Chapter 7

VIRTUAL MEMORY

Virtual memory is a technique that allows the execution of processes that may not be completely in memory. The main visible advantage of this scheme is that programs can be larger than physical memory.

Further, it abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. 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.

7.1 Background

The memory-management algorithms of Chapter 8 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 restriction 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 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 seldom larger than 10 by 10 elements. An assembler symbol

table may have room for 3000 symbols, although the average program has less than 200 symbols.

- Certain options and features of a program may be used rarely. For instance, the routines on U.S. government computers which balance the budget have not been used in years.

Even in those cases where the entire program is needed, it may not all 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.

- 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.

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 7.1).

Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available, 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.

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.

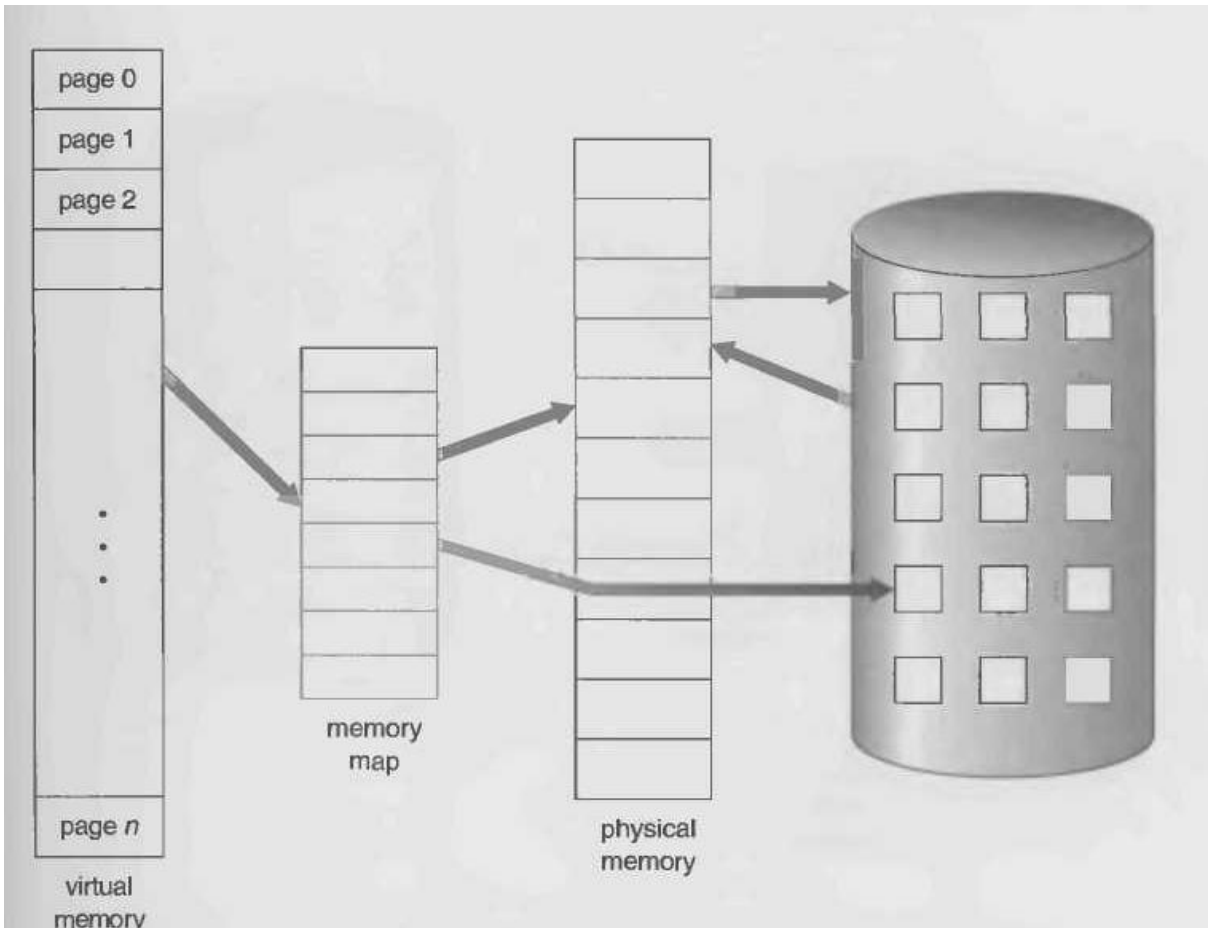

Figure 7.1 Diagram showing virtual memory larger than physical memory.

The IBM OS/2 operating system also uses demand segmentation. However, segment-replacement algorithms are more complex than are page-replacemen algorithms because the segments have variable sizes.

7.2 Demand Paging

A demand-paging system is similar to a paging system with swapping (Figure 7.2). 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.

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 one large contiguous address space, the use of the term *swap* is technically incorrect.

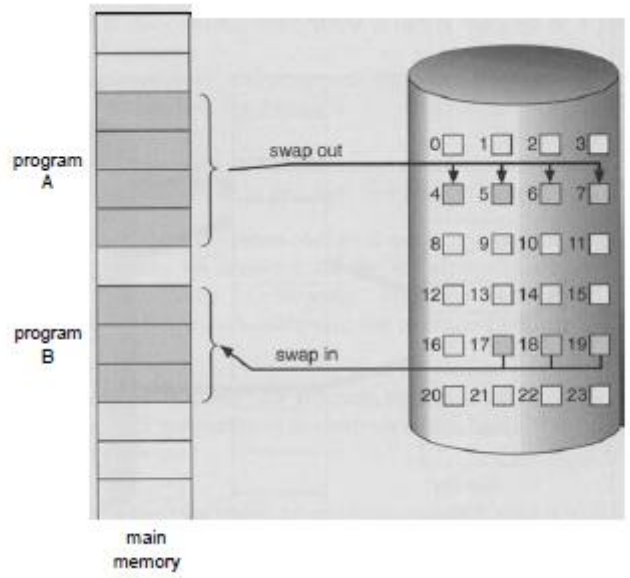

Figure 7.2 Transfer of a paged memory to contiguous disk space.

