Module 4

Background

Any 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. Dynamic loading can help to ease this restriction, but it generally requires special precautions and extra work by the programmer. The requirement that instructions must be in physical memory to be executed seems both necessary and reasonable; but it is also unfortunate, since it limits the size of a program to the size of physical memory.

The ability to execute a program that is only partially in memory would confer 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.
- 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 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.

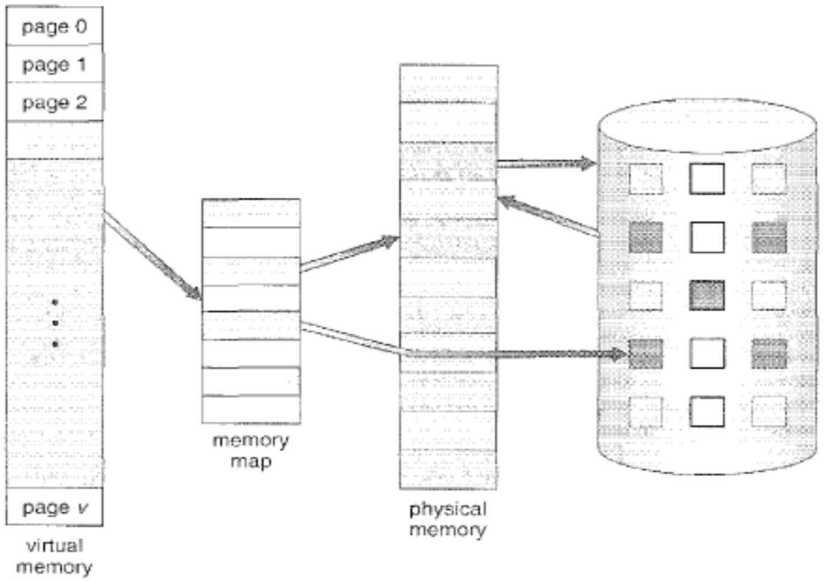

Figure 4.1 Diagram showing virtual memory that is larger than physical memory.

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 (Figure 4.1). Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available; she can concentrate instead on the problem to be programmed.

The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address-say, address O- and exists in contiguous memory, as shown in Figure 4.2.

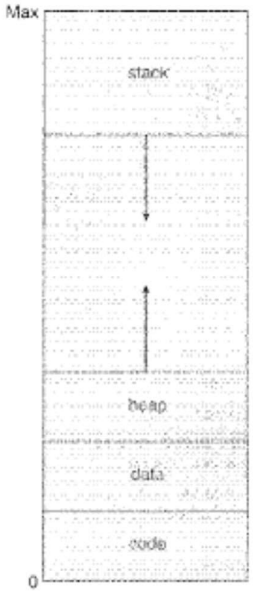

Figure 4.2 Virtual address space.

Note in Figure 4.2 that we allow for the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls. The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as **sparse address spaces**.

Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

In addition to separating logical memory from physical memory, virtual memory also allows files and

memory to be shared by two or more processes through page sharing:

- System libraries can be shared by several processes through mapping of the shared object into a virtual address space.
- Virtual memory enables processes to share memory (as shown in figure 4.3).
- Virtual memory can allow pages to be shared during process creation with the fork () system call, thus speeding up process creation.

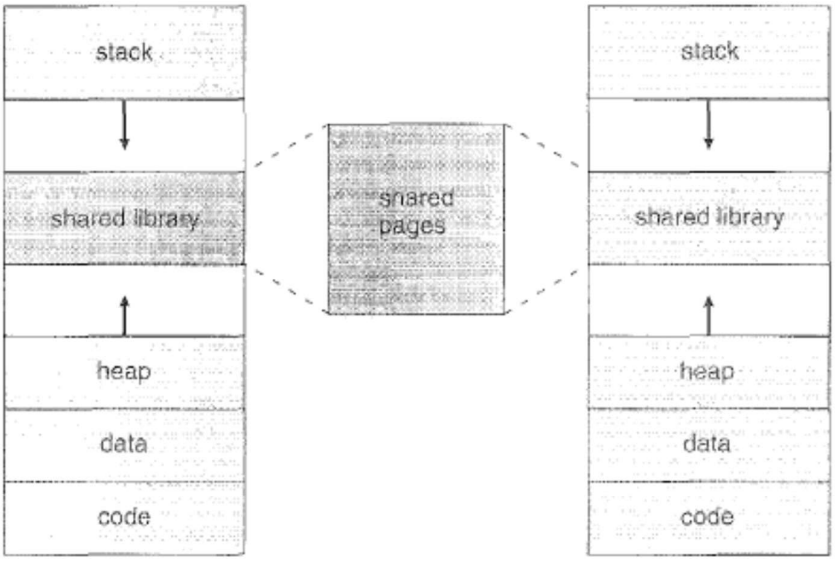

Figure 4.3 Shared library using virtual memory.

Demand Paging

Consider how an executable program might be loaded from disk into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that 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 initially 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 (Figure 4.4) 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.

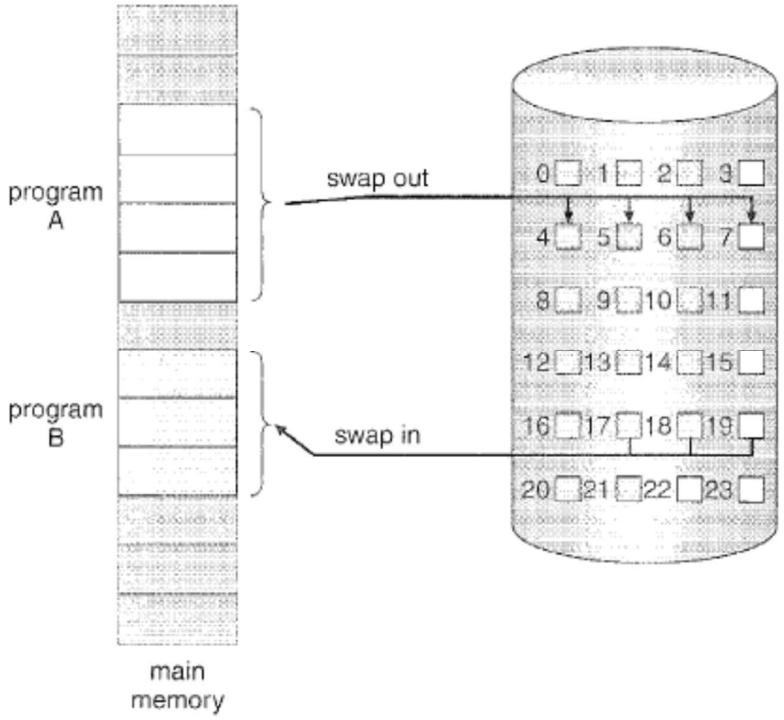

Figure 4.4 Transfer of a paged memory to contiguous disk space.

Basics concepts

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those 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 the pages that are in memory and the pages that are on the disk.

When this (valid and invalid) 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 4.5.

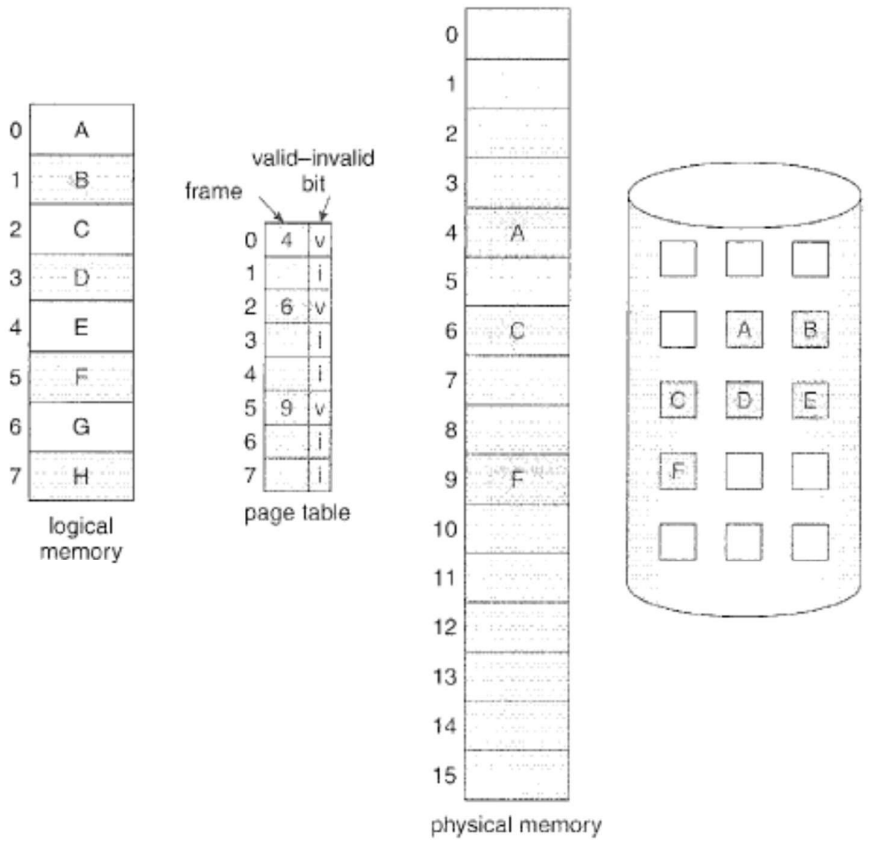

Figure 4.5 Page table when some pages are not in main memory.

But what happens if the process tries to access 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.

The procedure for handling this page fault is straightforward as listed below:

- We check an internal table (usually kept with the process control block) for this process to

determine whether the reference was a valid or an invalid memory access.

