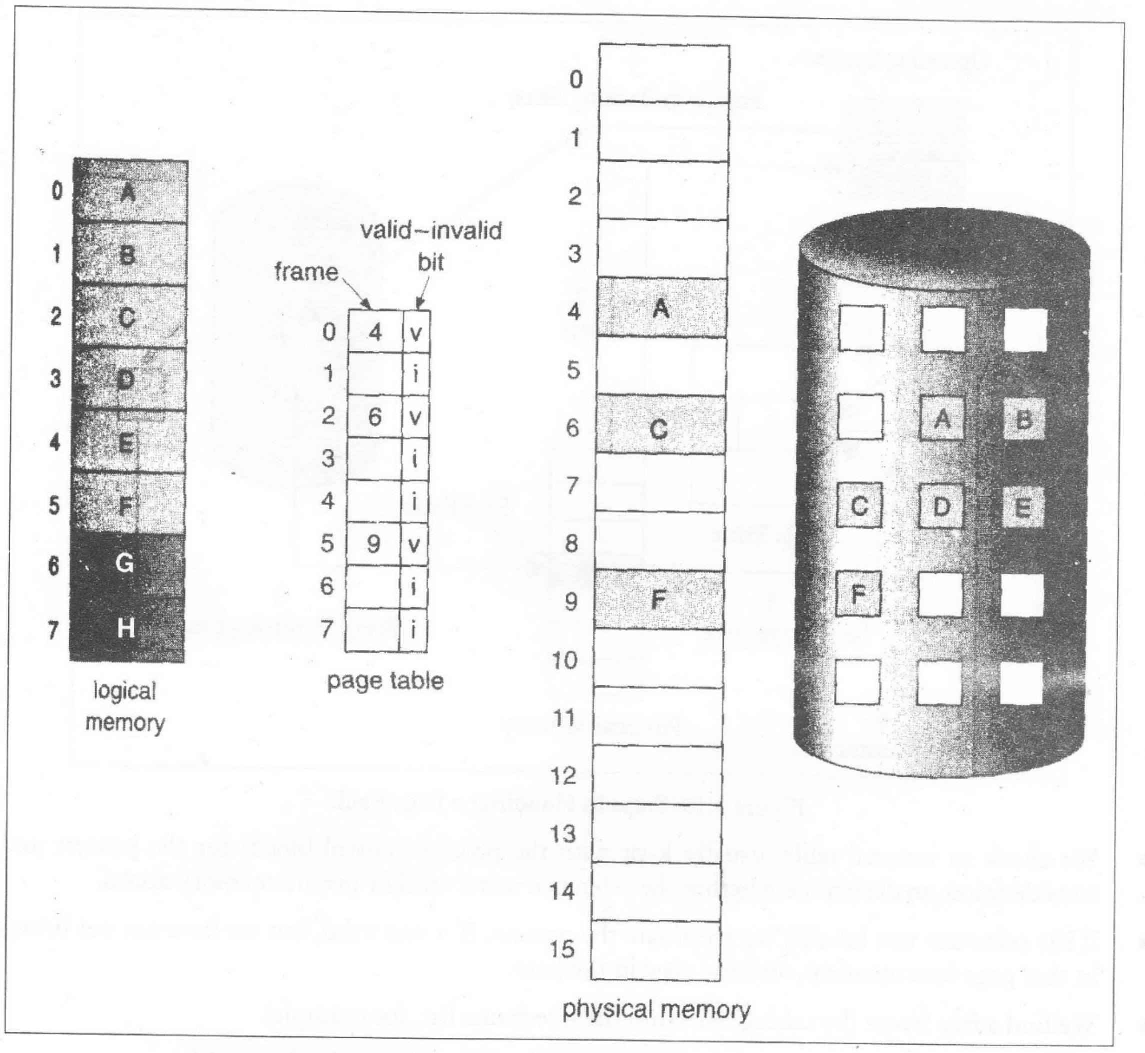

Figure 3.17: Page Table when Some Pages are not in Main Memory

Notice that marking a page invalid will have no effect if the process never attempts to access that page. Hence, if we guess right and only those pages that are actually needed are brought into the memory, the process will run exactly as though we had brought in all pages. While the process executes and accesses pages that are memory resident, execution proceeds normally.