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 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 those pages that are in memory and those pages that are on the disk. The *valid-invalid* bit scheme can be used for this purpose. This time, however, when this bit is set to "valid," 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. 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 simply marked invalid, or contains the address of the page on disk. This situation is depicted in Figure 7.3.

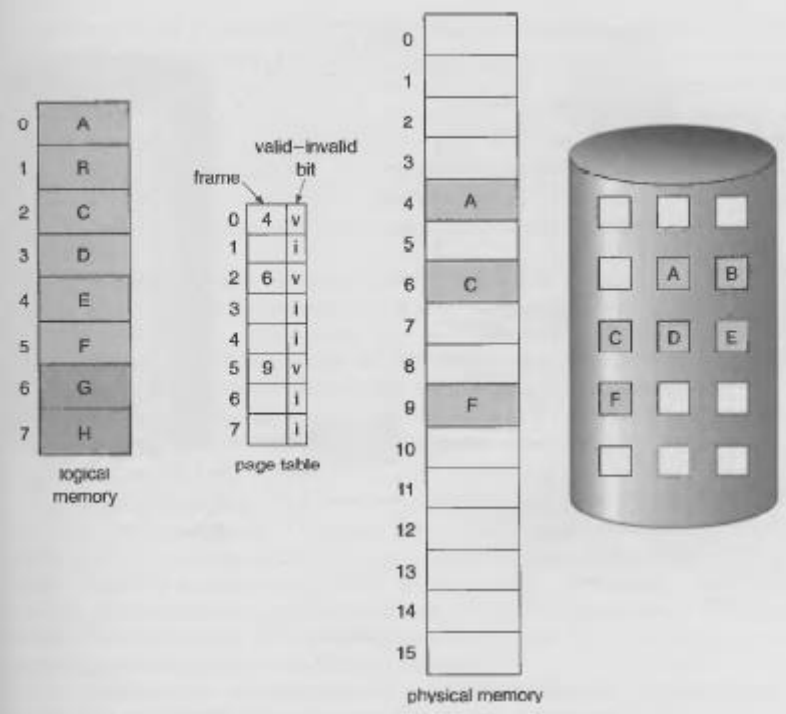

Figure 9.3 Page table when some pages are not in main memory.

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 page in all and only those pages that are actually needed, 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 9.4):

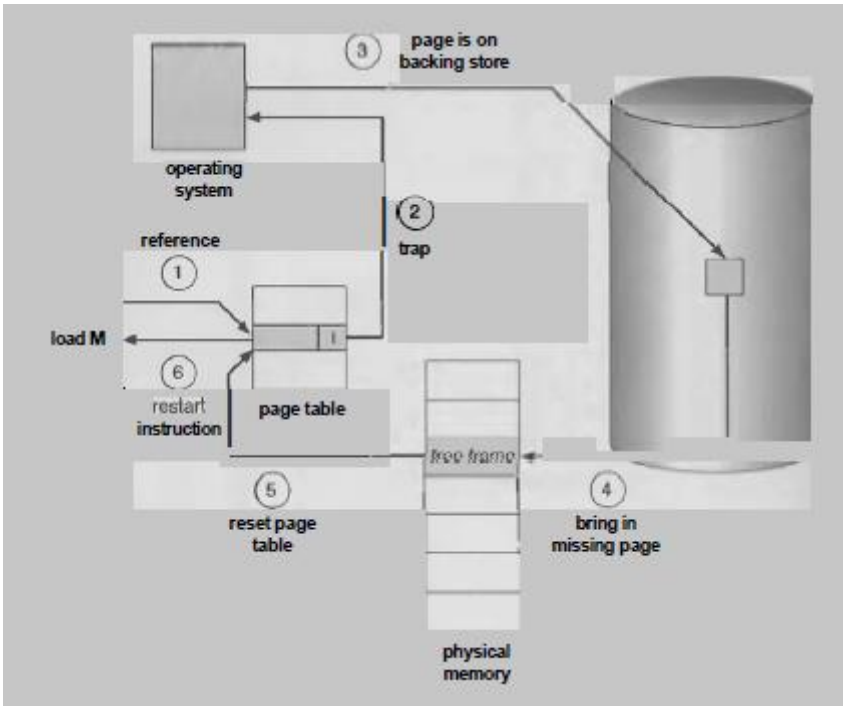

Figure 7.4 Steps in handling a page fault.

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

2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page in the latter.

3. We find a free frame (by taking one from the free-frame list, for example).

4. We schedule a disk operation to read the desired page into the newly allocated frame.

5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.

6. We restart the instruction that was interrupted by the 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 to the operating system (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, which is on a non-memory-resident page, the process would immediately fault for the page.

After this page was brought into memory, the process would continue to execute, faulting as necessary until every page that it needed was actually in memory. At that point, it could excecute 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. Fortunately, analysis of running processes show that this behavior is exceedingly unlikely.

Programs **tend** to have ***locality of reference,*** described in Section 7.7.1, which results in reasonable performance from demand paging.

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

1. Page table: This table has the ability to mark an entry invalid through a valid-invalid bit or special value of protection bits.

2. 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.***

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 anoperand, we must re-fetch the instruction, decode it again, and then fetch the operand.

As a worst case, 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, bring it in, 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 (possibly 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 microcode computes and 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 autodecrement and autoincrement modes (for example, the PDP-11). These addressing modes use a register as a pointer and automatically decrement or increment the register as indicated.

Autodecrement automatically decrements the register *before* using its contents as the operand address; autoincrement 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 a 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 that is changed during the execution of an instruction. This status 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.

7.3 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 *5* p *5* **1).** We would expect p to be close to zero; that is, there will be only a few page faults. The gfective 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. 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, 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 whiie the I/O occurs. This arrangement allows multiprogramming to maintain CPU utilization, but requires additional time to resume the page-fault service routine when the I/O transfer is complete.