- If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
- 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 trap. The process can now access the page as though it had always been in memory.

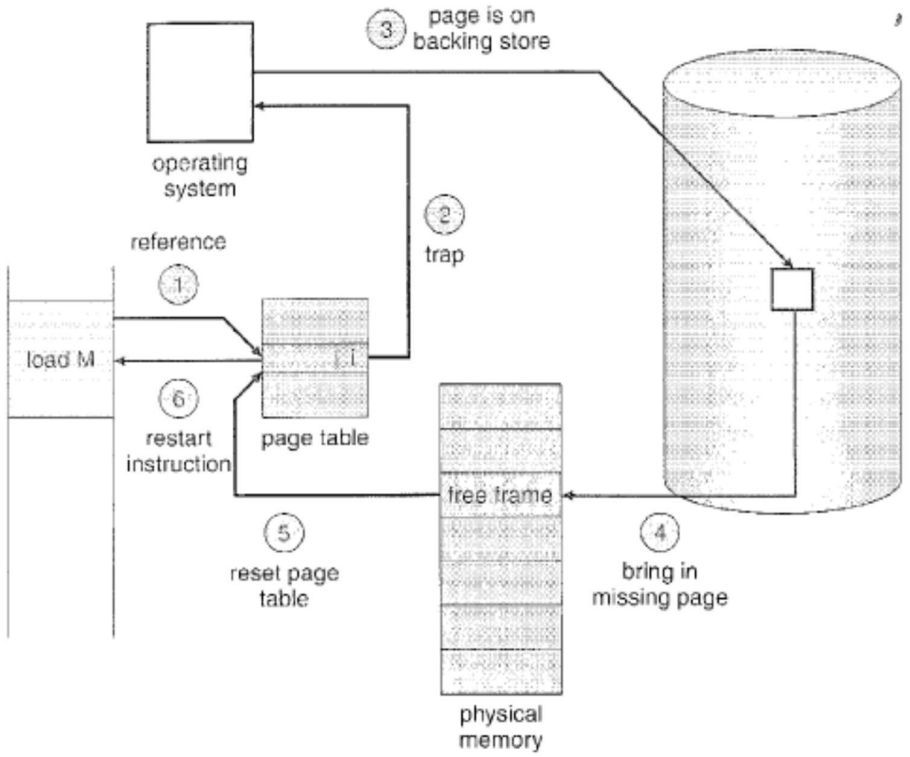

Figure 4.6 Steps in handling a page fault.

Figure 4.6 explains all above listed steps that shows exactly how a page fault is handled.

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.**

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

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-paged memory. For most computer systems, the memory-access time, denoted Ina, 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, we would expect to have only a few page faults. The **effective access time** is then

Effective access time = (1 - p) x ma + p x 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 [f0 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.

Copy-on-write

Process creation using the fork ( ) system call may initially bypass the need for demand paging by page sharing. This provides for rapid process creation and minimizes the number of new pages that must be allocated to the newly created process.

Fork ( ) system call creates a child process that is a duplicate of its parent. Fork ( ) worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. If child processes invoke the exec ( ) system call immediately after creation, the copying of the parent's address space may be unnecessary. Instead, we can use a technique called **copy-on-write** which works by allowing the parent and child processes initially to share the same pages.

These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created. Copy-on-write is illustrated in Figures 4.7 and Figure 4.8, which show the contents of the physical memory before and after process 1 modifies page C. For example, assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-an-write. The operating system" will then create a copy of this page, mapping it to the address space of the child process. The child process will then modify its copied page and not the page belonging to the parent process.

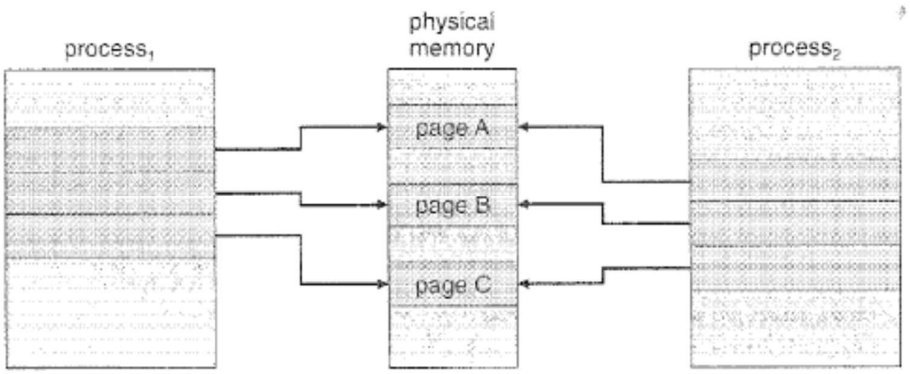

Figure 4.7 Before process 1 modifies page C.

The copy-on-write technique is used, only the pages that are modified by either process are copied; all unmodified pages can be shared by the parent and child processes.

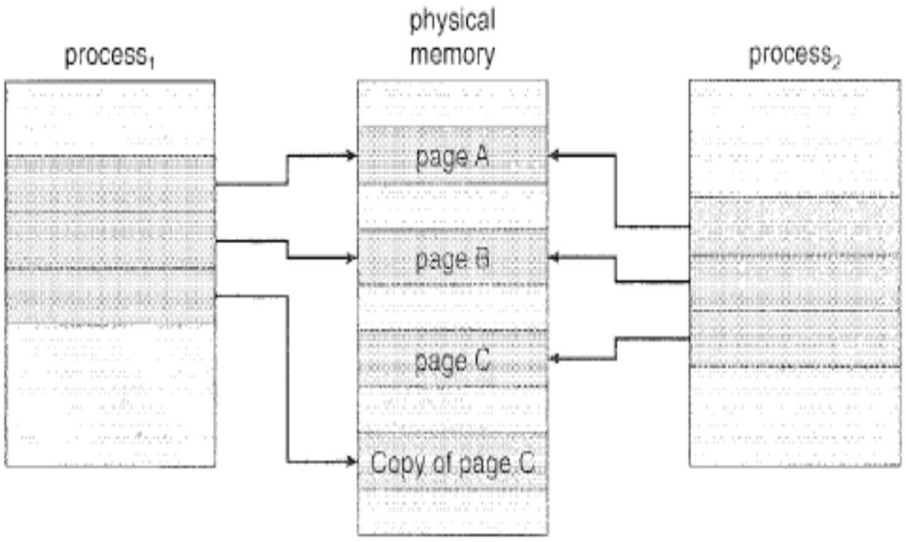

Figure 4.8 After process 1 modifies page C.

When it is determined that a page is going to be duplicated using copy-on-write, it is important to note the location from which the free page will be allocated. Many operating systems provide a pool of free pages for such requests. These free pages are typically allocated when the stack or heap for a process must expand or when there are copy-on-write pages to be managed. Operating systems typically allocate these pages using a technique known as zero-fill-on-demand.

Page replacement

When a user process is executing, a page fault occurs. The operating system determines where the desired page is residing on the disk but then finds that there are no free frames on the free-frame list; all memory is in use as shown in Figure 4.9.

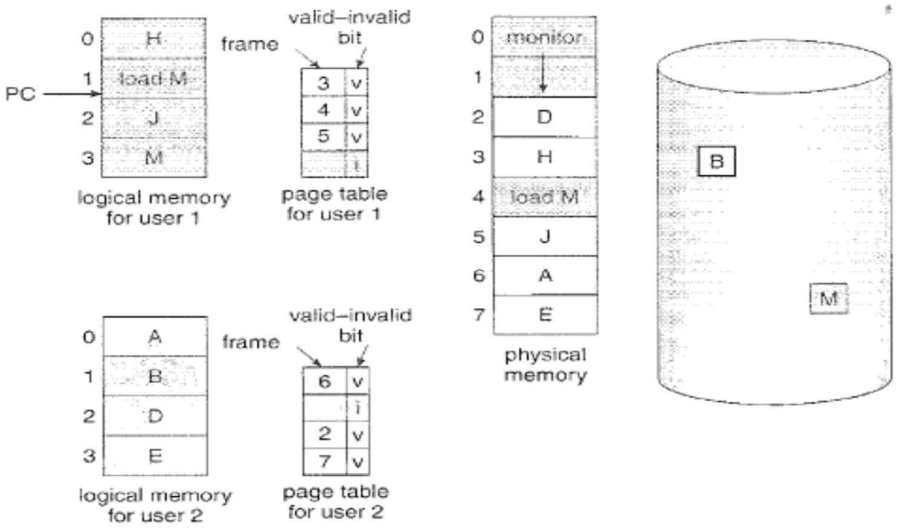

Figure 4.9 Need for page replacement.

Demand paging is the operating system's attempt to improve the computer system utilization and throughput. Users should not be aware that their processes are running on a paged system-paging should be logically transparent to the user. The operating system could instead swap out a process, freeing all its frames and reducing the level of multiprogramming. So Page Replacement is the good solution.

Basic Page Replacement

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by ~writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 4.10). We can now use the freed frame to hold the page for which the process fau1ted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.

2. Find a free frame:

    a. If there is a free fran1e, use it.

b. If there is no free frame, use a page-replacement algorithm to select a victim frame.

c. Write the victim frame to the disk; change the page and frame tables accordingly.