But what happens if the process tries to use a page that was not brought into memory? Access to a page marked invalid causes a page-fault trap. 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 (in an attempt to minimize disk-transfer overhead and memory requirements), rather than an invalid address error as a result of an attempt to use an illegal memory address (such as an incorrect array subscript). We must therefore, correct this oversight. The procedure for handling this page fault is simple (Figure 3.20):

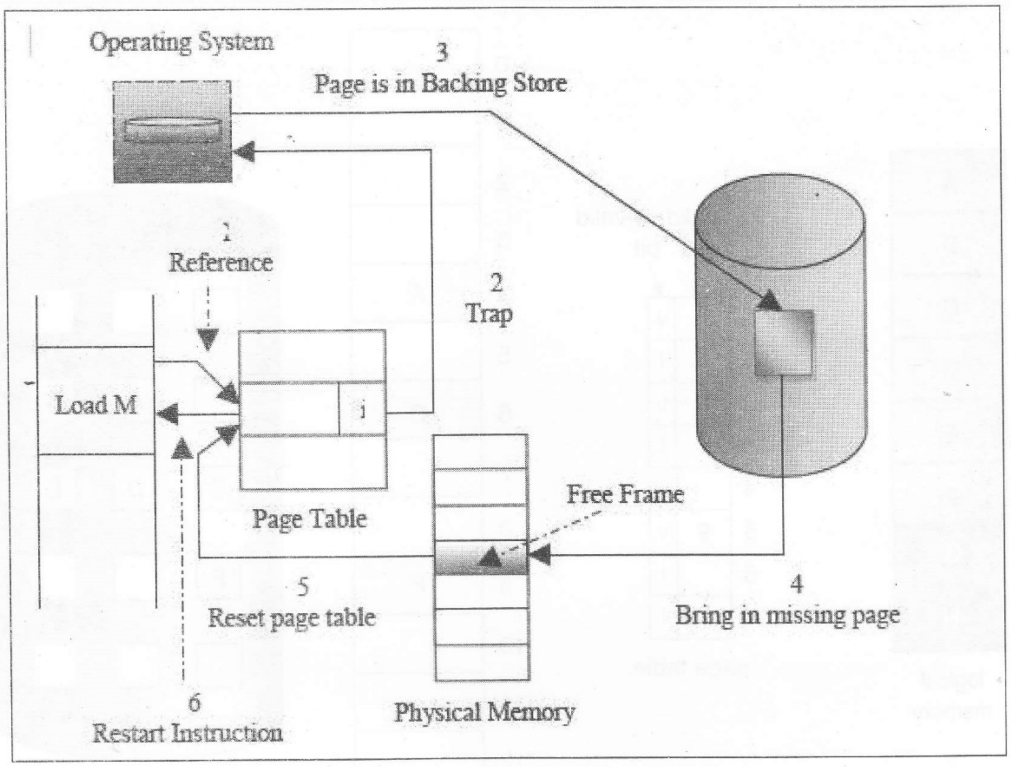

Figure 3.18: Steps in Handling a Page Fault

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

It is important to realize that, because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we can restart the process in exactly the same place and state, except that the desired page is now in memory and is accessible. In this way, we are able to execute a process, even though portions of it are not (yet) in memory. When the process tries to access locations that are not in memory, the hardware traps the operating system which is known as "page fault". The operating system reads the desired page into memory and restarts the process as though the page had always been in memory.

In the extreme case, we could start executing a process with no pages in memory. When the operating system sets the instruction pointer to the first instruction of the process would immediately fault for the page. After this page was brought into memory due to recourse taken in response to this page fault, the process would continue to execute, faulting as necessary until every page that it needed was actually in memory. At that point, it could execute with no more faults. This scheme is pure demand paging: never bring a page into memory until it is required.

Theoretically, some programs may access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance because of the overheads involved. Fortunately, analyses of running processes show that this behavior is exceedingly unlikely. Programs tend to have locality of reference, which results in reasonable performance from demand paging.

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

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

In addition to this hardware support, considerable software is needed, as we shall see. Some additional architectural constraints must be imposed. A crucial issue is the need to be able to restart any instruction after a page fault. In most cases, this requirement is easy to meet. A page fault could occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand not an instruction, then we must re-fetch the instruction, decode it again, and then fetch the operand.

As a worst-case scenario, consider a three-address instruction such as ADD the content of A to B placing the result in C. The steps to execute this instruction would be

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.
5. Store the sum in C.

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

The major difficulty occurs when one instruction may modify several different locations. For example, consider the IBM System 360/370 MVC (move character) instruction, which can move up to 256 bytes from one location to another (possible overlapping) location. If either block (source or destination) straddles a page boundary, a page fault might occur after the move is partially done. In addition, if the source and destination blocks overlap, the source block may have been modified, in which case we cannot simply restart the instruction.

This problem can be solved in two different ways. In one solution, the micro-code computes attempts to access both ends of both blocks. If a page fault is going to occur, it will happen at this step, before anything is modified. The move can then take place, as we know that no page fault can occur, since all the relevant pages are in memory. The other solution uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are written back into memory before the trap occurs. This action restores memory to its state before the instruction was started, so that the instruction can be repeated.

A similar architectural problem occurs in machines that use special addressing modes, including auto-decrement and auto-increment modes (for example, the PDP-11). These addressing modes use a register as indicated. Auto-decrement automatically decrements the register before using its contents as the operand address; auto-increment automatically increments the register after using its contents as the operand address. Thus, the instruction

MOV (R2) + , - (R3)

copies the contents of the location pointed to by register 2 into the location pointed to by register 3. Register 2 is incremented (by 2 for a word, since the PDP-11 is a byte-addressable computer) after it is used as a pointer; register 3 is decremented (by 2) before it is used as pointer. Now consider what will happen if we get a fault when trying to store into the location pointed to by register 3. To restart the instruction, we must reset the two registers to the values they had before we started the execution of the instruction.