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

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

The first and third tasks 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 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 queueing time as we wait for the paging device to be free to service our request, increasing the time to swap even more.

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

effective access time = (1 - p) x (100) + p (25 milliseconds)
= (1 - p) **x** 100 + p x 25,000,000
= 100 + 24,999,900 x p.

We see then that the effective access time is directly proportional to the page-fault rate. 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-percent degradation, we need

$$110 > 100 + 25{,}000{,}000 \times p,$$
$$10 > 25{,}000{,}000 \times p,$$
$$p < 0.0000004.$$

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.

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 **1/0** 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; it is used in BSD UNIX.

7.4 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, then demand paging saves the I/O necessary to load the five pages that are never used. We could also increase our degree of multiprogramming 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 multiprogramming, we are ***ozler-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.

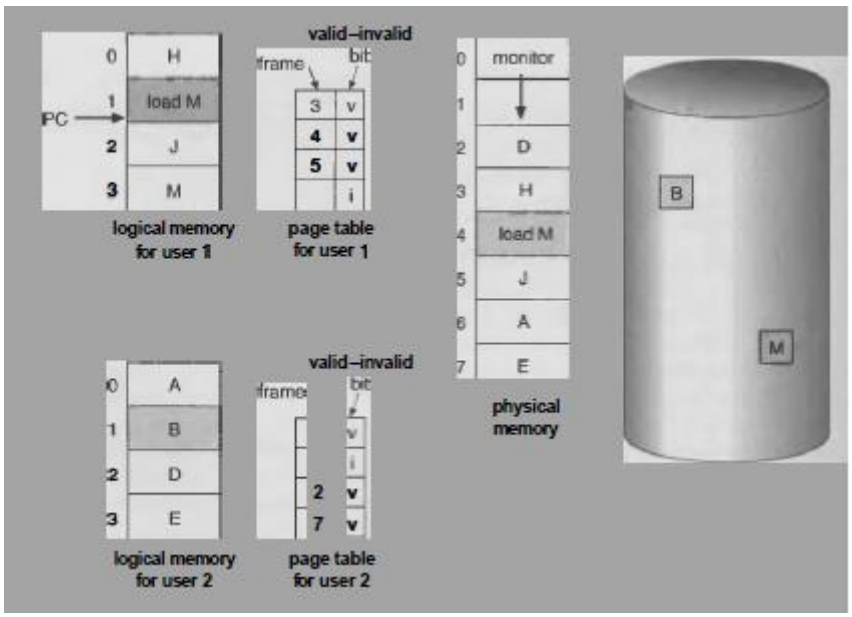

Figure 7.5 Need for page replacement.

Over-allocating will show up in the following way. While a user process is executing, a page fault occurs. The hardware traps to the operating system, which checks its internal tables to see that this is a page fault and not an illegal memory access. The operating system determines where the desired page is residing on the disk, but then finds there are no free frames on the free-frame list; all memory is in use (Figure 7.5). The operating system has several options at this point. It could terminate the user process. However, demand paging is something that the operating system is doing to improve the computer system's 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. So this option is not the best choice.

We could swap out a process, freeing all its frames, and reducing the level of multiprogramming. This option is a good idea at times, and we consider it further in Section 7.7. First, we shall discuss a more intriguing possibility: 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 9.6). The freed frame can now be used to hold the page for which the process faulted. The page-fault service routine is now modified 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 frame, use it.
    b. Otherwise, use a page-replacement algorithm to select a victim frame.
    c. Write the victim page to the disk; change the page and frame tables accordingly.
3. Read the desired page into the (newly) free frame; change the page and frame tables.
4. Restart the user process.

Notice that, 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 will increase the effective access time accordingly.

This overhead can be reduced by the use of a modih (dirty) bit. Each page or frame may have 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 its modify bit.

If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write that page to the disk. If the modify bit is not set, however, the page has not been modified since it was read into memory.

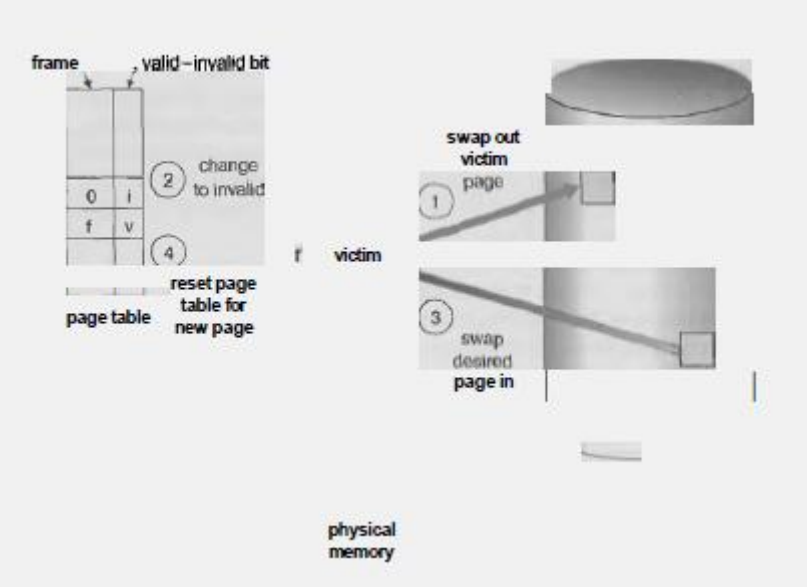

Figure 7.6 Page replacement.

Therefore, if the copy of the page on the disk has not been overwritten (by some other page, for example), we can avoid writing the memory page to the disk; it is already there. This technique also applies to read-only pages (for example, pages of binary code). Such pages cannot be modified; thus, they may be discarded when desired. This scheme can reduce significantly the time to service a page fault, since it reduces I/O time by one-half *if* the page is not modified.