3. Read the desired page into the newly freed frame; change the page and frame tables.

4. Restart the user process.

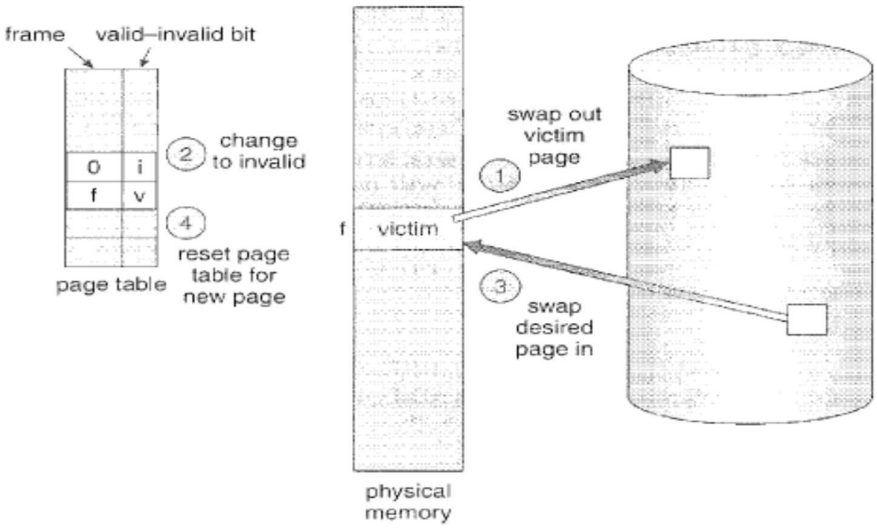

Figure 4.10 Page Replacement.

If no frames are free, **two-page transfers** (one out and one in) are required. This situation effectively **doubles the page-fault service time** and increases the effective access time accordingly.

We can reduce this overhead by using a **Modify bit** or **Dirty bit.** Here each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine it's modified bit. If the bit is set, we know that the page has been **modified** since it was read in from the disk.

In this case, we must write the page to the disk. If the modify bit is not set, however, the page has **not modified** since it was read into memory. In this case, we need not write the memory page to the disk, as it is already in disk. This reduces the time required to service a page fault, since it reduces I/O time by one-half if the page has not been modified.

To implement Page replacement, we have to develop Frame allocation algorithm and Page replacement algorithm. If we have multiple processes in memory, we must decide how many frames

to allocate to each process; and when page replacement is required, we must select the frames that are to be replaced.

We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a reference string. We can generate reference strings artificially (by using a random-number generator, for example), or we can trace a given system and record the address of each memory reference. The latter choice produces a large number of data (on the order of 1 million addresses per second).

To reduce the number of data we use two facts: first for a given page size and second if we have a reference to a page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the first reference, so the immediately following references will not fault.

FIFO page replacement

The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

We next illustrate several page-replacement algorithms. In doing so, we use the reference string

7,0, 1,2,0,3,0,4,2,3,0,3,2, 1,2,0, 1, 7, 0, 1

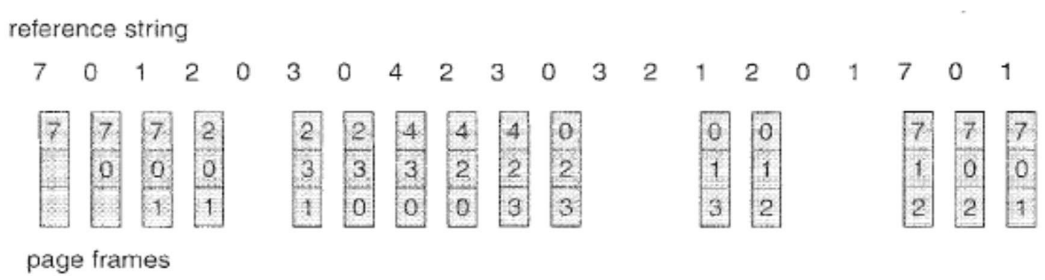

Figure 4.11 FIFO page-replacement algorithm

For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory,

we have no fault for this reference.

The first reference to 3 results in replacement of page 0, since it is now first in line. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 4.11. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether. Notice that the number of faults for four frames will be greater than the number of faults for three frames. This most unexpected result is known as **Belady's anomaly**.

Optimal Page Replacement

One result of the discovery of Belady's anomaly was the search for an optimal page-replacement algorithm. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN. It is simply this.

Again we can use the same reference string:

7,0, 1,2,0,3,0,4,2,3,0,3,2, 1,2,0, 1, 7, 0, 1

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 4.12.

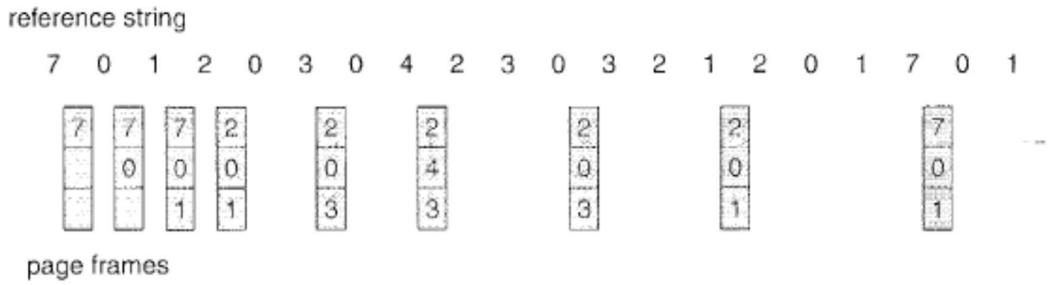

Figure 4.12 Optimal page-replacement algorithm.

With only nine-page faults, optimal replacement is much better than a FIFO algorithm, which resulted in fifteen faults. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

LRU Page Replacement

If the optimal algorithm is not feasible, perhaps an approximation of the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward

versus forward in time) is that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be used. If we use the recent past as an approximation of the near future, then we can replace the page that has not been used for the longest period of time (Figure 4.13). This approach is the least-recently-used (LRU) algorithm.

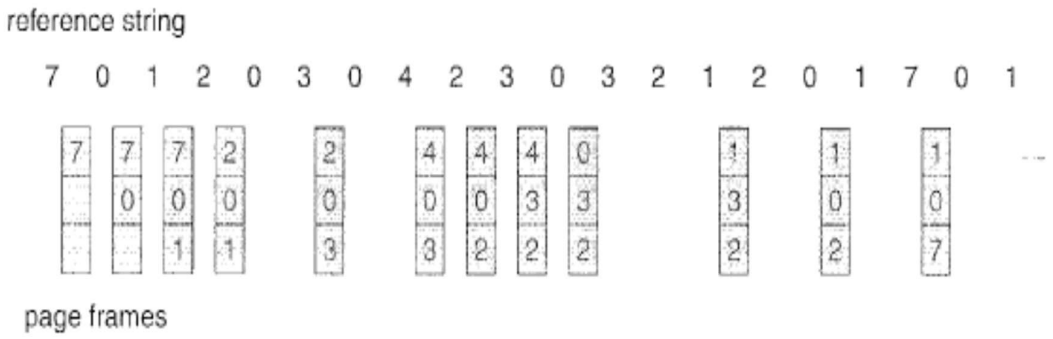

Figure 4.13 LRU page-replacement algorithm

The result of applying LRU replacement to our example reference string is shown in Figure 4.13. The LRU algorithm produces 12 faults. Notice that the first 5 faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU rep1itcement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with 12 faults is much better than FIFO replacement with 15.

Like optimal replacement, LRU replacement does not suffer from Belady's anomaly. Both belong to a class of page-replacement algorithms, called **stack** algorithms that can never exhibit Belady's anomaly. A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n + 1 frames.

LRU-Approximation Page Replacement

Few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support, and other page replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set by the hardware whenever that page is referenced (either a read

or a write to any byte in the page). Reference bits are associated with each entry in the page table.

Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.

Additional-Reference-Bits Algorithm

We can gain additional ordering information by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory. At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.

These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, for example, then the page has not been used for eight time periods; a page that is used at least once in each period has a shift register value of 11111111. A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.

The number of bits of history can be varied, of course, and is selected (depending on the hardware available) to make the updating as fast as possible. In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the **second-chance page replacement** algorithm.

Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page (as shown in figure 4.14).

When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

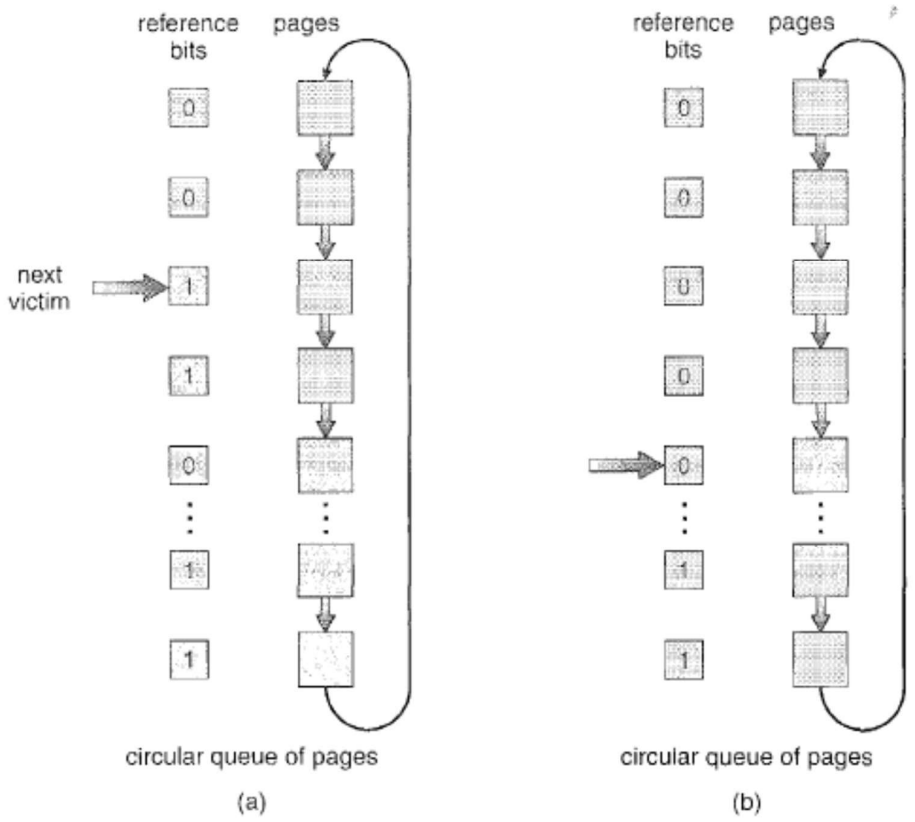

Figure 4.14 Second-chance (clock) page-replacement algorithm.