One solution is to create a new special status register to record the register number and amount modified for any register allows the operating system to "undo" the effects of a partially executed instruction that causes a page fault.

These are by no means the only architectural problems resulting from adding paging to an existing architecture to allow demand paging, but they illustrate some of the difficulties. Paging is added between the CPU and the memory in a computer system. It should be entirely transparent to the user process. Thus, people often assume that paging could be added to any system. Although this assumption is true for a non-demand paging environment, where a page fault represents a fatal error, it is not true in the case, where a page fault means only that an additional page must be brought into memory and the process restarted.

Performance of Demand Paging

Demand paging can have a significant effect on the performance of a computer system. To see why, let us compute the *effective access* time for a demand-paged memory. The memory access time, ma, for most computer systems now ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If however, a page fault occurs, we must first read the relevant page from disk, and then access the desired word.

Let p be the probability of a page fault ($0 <= p <= 1$)). We would expect p to be close to zero; that is, there will be only a few page faults. The effective access time is then

$$\text{effective-access time} = (1 - p) * \text{ma} + p * \text{page-fault-time}.$$

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. Cause trap to the operating system.
2. Save the user registers and process state on the memory stack of the process.

3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine location of the page on the disk.
5. Issue a read from the disk to a free frame:
   (a) Wait in a queue for 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. Interrupt from the disk (I/O completed).
8. Save the registers and process state for the other user (if step 6 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 may be 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. This arrangement allows multi-programming 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 may be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each. The page-switch time, on the other hand, will probably be close to 24 milliseconds. A typical hard disk has an average latency of 8 milliseconds, a seek-time of 15 milliseconds, and a transfer time of 1 millisecond. Thus, the total paging time would be close to 25 milliseconds, including hardware and software time. Remember also that we are looking at only the device service time. If a queue of processes is waiting for the device (other processes that have caused page faults), we have to add device queuing-time as we wait for the paging device to be free to service our request, increasing the time to swap even more than the one calculated.

If we take an average page-fault service time of 25 milliseconds and a memory access time of 100 nanoseconds, then the effective access time in nanoseconds is

$$\begin{aligned} \text{Effective-access-time} &= (1 - p) * (100) + p * 25 \text{ milliseconds} \\ &= (1 - p) * 100 + p * 25{,}000{,}000 \\ &= 100 - 100p + 25000000p \\ &= 100 + 24999900p \end{aligned}$$

We see then that the effective access time is directly proportional to the page-fault rate (or probability). If one access out of 1000 causes a page fault, the effective access time is 25 microseconds. The computer would be slowed down by a factor of 250 because of demand paging. If we want less than 10 per cent degradation, we need

$$110 > 100 + 25{,}000{,}000 * p$$

$$10 > 25{,}000{,}000 * p$$

$$p < 0.0000004$$

That is, to keep the slowdown due to paging to a reasonable level, we can allow only less than 1 memory access out of 2,500,000 to page fault. That is, to keep the slowdown due to paging to a reasonable level, we can allow only less than 1 memory access out of 2,500,000 to page fault.

As is obvious, it is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically.

One additional aspect of demand paging is the handling and overall use of swap space. Disk I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used. It is, therefore, possible for the system to gain better paging throughput, by copying an entire file image into the swap space at process startup, and then to perform demand paging from the swap space. Systems with limited swap space can employ a different scheme when binary files are used. Demand pages for such files are brought directly from the file system. However, when page replacement is called and read in from the file system again if needed. Yet another option is initially to demand pages from the file system, but to write the pages to swap space as they are replaced. This approach will ensure that only needed pages are ever read from the file system, but all subsequent paging is done from swap space. This method appears to be a good compromise. This scheme is used in BSD UNIX.

3.5.2 Page Replacement

In our presentation so far, the page-fault rate is not a serious problem, because each page is faulted for at most once, when it is first referenced. This representation is not strictly accurate. Consider that, if a process of 10 pages actually uses only one-half of them, and then demand paging saves the I/O necessary to load the five pages that are never used. We could also increase our degree of multi-programming by running twice as many processes. Thus, if we had 40 frames, we could run eight processes, rather than the four that could run if each required 10 frames (five of which were never used).

If we increase our degree of multi-programming, we are over-allocating memory. If we run six processes, each of which is 10 pages in size, but actually uses only five pages, we have higher CPU utilization and throughput, with 10 frames to spare. It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all 10 of its pages, resulting in a need for 60 frames, when only 40 are available. Although this situation may be unlikely, it becomes much more likely as we increase the multiprogramming level, so that the average memory usage is close to the available physical memory. (In our example, why stop at a multi-programming level of six, when we can move to a level of seven or eight?)