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, a very large virtual memory can be provided for programmers on a smaller physical

memory. With non-demand paging, user addresses were mapped into physical addresses, allowing the two sets of addresses to be quite different. All of the pages of a process still must be in physical memory however. With demand paging, the size of the logical address space is no longer constrained by physical memory.

If we have a user process of 20 pages, we can execute it in 10 frames simply by using demand paging, and using a replacement algorithm to find a free frame whenever necessary. If a page that has been modified is to be replaced, its contents are copied to the disk. A later reference to that page will cause a page fault. At that time, the page will be brought back into memory, perhaps replacing some other page in the process.

We must solve two major problems to implement demand paging: We must develop aframe-allocation algorithm and a page-replacement algorithm. If we have multiple processes in memory, we must decide how many frames to allocate to each process. Further, when page replacement is required, we must select the frames that are to be replaced. Designing appropriate algorithms to solve these problems is an important task, because disk I/O is so expensive. Even slight improvements in demand-paging methods yield large gains in system performance.

7.5 Page-Replacement Algorithms

There are many different page-replacement algorithms. Probably every operating system has its own unique replacement scheme. How do we select a particular replacement algorithm? In general, we want the one with the lowest page-fault rate.

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 refevence string. We can generate reference strings artificially (by a random-number generator, for example) or by tracing a given system and recording 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 note two things.

First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, not the entire address. 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; the immediately following references will not fault.

For example, if we trace a particular process, we might record the following address sequence:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105,

which, at 100 bytes per page, is reduced to the following reference string

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

To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available.

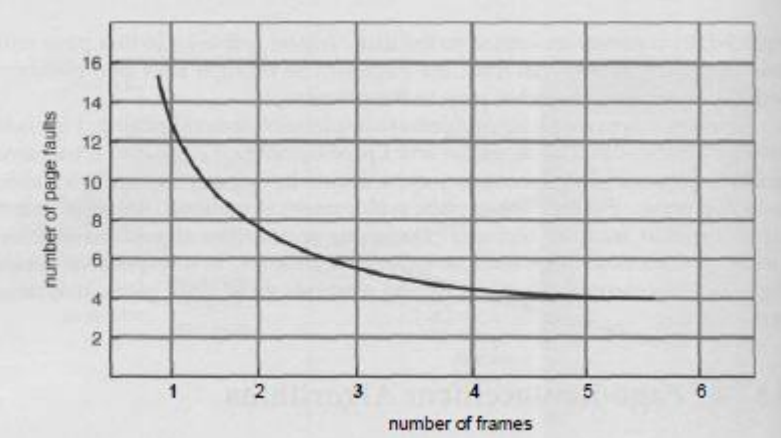

Figure 7.7 Graph of page faults versus the number of frames.

Obviously, as the number of frames available increases, the number of page faults will decrease. For the reference string considered previously, for example, if we had three or more frames, we would have only three faults, one fault for the first reference to each page. On the other hand, with only one frame available, we would have a replacement with every reference, resulting in 11 faults.

In general, we expect a curve such as that in Figure 7.7. As the number of frames increases, the number of page faults drops to some minimal level. Of course, adding physical memory increases the number of frames.

To illustrate the page-replacement algorithms, we shall use the reference String for a memory with three frames.

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

7.5.1 FIFO Algorithm

The simplest page-replacement algorithm is a *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.

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.

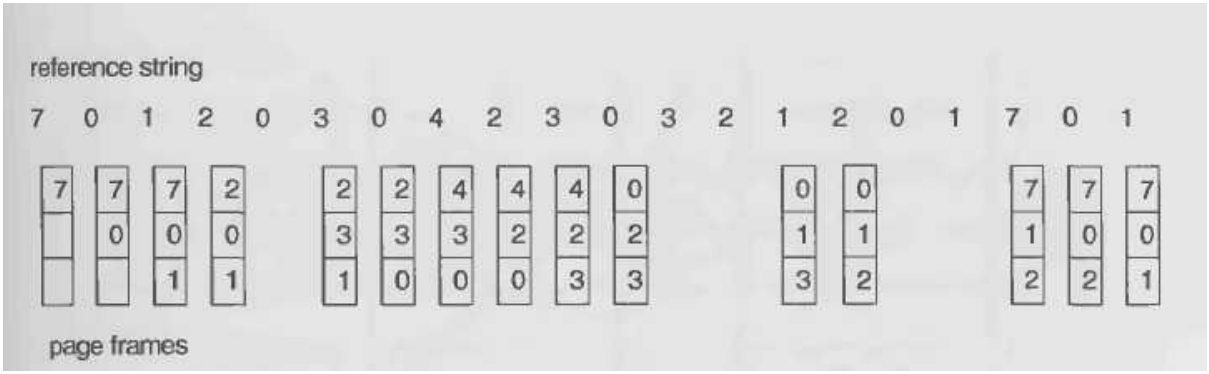

Figure 7.8 FIFO page-replacement algorithm.

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 page 0 being replaced, since it was the first of the three pages in memory (0, 1, and 2) to be brought in.

This replacement means that the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 7.8. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. The page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.

Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we page out an active page to bring in a new one, a fault occurs almost immediately to retrieve the active page. Some other page will need to be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution, but does not cause incorrect execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the reference string

1,2,3,4,1,2,5,1,2,3,4, 5.

Figure 7.9 shows the curve of page faults versus the number of available frames.