Enhanced Second-Chance Algorithm

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

1. (0,0) neither recently used nor modified-best page to replace.

2. (0, 1) not recently used but modified-not quite as good, because the page will need to be written out before replacement.

3. (1, 0) recently used but clean-probably will be used again soon.

4. (1, 1) recently used and modified-probably will be used again soon, and the page will be need to be written out to disk before it can be replaced.

Counting-Based Page Replacement

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

- The **least frequently used (LFU)** page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.
- The **most frequently used (MFU)** page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Page-Buffering Algorithms

Other procedures are often used in addition to a specific page-replacement algorithm. For example, systems commonly keep a pool of free frames. When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out.

Applications and Page Replacement

In certain cases, applications accessing data through the operating system's virtual memory perform worse than if the operating system provided no buffering at all. A typical example is a database, which provides its own memory management and I/O buffering. Applications like this understand their memory use and disk use better than does an operating system that is implementing algorithms for general-purpose use. If the operating system is buffering I/O, and the application is doing so as well, then twice the memory is being used for a set of I/O.

Allocation of Frames

We turn next to the issue of allocation. How do we allocate the fixed amount of free memory among the various processes? If we have 93 free frames and two processes, how many frames

does each process get?

The simplest case is the single-user system. Consider a single-user system with 128 K8 of memory composed of pages 1 K8 in size. This system has 128 frames. The operating system may take 35 K8, leaving 93 frames for the user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of

page faults.

The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

Minimum Number of Frames

We must also allocate at least a minimum number of frames. Here, we look more closely at the latter requirement. One reason for allocating at least a minimum number of frames involves performance. Obviously, as the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution. In addition, remember that, when a page fault occurs before an executing instruction is complete, the instruction must be restarted. Consequently, we must have enough frames to hold all the different pages that any single instruction can reference.

Allocation Algorithms

The easiest way to split the frames among n processes is to give everyone an equal share, m/ n frames. For instance, if there are 93 frames and five processes, each process will get 18 frames. The leftover three frames can be used as a free-frame buffer pool. This scheme is called equal allocation.

An alternative is to recognize that various processes will need differing amounts of memory. Consider a system with a 1-KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted.

Global versus Local Allocation

Another important factor in the way frames are allocated to the various processes is page replacement. With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: global replacement and local replacement. Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another Local replacen1ent requires that each process select from only its own set of allocated frames.

For example, consider an allocation scheme where we allow high-priority processes to select frames from low-priority processes for replacement. A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation at the expense of a low-priority process.

Thrashing

If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process's execution. We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of intermediate CPU scheduling.

In fact, look at any process that does not have "enough" frames. If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

This high paging activity is called **thrashing**. A process is thrashing if it is spending more time paging than executing.

Cause of Thrashing

The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

We can limit the effects of thrashing by using a local replacement algorithm (or priority replacement algorithm). With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well. However, the problem is not entirely solved. If processes are thrashing, they will be in the queue for the paging device most of the time.

To prevent thrashing, we must provide a process with as many frames as it needs. But how do we

know how many frames it "needs"? There are several techniques. The **working-set** strategy starts by looking at how many frames a process is actually using. This approach defines the locality model of process execution.

For example, when a function is called, it defines a new locality. In this locality, memory references are made to the instructions of the function call, its local variables, and a subset of the global variables. When we exit the function, the process leaves this locality, since the local variables and instructions of the function are no longer in active use. We may return to this locality later.

Working set model

As mentioned, the working-set model is based on the assumption of locality. This model uses a parameter, Δ to define the working-set window. The idea is to examine the most recent Δ page references. The set of pages in the most recent Do page references is the working set (Figure 4.15). If a page is in active use, it will be used in the working set. If it is no longer being used, it will be dropped from the working set Δ. time units after its last reference. Thus, the working set is an approximation of the program's locality.

For example, given the sequence of memory references shown in Figure 4.15, if Δ. = 10 memory references, then the working set at time t_1 is {1, 2, 5, 6, and 7}. By time t_2, the working set has changed to {3, 4}.

The accuracy of the working set depends on the selection of Δ. If Δ is too small, it will not encompass the entire locality; if Δ is too large, it may overlap several localities. In the extreme, if Δ. is infinite, the working set is the set of pages touched during the process execution.

This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization. The difficulty with the working-set model is keeping track of the working set. The working-set window is a movi.ng window. At each memory reference, a new reference appears at one end and the oldest reference drops off the other end. A page is in the working set if it is referenced anywhere in the working-set window.

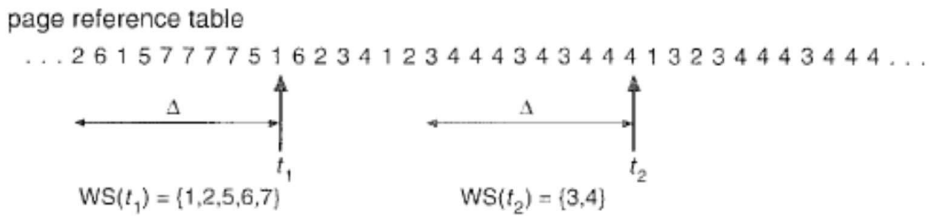

Figure 4.15 Working set model.

Page Fault frequency

The working-set model is successful, and knowledge of the working set can be useful for preparing, but it seems a clumsy way to control thrashing. A strategy that uses the page-fault frequency (PFF) takes a more direct approach.

The specific problem is how to prevent thrashing. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds on the desired page-fault rate (Figure 4.16).

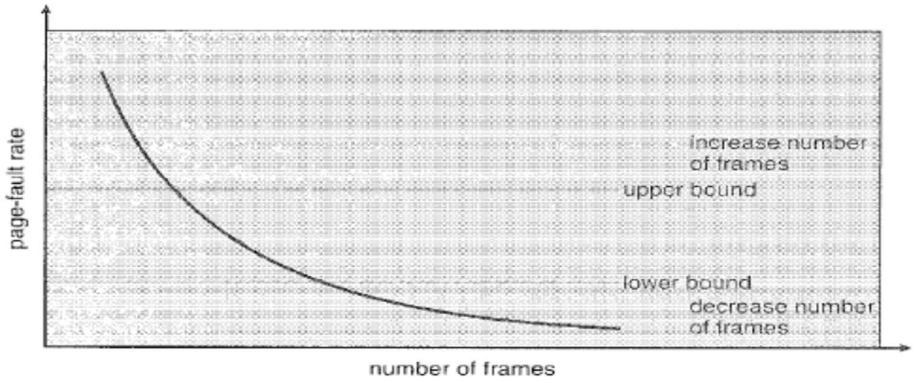

Figure 4.16 Working set model

File System

File Concept

Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**. Files are mapped by the operating system onto physical

Module 4

Memory management strategies

Background

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 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.

Basic Hardware

Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. 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. Completing a memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to **stall,** since it does not have the data required to complete the instruction that it is executing.

The remedy is to add fast memory between the CPU and main memory. A memory buffer used to accommodate a speed differential, called Cache..

We first need to make sure that each process has a separate memory space. For 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. This protection is provided by using two registers, usually a **base** and a **limit** as illustrated in Figure 3.13. The **base** register holds the smallest legal physical memory address; the **limit** register specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

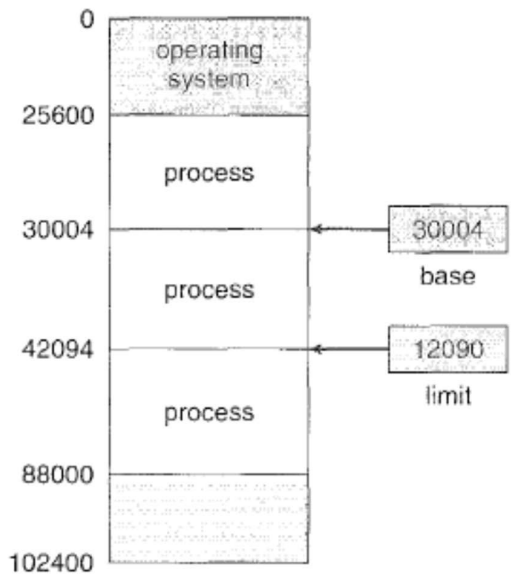

Figure 3.13 A base and a limit register define a logical address space.

Protection of memory space is accomplished by having the CPU hardware compare every 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 as shown in figure 3.14. This scheme prevents a user program from 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. This allows the operating system to change the value of the registers but prevents user programs from changing the contents. The operating system, executing in kernel mode, is given unrestricted access to both operating system memory and users' memory.

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

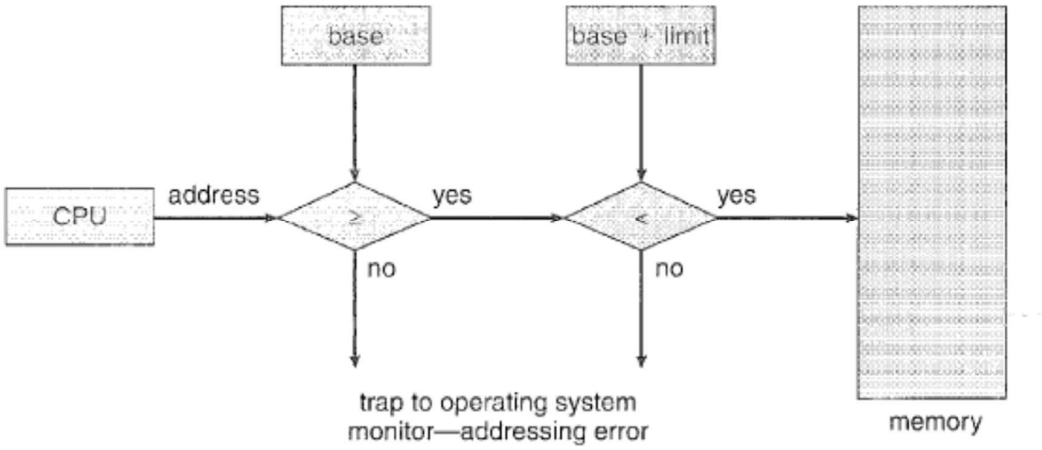

Figure 3.14 Hardware address protection with base and limit registers.

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.

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

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.
- **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.

How the user program is processed, based on above times, is as shown in figure 3.15.

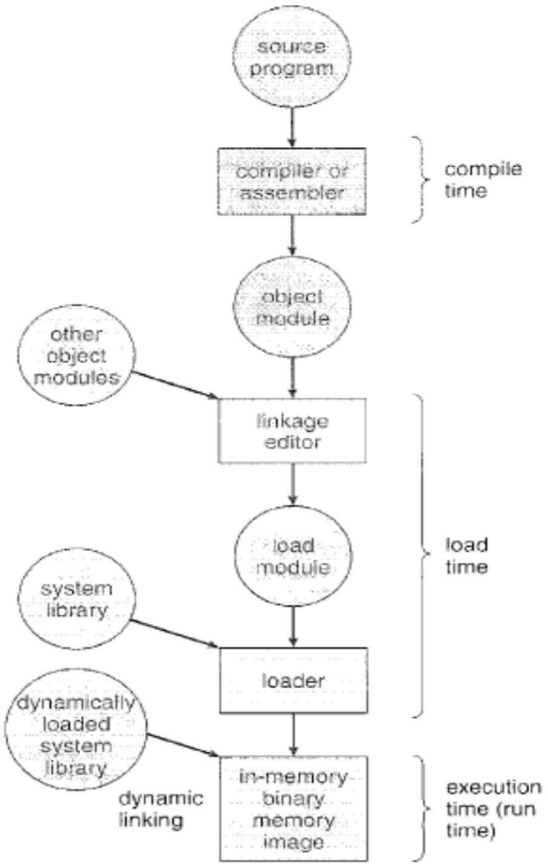

Figure 3.15 Multistep processing of a user program.

Logical versus Physical Address Space:

An address generated by the CPU is commonly referred as **logical address**. Whereas an address seen by the memory unit-that is, the one loaded into 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 address. In execution time binding, we refer to the logical address as virtual address. The set of all logical addresses generated by a program is a logical address space; the set of all physical addresses corresponding to these logical addresses is a physical address space.

The run-time mapping from virtual to physical addresses is done by a hardware 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 3.16). 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.

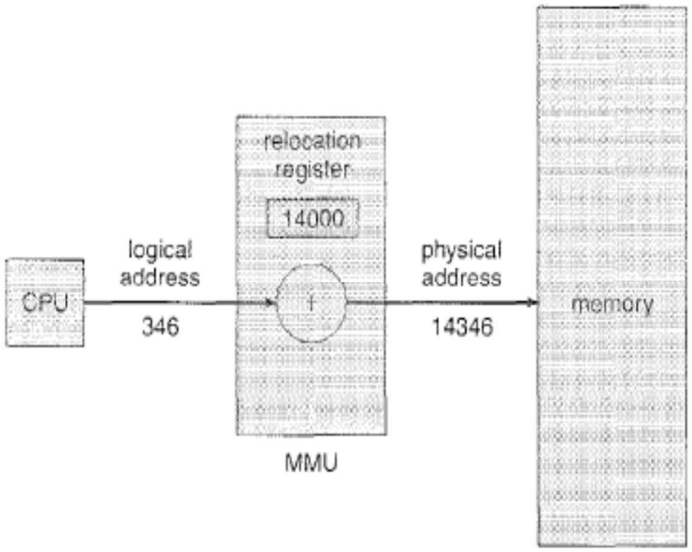

Figure 3.16 Dynamic relocation using a relocation register.

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.

Dynamic Loading

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. 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.

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

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 (refer figure 3.15).

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 aut01natically 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 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.

Swapping

A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution. 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 3.17).

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.

This approach can be used for priority based scenario also. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called **roll out, roll in.**

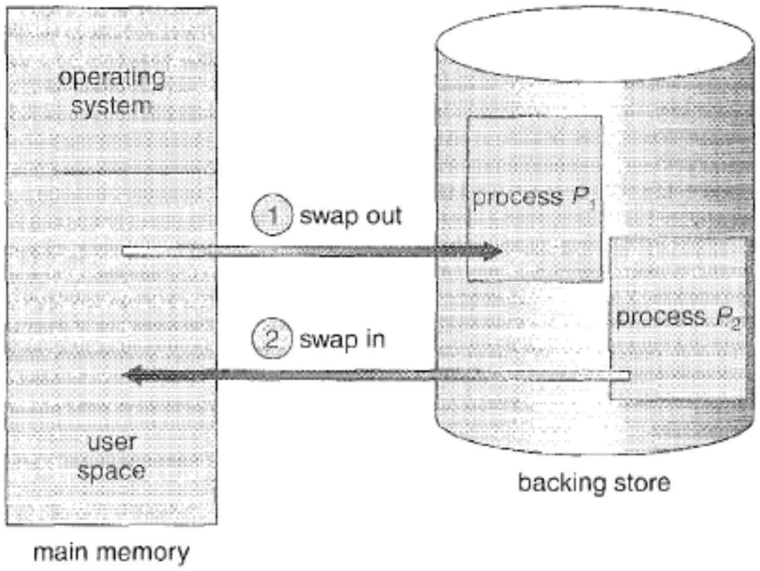

Figure 3.17 Swapping of two processes using a disk as a backing store.

Swapping requires a backing store. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. The system maintains a ready queue consisting of all

processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.

Notice that the major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped. Swapping is constrained by other factors 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/O. A process may be waiting for an I/O operation when we want to swap that process to free up memory.

Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate the parts of the 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 irtterrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.

Memory Mapping and Protection

We can provide these features by using a relocation register, 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 example, relocation = 100040 and limit = 74600). 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 3.18).

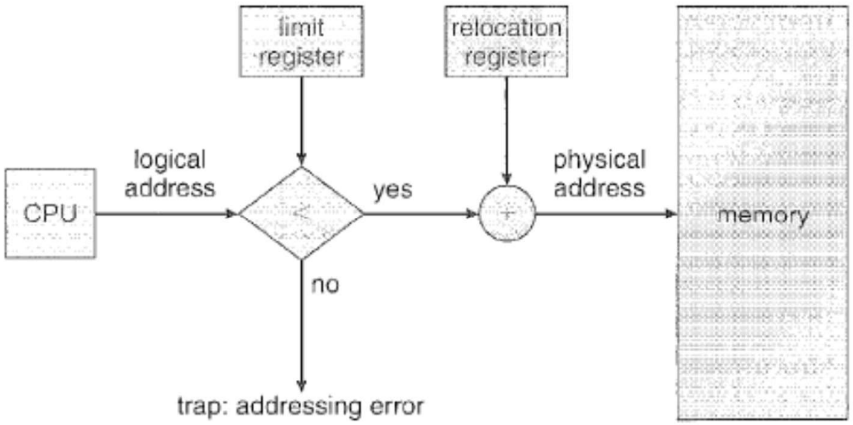

Figure 3.18 Hardware support for relocation and limit registers.

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by the CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

Memory Allocation

One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple 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 **fixed-partition scheme,** the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a hole. When a process arrives and needs memory, we search for a hole large enough for this process. If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.

There are three main strategies to select a hole from the set of available holes:

- **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can

stop searching as soon as we find a free hole that is large enough.

- **Best fit**. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is 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.

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.

Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation** -memory that is internal to a partition but is not being used.

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

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 the latter is available. Two complementary techniques achieve this solution: **paging** and **segmentation**.

Paging

Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous. Paging avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store; most memory-management schemes used before the introduction of paging suffered from this problem.

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 the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

The hardware support for paging is illustrated in Figure 3.19. Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d).** The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in Figure 3.20.

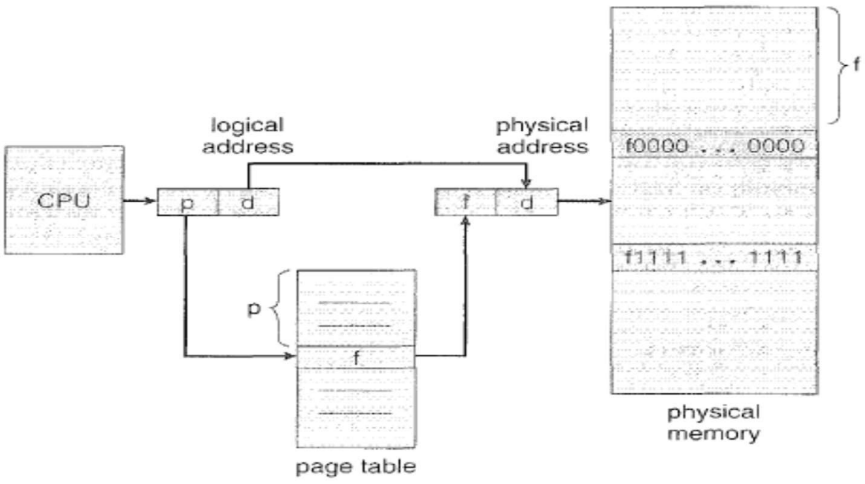

Figure 3.19 Paging hardware

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.