We notice that the number of faults for four frames (10) is *greater* than the number of faults for three frames (nine)! This result is most unexpected and is known as *Belady's anomaly*. Belady's anomaly reflects the fact that, for some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

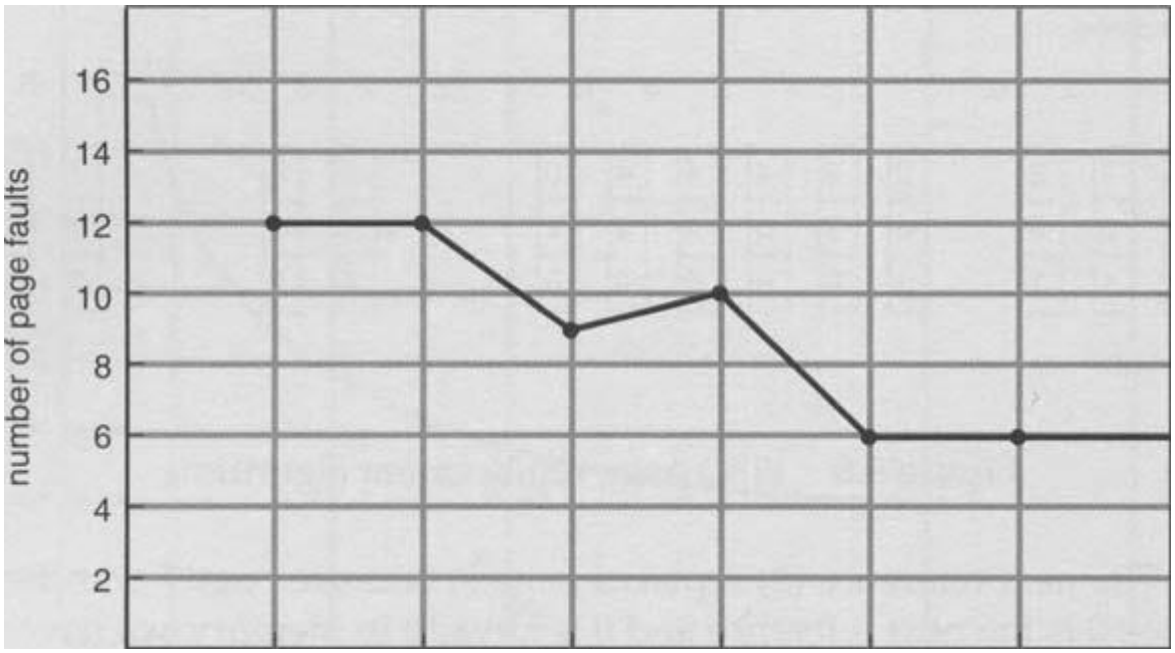

Figure 7.9 Page-fault curve for FIFO replacement on a reference string.

7.5.2 Optimal Algorithm

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. An optimal algorithm will never suffer from Belady's anomaly. An optimal page-replacement algorithm exists, and has been called OPT or MIN. It is simply Replace the page that will not be used for the longest period of time.

Use of this page-replacement algorithm guarantees the lowest possible pagefault rate for a fixed number of frames. For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 9.10. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page *0* will be used at ***5,*** and page 1 at 14. The reference to page **3** replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again.

With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults. (If we ignore the first three, which all

algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with less than nine faults.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

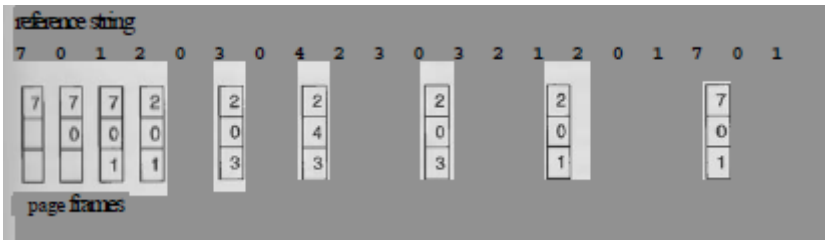

Figure 7.10 Optimal page-replacement algorithm.

As a result, the optimal algorithm is used mainly for comparison studies. For instance, it may be quite useful to know that, although a new algorithm is not optimal, it is within 12.3 percent of optimal at worst and within 4.7 percent on average.

7.5.3 LRU Algorithm

If the optimal algorithm is not feasible, perhaps an approximation to the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward or forward in time) is that the FIFO algorithm uses the time when a page was brought into memory; 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 will replace the page that *has not been used* for the longest period of time (Figure 9.11). This approach is the *least recently used **(LRU)*** algorithm.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time. This strategy is the optimal page-replacement algorithm looking backward in time, rather than forward. (Strangely, if we let SR be the reverse of a reference string S, then the page-fault rate for the OPT algorithm on S is the same as the page-fault rate for the OPT algorithm on reference string SR.

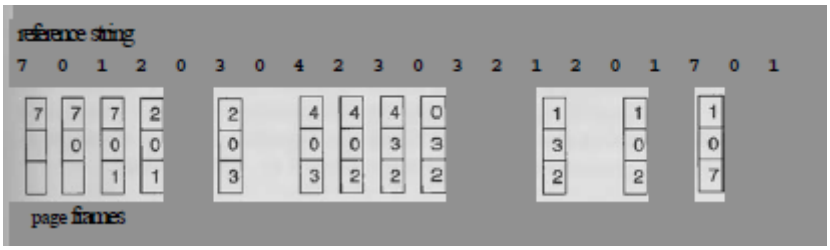

Figure 7.11 LRU page-replacement algorithm.

Similarly, the page-fault rate for the LRU algorithm on S is the same as the page-fault rate for the LRU algorithm on SR.) The result of applying LRU replacement to our example reference string is shown in Figure 9.11. The LRU algorithm produces 12 faults. Notice that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. The most recently used page is page **0,** and just before that page **3** was used. 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, of the three pages in memory **{0,3,4),** page **3** is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15.

The LRU policy is often used as a page-replacement algorithm and is considered to be quite good. The major problem is ***how*** to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

1. **Counters:**

In the simplest case, we associate with each page-table entry a time-of-use field, and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a -reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page table for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value.

This scheme requires a search of the page table to find the LRU page, and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.

2. Stack:

Another approach to implementing LRU replacement is to keep a ***stack*** of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the top of the stack is always the most recently used page and the bottom is the LRU page (Figure 7.12). Because entries must be removed from the middle of the stack, it is best implemented by a doubly linked list, with a head and tail pointer.

Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode implementations of LRU replacement.