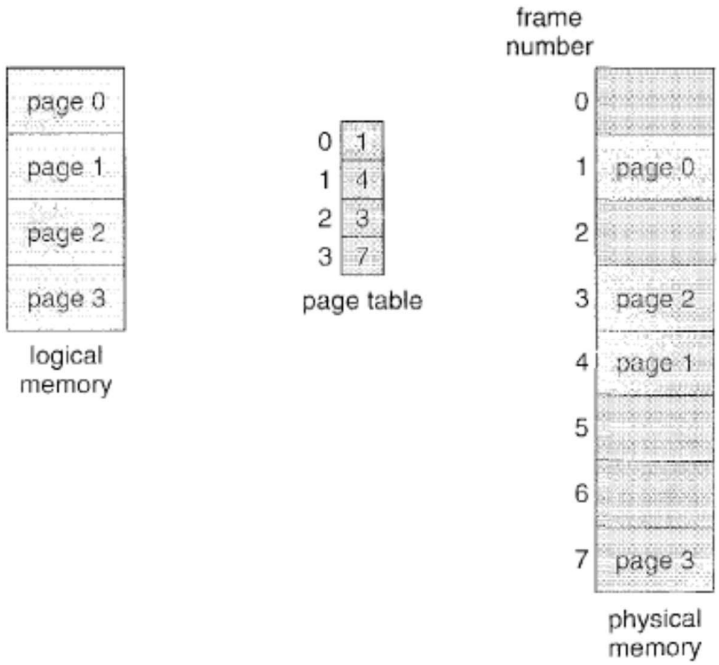

Figure 3.20 Paging model of logical and physical memory.

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

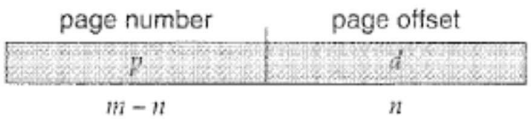

Figure 3.21 page number and page offset

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

As a concrete (although minuscule) example, consider the memory in Figure 3.22. 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).

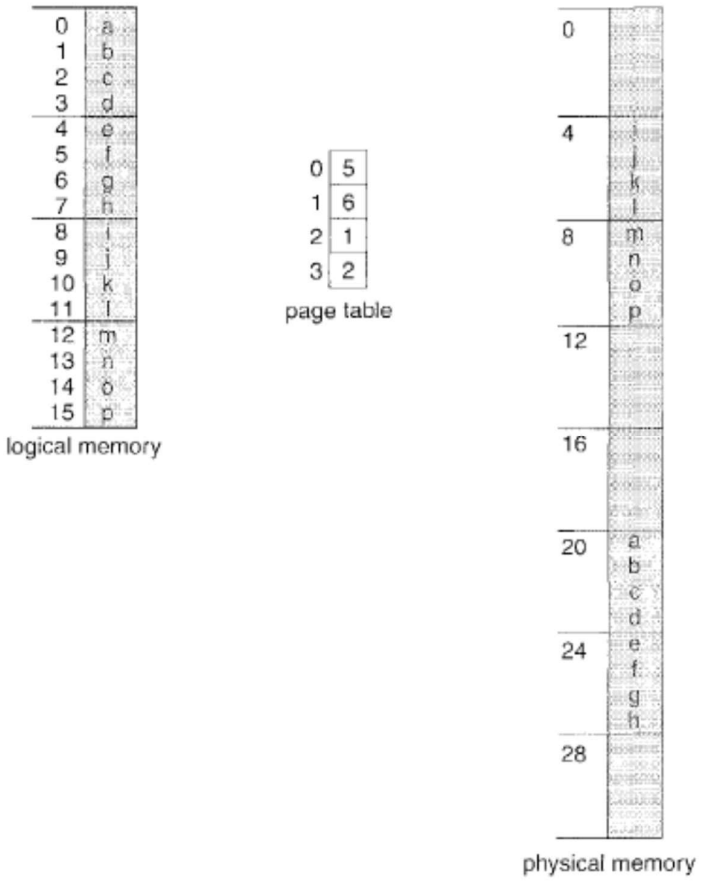

Figure 3.22 Paging example for a 32-byte memory with 4-byte pages.

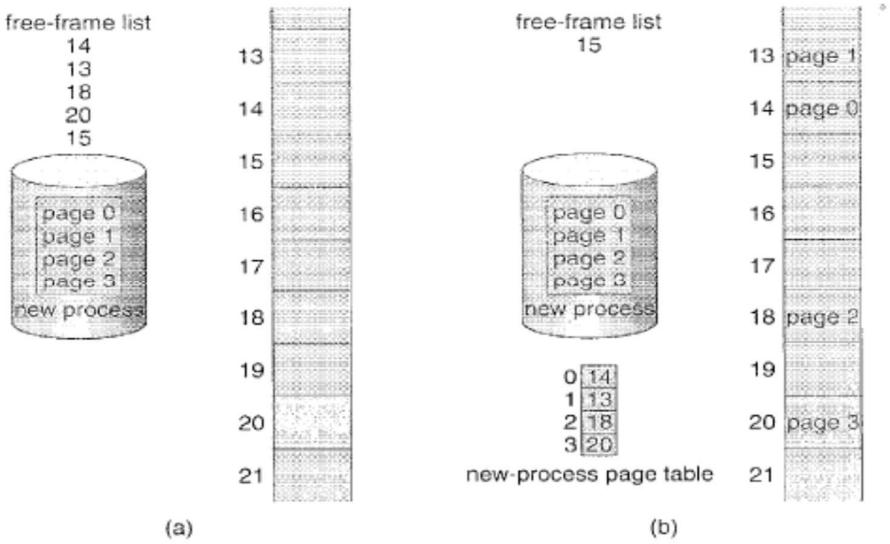

Figure 3.23 Free frames (a) before allocation and (b) after allocation.

Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5 x 4) + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6 x 4) + 0). Logical address 13 maps to physical address 9.

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

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 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 (as shown in figure 3.23).

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. 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. The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated registers.

The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). In this, 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 ch8/8. 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 look-aside buffer (TLB). The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value (as shown in figure 3.24). When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; the hardware, however, is expensive. If a page number is found in TLB, it is called as a **hit**, otherwise it is a **miss**.

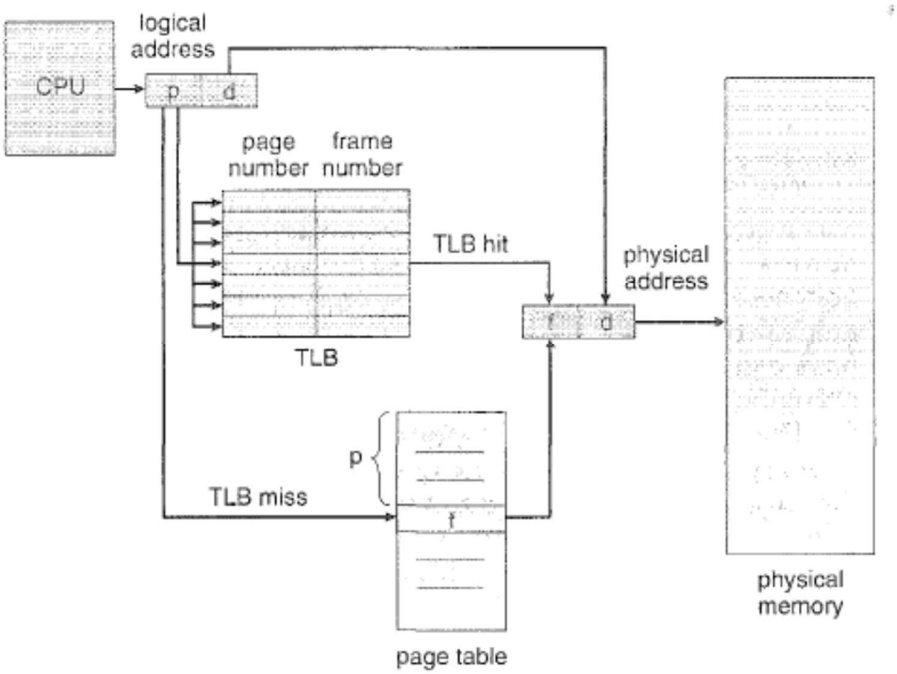

Figure 3.24 Paging hardware with TLB.

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

ratio. An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time.

Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table. One additional bit is generally attached to each entry in the page table: a **valid-invalid bit**. When this bit is set to "valid," the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to "invalid," the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid-invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.

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 3.25. 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).

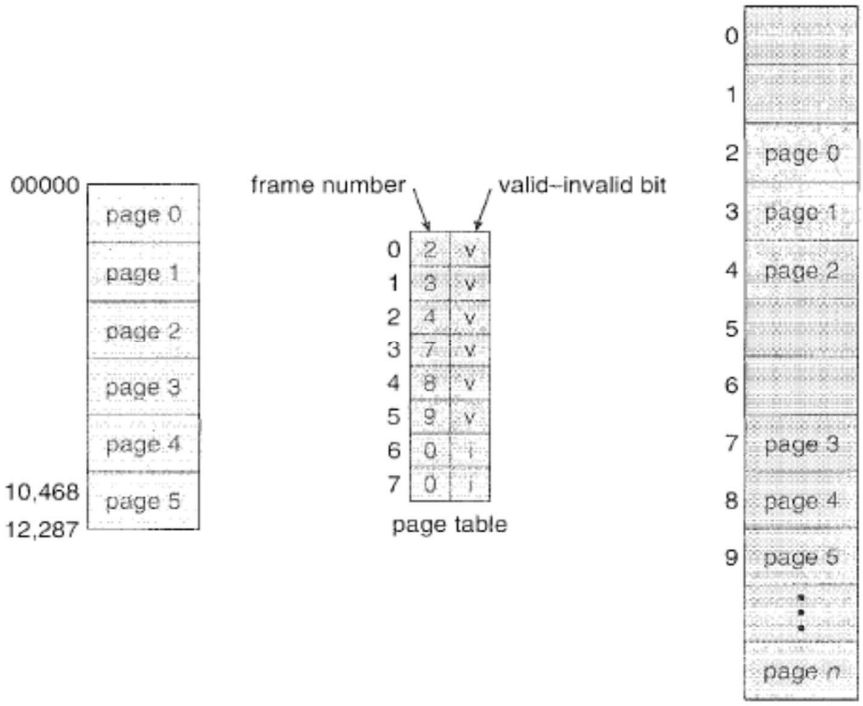

Figure 3.25 Valid (v) or invalid (i) bit in a page table.

Shared Pages

An advantage of paging is the possibility of **sharing common code**. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. If the code is reentrant code (or pure code), however, it can be shared, as shown in Figure 3.26. 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.

Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB-a significant savings. Other heavily used programs can also be shared-compilers, window systems, run-time libraries, database systems, and so on. To be sharable, the code must be reentrant.

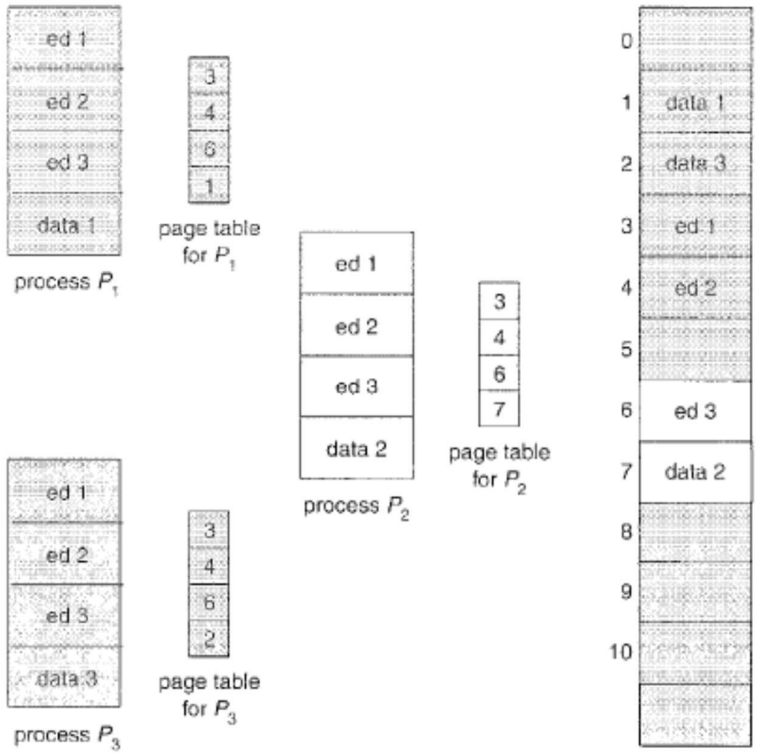

Figure 3.26 Sharing of code in a paging environment.

Organizing memory according to pages provides numerous benefits in addition to allowing several processes to share the same physical pages.

Structure of the Page Table

Hierarchical paging

Most modern computer systems support a large logical address space (2^{32} to 2^{64}). In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (2^{12}), then a page table may consist of up to 1 million entries (2^{32} /2^{12}).

If each entry consists of 4 bytes, each process may need up to 4MB of physical address space for the page table alone. We would not want to allocate the page table contiguously in main memory. The solution to this problem is to divide the page table into smaller pieces. We can do this division in several ways. One way is to use a two-level paging algorithm, in which the page table itself is also paged (Figure 3.27).

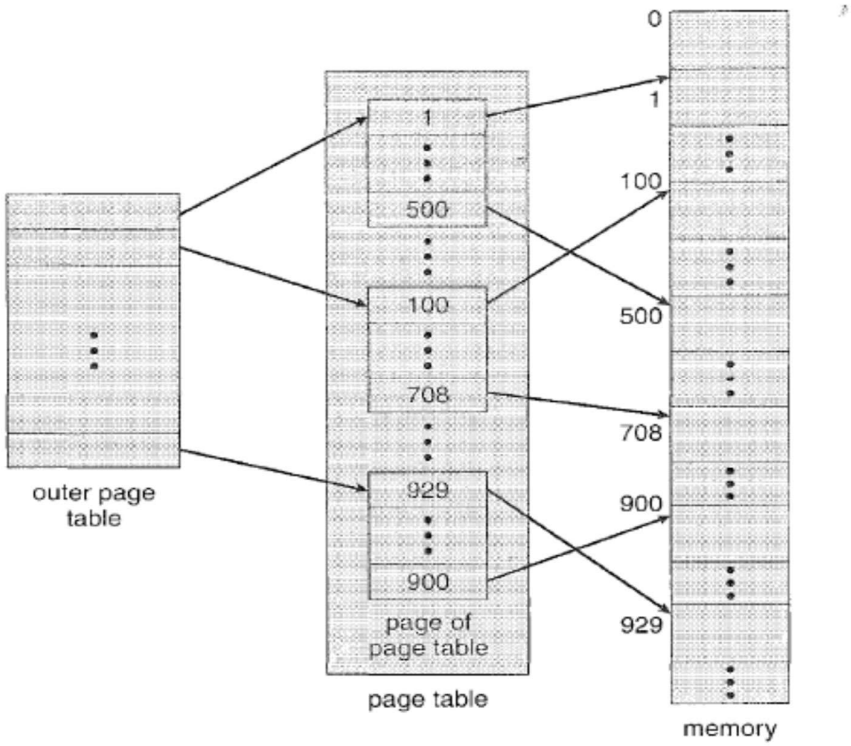

Figure 3.27 A two-level page-table scheme

In system with a 32-bit logical address space and a page size of 4 KB. A logical address is

divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as shown in figure 3.28.

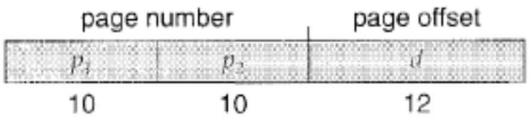

Figure 3.28 An example of logical address space division

Where P1 is an index into the outer page table and P2 is the displacement within the page of the outer page table. The address-translation method for this architecture is shown in figure 3.29. Because address translation works from the outer page table inward, this scheme is also known as a Forward mapped page table.

Suppose a system with 64-bit logical address space, a two-level paging scheme is no longer appropriate. Suppose that the page size in such a system is 4 KB (2^{12}). In this case, the page

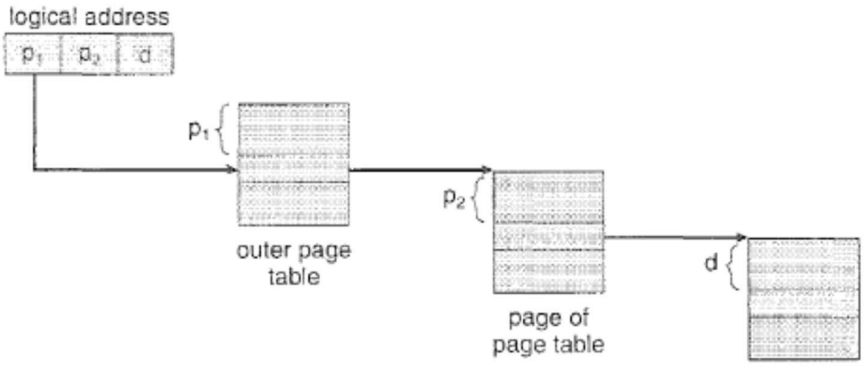

Figure 3.29 Address translation for a two-level 32-bit paging architecture.

table consists of up to 2^{52} entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 2^{10} 4-byte entries. The address looks like as shown in figure 3.30.

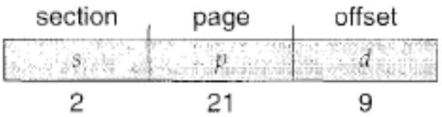

Figure 3.30 An address on VAX architecture.

Where S designates the section number, p is an index into the page table, and d is the displacement within the page. Even when this scheme is used, the size of a one-level page

table for a VAX process using one section is 2^{21} bits * 4 bytes per entry =8 MB. SO that main-memory use is reduced further, the VAX pages the user-process page tables.

If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 2^{10} 4-byte entries. The addresses look like this as shown in the figure 3.31.

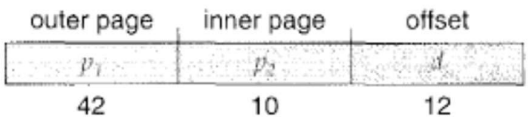

Figure 3.31 Address in two level paging.

The outer page table consists of 2^{42} entries, or 2^{44} bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces. This approach is also used on 32-bit processors for added flexibility and efficiency.