Neither optimal replacement nor LRU replacement suffers from Belady's anomaly. There is 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.

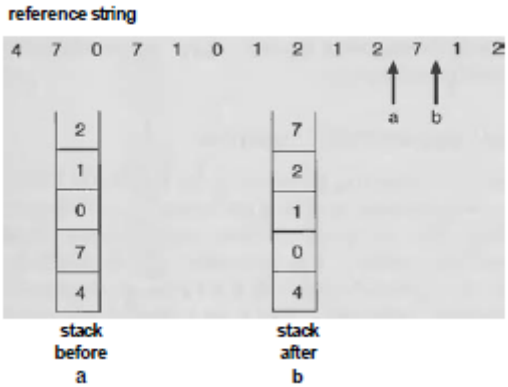

Figure 7.12 Use of a stack to record the most recent page references.

For LRU replacement, the set of pages in memory would be the *n* most recently referenced pages. If the number of frames is increased, these *n* pages will still be the most recently referenced and so will still be in memory.

Note that neither implementation of LRU would be conceivable without hardware assistance beyond the standard TLB registers. The updating of the clock fields or stack must be done for every memory reference. If we were to use an interrupt for every reference, to allow software to update such data structures, it would slow every memory reference by a factor of at least 10, hence slowing every user process by a factor of 10. Few systems could tolerate that level of overhead for memory management.

7.5.4 LRU Approximation Algorithms

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 *re$erence bit.* The reference bit for a page is set, by the hardware, whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.

Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits. We do not know the ***order*** of use, but we know which pages were used and which were not used. This partial ordering information leads to many page-replacement algorithms that approximate LRU replacement.

7.5.4.1 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 &bit byte, shifting the other bits right 1 bit, 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,then the page has not been used for eight time periods; a page that is used at leasc once each period would have a shift register value of 11111111.

A page with a history register value of 11000100 has been used more recently than has one with 01110111. If we interpret these &bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced. Notice that the numbers are not guaranteed to be unique, however. We can either replace (swap out) all pages with the smallest value, or use a FIFO selection among them.

The number of bits of history can be varied, of course, and would be 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.

7.5.4.2 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. If the reference bit is 1, however, we give that page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared and its arrival time is reset to the current time.

Thus, a page that is given a second chance will not be replaced until all other pages are replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

One way to implement the second-chance (sometimes referred to as the clock) algorithm is as a circular queue. A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits (Figure 7.13).

Once a victim page is found, the page is replaced and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance.

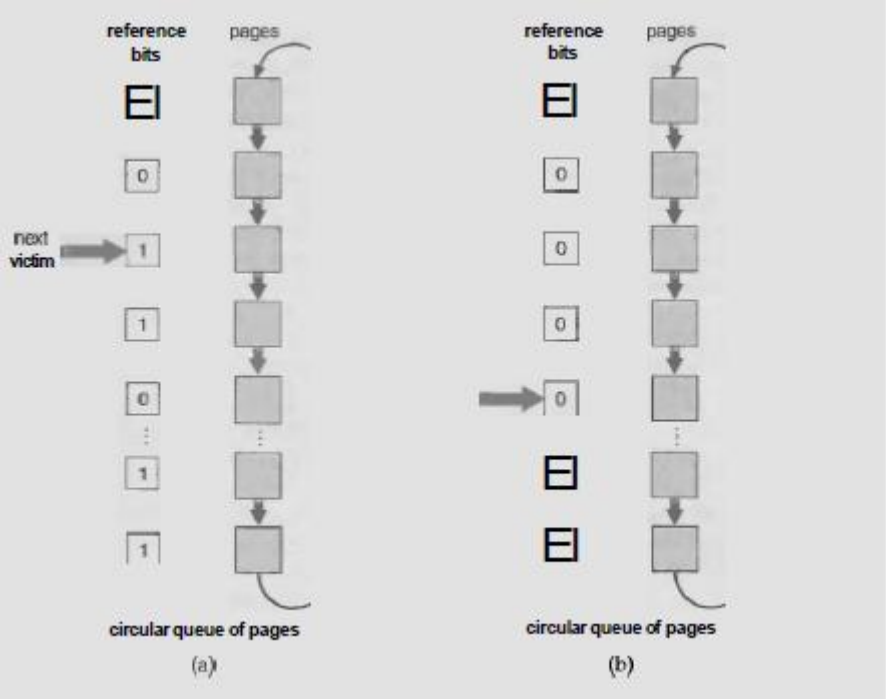

Figure 7.13 Second-chance (clock) page-replacement algorithm.

It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.

7.5.4.3 Enhanced Second-Chance Algorithm

The second-chance algorithm described above can be enhanced by considering both the reference bit and the modify bit (Section 9.4) as an ordered pair. With these 2 bits, we have the following four possible classes:

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

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

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

4. (1,l) recently used and modified-probably will be used again, and write out will be needed before replacing it When page replacement is called for, each page is in one of these four classes.

We use the same scheme as the clock algorithm, but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced.

This algorithm is used in the Macintosh virtual-memory-management scheme. The major difference between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified to reduce the number of I/Os required.

7.5.5 Counting Algorithms

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

1. LFU Algorithm:

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.

This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

2. MFU Algorithm:

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.As you might expect, neither MFU nor LFU replacement is common. The implementation of these algorithms is fairly expensive, and they do not approximate OPT replacement very well.

7.5.6 Page Buffering Algorithm

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. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to the disk. [ts modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement, and will not need to be written 3ut.

Another modification is to keep a pool of free frames, but to remember which page was in each frame. Since the frame contents are not modified when 3 frame is written to the disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused. No I/O is needed in this case. When a page fault occurs, we first check whether the desired page is in the free-frame pool. If it is not, we must select a free frame and read into it.

This technique is used in the VAX/VMS system, with a FIFO replacement algorithm. When the FIFO replacement algorithm mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame buffer, and no 1/0 is necessary. The free-frame buffer provides protection against the relatively poor, but simple, FIFO replacement algorithm. This method is necessary because the early versions of the VAX did not correctly implement the reference bit.