We can divide the outer page table in various ways. We can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (2^{10} entries, or 2^{12} bytes); a 64-bit address space is still daunting as shown in the figure 3.32.

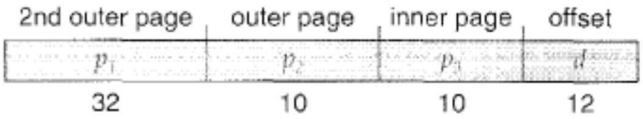

Figure 3.32 Address in three level paging.

The next step would be a four-level paging scheme, where the second level outer page table itself is also paged.

Hashed Page Tables

A common approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the

desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure 3.33.

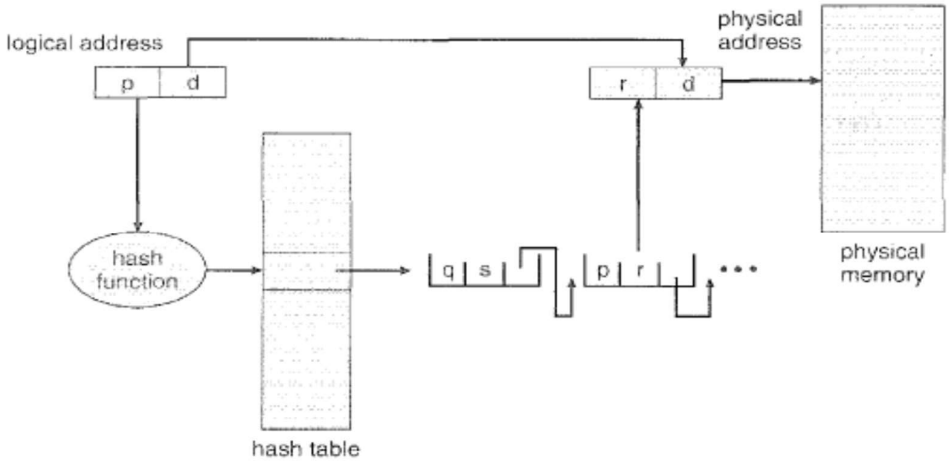

Figure 3.33 Hashed page.

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

Inverted Page Tables

Usually, each process has an associated page table. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless of the latter's validity). This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is and to use that value directly. One of the drawbacks of th.is method is that each page table may consist of millions of entries.

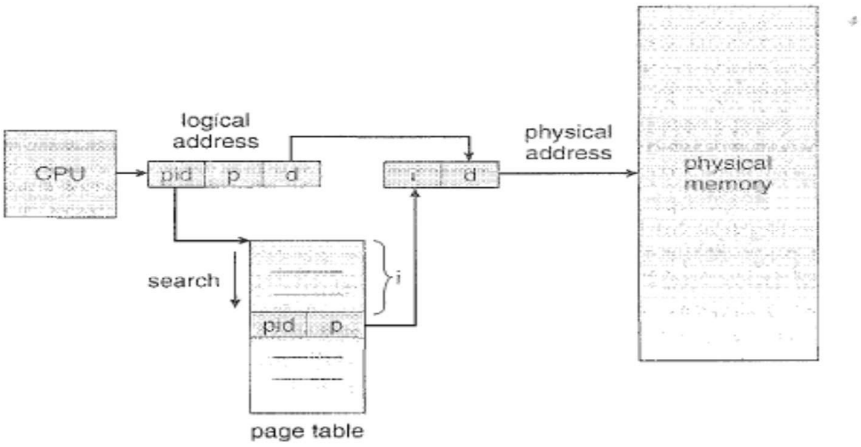

Figure 3.34 Inverted page table.

To solve this problem, we can use an inverted page table. An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. Figure 3.34 shows the operation of an inverted page table.

To illustrate this method, we describe a simplified version of the inverted page table used in the IBM RT. Each virtual address in the system consists of a triple

<process-id, page-number, offset>.

Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier. When a memory reference occurs, part of the virtual address, consisting of <process-id, page number>, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found-say, at entry i-then the physical address <i, offset> is generated. If no match is found, then an illegal address access has been attempted.

Systems that use inverted page tables have difficulty implementing shared memory. Shared memory is usually implemented as multiple virtual addresses (one for each process sharing the memory) that are mapped to one physical address.

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. As we have already seen, the user's view of memory is not the same as the actual physical memory.

Basic Method

Consider how you think of a program when you are writing it. You think of it as a main program with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. You talk about ""the stack," '"the math library," "the main program," without caring what addresses in memory these elements occupy (as shown in figure 3.35).

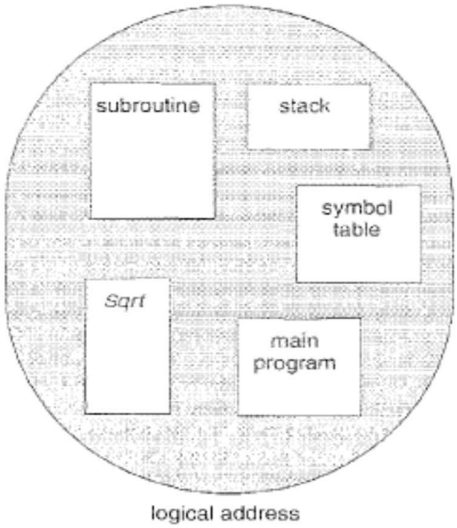

Figure 3.35 User's view of a program.

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 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>.

A C compiler might create separate segments for the following:

1. The code
2. Global variables

3. The heap, from which memory is allocated

4. The stacks used by each thread

5. The standard C library

Hardware

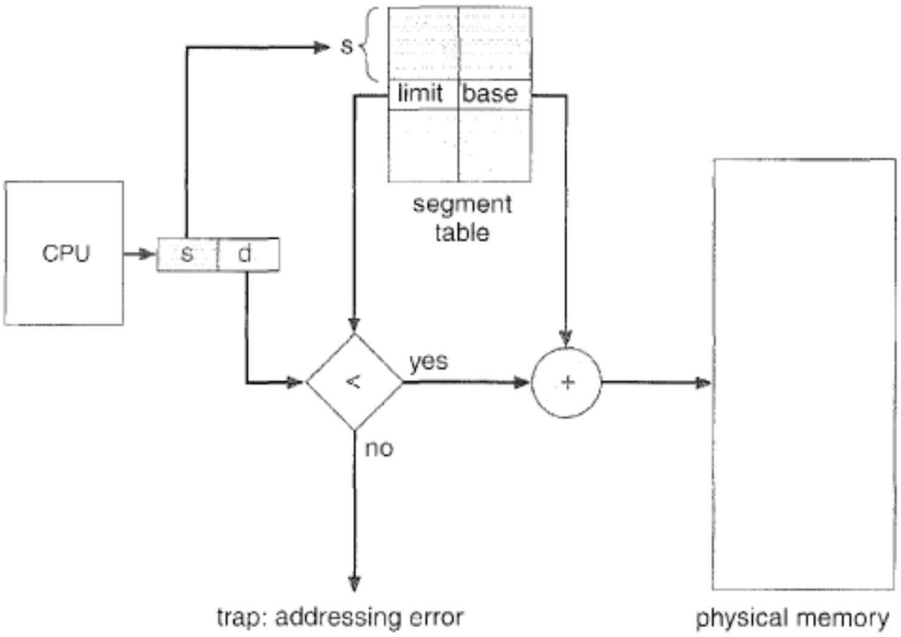

Figure 3.36 Segmentation hardware.

The use of a segment table is illustrated in Figure 3.36. 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.

As an example, consider the situation shown in Figure 3.37. 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.

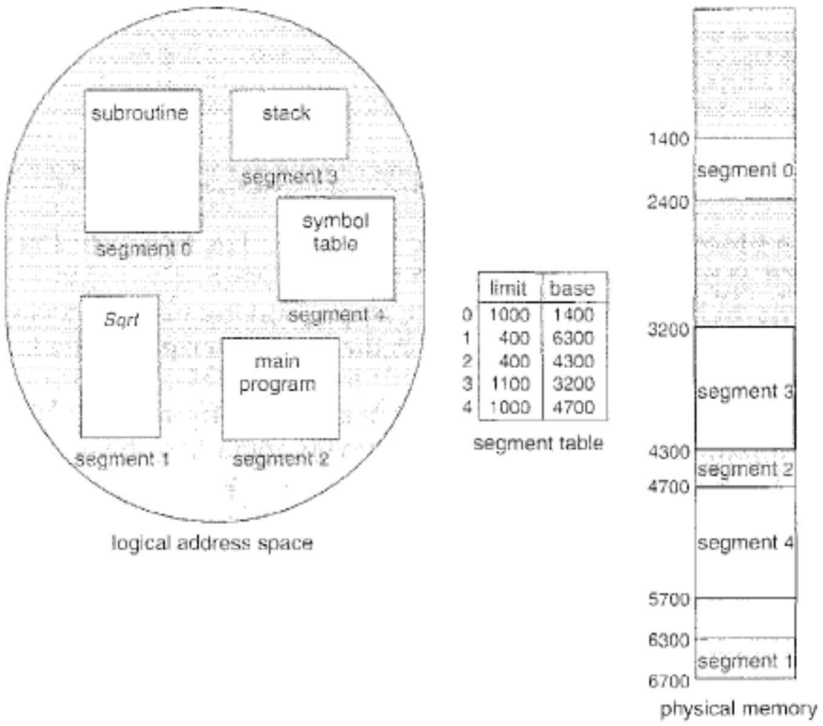

Figure 3.37 Example of segmentation.

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 would result in a trap to the operating system, as this segment is only 1,000 bytes long.