7.6 Allocation of Frames

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 of virtual memory is the single-user system.

Consider a single-user system with 128K memory composed of pages of size 1K. Thus, there are 128 frames. The operating system may take 35K, 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 pagereplacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the ninety-fourth, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

There are many variations on this simple strategy. We can require that the operating system allocate all its buffer and table space from the free-frame list. When this space is not in use by the operating system, it can be used to support user paging. We could try to keep three free frames reserved on the free-frame ht at all times. Thus, when a page fault occurs, there is a free frame available to page into. While the page swap is taking place, a replacement can be selected, which is then written to the disk as the user process continues to execute.

Other variants are also possible, but the basic strategy is clear: The user process is allocated any free frame. A different problem arises when demand paging is combined with multiprogramming.Multiprogramming puts two (or more) processes in memory at the same time.

7.6.1 Minimum Number of Frames

There are, of course, various constraints on our strategies for the allocations of frames. We cannot allocate more than the total number of available frames (unless there is page sharing). There is also a minimum number of frames that can be allocated. Obviously, as the number of frames allocated to each process decreases, the page fault-rate increases, slowing process execution.

Besides the undesirable performance properties of allocating only a few

frames, there is a minimum number of frames that must be allocated. This minimum number is defined by the instruction-set architecture. 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.

For example:

consider a machine in which all memory-reference instructions have only one memory address. Thus, we need at least one frame for the instruction and one frame for the memory reference. In addition, if onelevel indirect addressing is allowed (for example, a load instruction on page 16 can refer to an address on page 0, which is an indirect reference to page 23), then paging requires at least three frames per process. Think about what might happen if a process had only two frames.

The minimum number of frames is defined by the computer architecture. For example, the move instruction for the PDP-11 is more than one word for some addressing modes, and thus the instruction itself may straddle two pages. In addition, each of its two operands may be indirect references, for a total of six frames. The worst case for the IBM 370 is probably the MVC instruction.

Since the instruction is storage to storage, it takes 6 bytes and can straddle two pages. The block of characters to move and the area to be moved to can each also straddle two pages. This situation would require six frames. (Actually, the worst case is if the MVC instruction is the operand of an EXECUTE instruction that straddles a page boundary; in this case, we need eight frames.)

The worst-case scenario occurs in architectures that allow multiple levels of indirection (for example, each 16-bit word could contain a 15-bit address plus a 1-bit indirect indicator). Theoretically, a simple load instruction could reference an indirect address that could reference an indirect address (on another page) that could also reference an indirect address (on yet another page), and so on, until every page in virtual memory had been touched.

Thus, in the worst case, the entire virtual memory must be in physical memory. To overcome this difficulty, we must place a limit on the levels of indirection (for example, limit an instruction to at most 16 levels of indirection). When the first indirection occurs, a counter is set to 16; the counter is then

decremented for each successive indirection for this instruction. If the counter is decremented to 0, a trap occurs (excessive indirection).

This limitation reduces the maximum number of memory references per instruction to 17, requiring the same number of frames. The minimum number of frames per process is defined by the architecture, whereas the maximum number is defined by the amount of available physical memory. In between, we are still left with significant choice in frame allocation.

7.6.2 Allocation Algorithms

The easiest way to split *m* 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 could 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. If a small student process of 10K and an interactive database of 127K 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 wasted.

To solve this problem, we can use *proportional allocation.* We allocate available memory to each process according to its size. Let the size of the virtual memory for process p,- be s,-, and define Then, if the total number of available frames is *m*, we allocate a,- frames to process p,-, where *a*, is approximately

$$a_i = s_i / S \times m.$$

Of course, we must adjust each *ai* to be an integer, which is greater than the minimum number of frames required by the instruction set, with a sum not exceeding *m.*

For proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating four frames and 57 frames, respectively, since

$10/137 \times 62 \approx 4,$
$127/137 \times 62 \approx 57.$

In this way, both processes share the available frames according to their "needs," rather than equally.

In this way, both processes share the available frames according to their "needs," rather than equally.In both equal and proportional allocation, of course, the allocation to each process may vary according to the multiprogramming level. If the multipre gramming level is increased, each process will lose some frames to provide the memory needed for the new process.

On the other hand, if the multiprogramming level decreases, the frames that had been allocated to the departed process can now be spread over the remaining processes.

Notice that, with either equal or proportional allocation, a high-priority process is treated the same as a low-priority process. By its definition, however, we may want to give the high-priority process more memory to speed its execution, to the detriment of low-priority processes.

One approach is to use a proportional allocation scheme where the ratio of frames depends not on the relative sizes of processes, but rather on the processes' priorities, or on a combination of size and priority.

7.6.3 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; one process can take a frame from another.

Local replacement 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 the low-priority process.

With a local replacement strategy, the number of frames allocated to a process does not change. With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it (assuming that other processes do not choose ***its*** frames for replacement).

One problem with a global replacement algorithm is that a process cannot control its own page-fault rate. The set of pages in memory for a process depends not only on the paging behavior of that process, but also on the paging behavior of other processes. Therefore, the same process may perform quite differently (taking 0.5 seconds for one execution and 10.3 seconds for the next execution) due to totally external circumstances. Such is not the case with a local replacement algorithm.

Under local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process. For its part, local replacement might hinder a process by not making available to other, less used pages of memory. Thus, global replacement generally results in greater system throughput, and is therefore the more common method.

7.7 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' 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. Although it is technically possible to reduce the number of allocated frames to the minimum, there is some (larger) number of pages that are in active use. If the process does not have this number of frames, it will very 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 very quickly faults again, and again, and again. The process continues to fault, replacing pages for which it will then fault and bring back in right away.

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

7.7.1 Cause of Thrashing

Thrashing results in severe performance problems. Consider the following scenario, which is based on the actual behavior of early paging systems.

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, replacing pages with no regard to the process to which they belong.

Now suppose a process enters a new phase in its execution and needs more frames. It starts faulting and taking pages away from other processes. These processes need those pages, however, and so they also fault, taking pages 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.

The CPU scheduler sees the decreasing CPU utilization, and ***increases*** the degree of multiprogramming as a result. The new process tries to get started by taking pages from running processes, causing more page faults, and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more.

Thrashing has occurred and system throughput plunges. The pagefault rate increases tremendously. As a result, the effective memory access time increases.

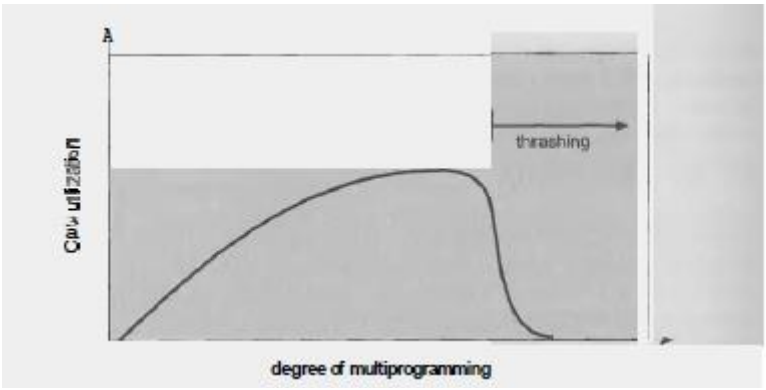

Figure 7.14 Thrashing.

No work is getting done, because the processes are spending all their time paging. This phenomenon is illustrated in Figure ***9.14.*** CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must ***decrease*** the degree of mu1 tiprogramming.

The effects of thrashing can be limited by using a ***local*** (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 also. Pages are replaced with regard to the process of which they are a part. However, if processes are thrashing, they will be in the queue for the paging device mod of the time.

The average service time for a page fault will increase, due to the longer average queue for the paging device. Thus, the effective access time will increase even for a process that is not thrashing.

To prevent thrashing, we must provide a process as many frames as il needs. But how do we know how many frames it "needs"? There are several techniques. The working-set strategy (discussed in Section 7***.7.2)*** starts bj looking at how many frames a process is actually using. This approach define the ***locality model*** of process execution.

The locality model states that, as a process executes, it moves from localitj to locality. A locality is a set of pages that are actively used together

(Figure 7***.15).*** A program is generally composed of several different localities, which may overlap.

For example, when a subroutine is called, it defines a new locality. In this local variables, and a subset of the global variables. When the subroutine is exited, the process leaves this locality, since the local variables and instructions of the subroutine are no longer in active use. We may return to this locality later. Thus, we see that localities are defined by the program structure and its data structures.

locality, memory references are made to the instructions of the subroutine, its

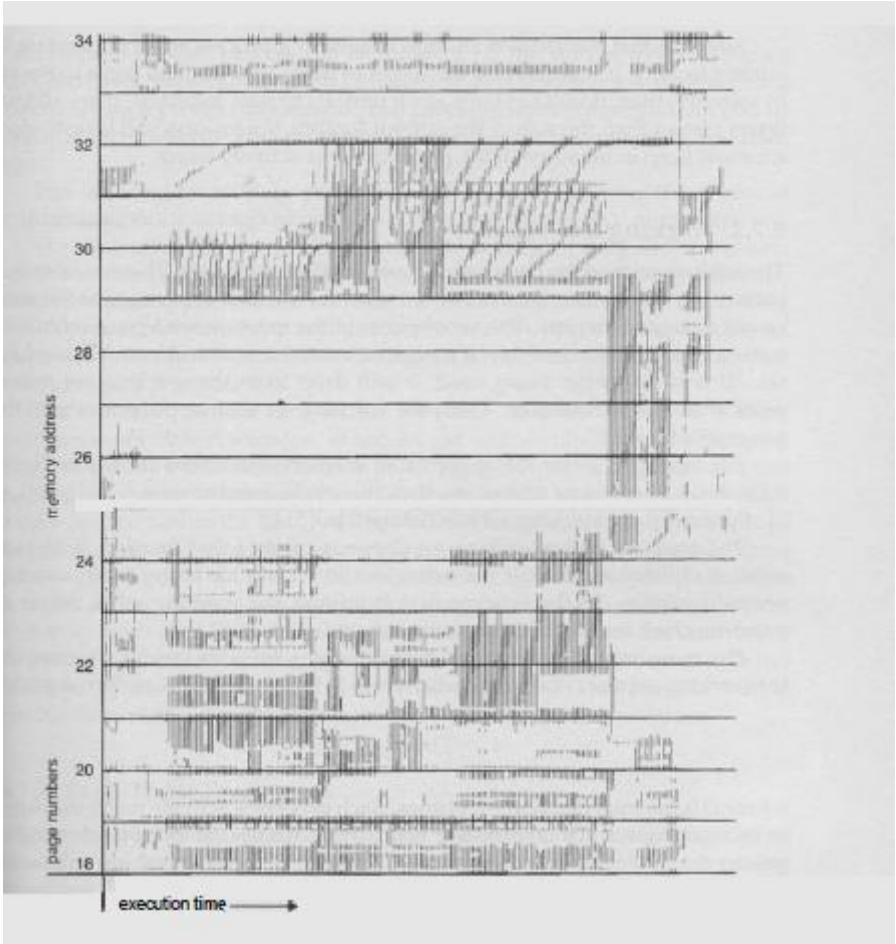

Figure 7.15 Locality in a memory reference pattern.

The locality model states that all programs will exhibit this basic memory reference structure. Note that the locality model is the unstated principle behind

the caching discussions so far in this book. If accesses to any types of data were random rather than patterned, caching would be useless.

Suppose that we allocate enough frames to a process to accommodate its current locality. It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities. If we allocate fewer frames than the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.