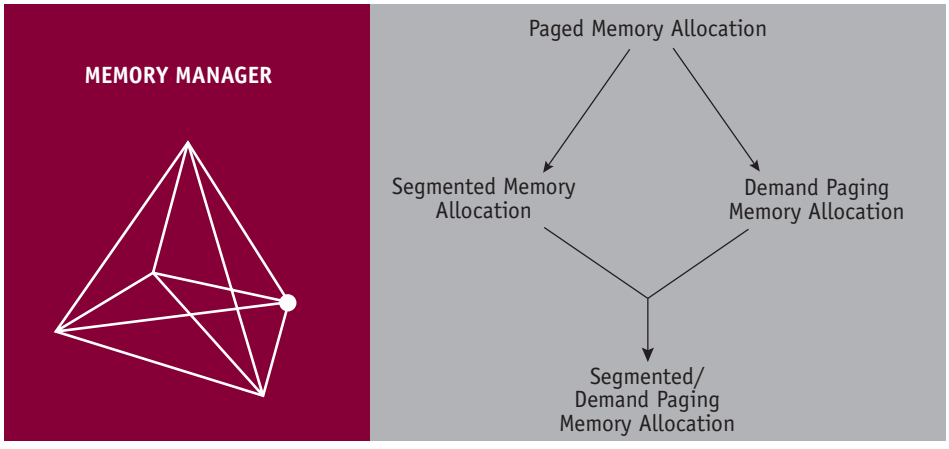
Chapter 3 Memory Management: Virtual Memory



"Nothing is so much strengthened by practice, or weakened by neglect, as memory."

—Quintillian *(A.D. 35–100)*

Learning Objectives

After completing this chapter, you should be able to describe:

- The basic functionality of the memory allocation methods covered in this chapter: paged, demand paging, segmented, and segmented/demand paged memory allocation
- The influence that these page allocation methods have had on virtual memory
- The difference between a first-in first-out page replacement policy, a least-recently-used page replacement policy, and a clock page replacement policy
- The mechanics of paging and how a memory allocation scheme determines which pages should be swapped out of memory
- The concept of the working set and how it is used in memory allocation schemes
- The impact that virtual memory had on multiprogramming
- Cache memory and its role in improving system response time

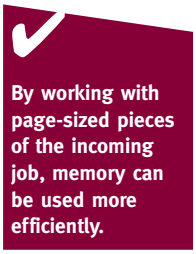
In the previous chapter we looked at simple memory allocation schemes. Each one required that the Memory Manager store the entire program in main memory in contiguous locations; and as we pointed out, each scheme solved some problems but created others, such as fragmentation or the overhead of relocation.

In this chapter we'll follow the evolution of virtual memory with four memory allocation schemes that first remove the restriction of storing the programs contiguously, and then eliminate the requirement that the entire program reside in memory during its execution. These schemes are paged, demand paging, segmented, and segmented/demand paged allocation, which form the foundation for our current virtual memory methods. Our discussion of cache memory will show how its use improves the performance of the Memory Manager.

Paged Memory Allocation

Before a job is loaded into memory, it is divided into parts called pages that will be loaded into memory locations called page frames. **Paged memory allocation** is based on the concept of dividing each incoming job into **pages** of equal size. Some operating systems choose a page size that is the same as the memory block size and that is also the same size as the sections of the disk on which the job is stored.

The sections of a disk are called **sectors** (or sometimes blocks), and the sections of main memory are called **page frames.** The scheme works quite efficiently when the pages, sectors, and page frames are all the same size. The exact size (the number of bytes that can be stored in each of them) is usually determined by the disk's sector size. Therefore, one sector will hold one page of job instructions and fit into one page frame of memory.



Before executing a program, the Memory Manager prepares it by:

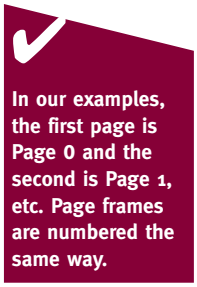
1. Determining the number of pages in the program
2. Locating enough empty page frames in main memory
3. Loading all of the program's pages into them

When the program is initially prepared for loading, its pages are in logical sequence—the first pages contain the first instructions of the program and the last page has the last instructions. We'll refer to the program's instructions as bytes or words.

The loading process is different from the schemes we studied in Chapter 2 because the pages do not have to be loaded in adjacent memory blocks. In fact, each page can be stored in any available page frame anywhere in main memory.

The primary advantage of storing programs in noncontiguous locations is that main memory is used more efficiently because an empty page frame can be used by any page of any job. In addition, the compaction scheme used for relocatable partitions is eliminated because there is no external fragmentation between page frames (and no internal fragmentation in most pages).

However, with every new solution comes a new problem. Because a job's pages can be located anywhere in main memory, the Memory Manager now needs a mechanism to keep track of them—and that means enlarging the size and complexity of the operating system software, which increases overhead.

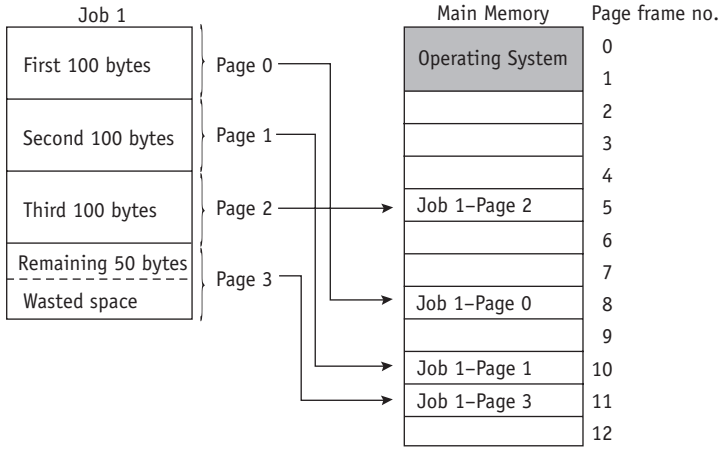


In our examples, the first page is Page 0 and the second is Page 1, etc. Page frames are numbered the same way.

The simplified example in Figure 3.1 shows how the Memory Manager keeps track of a program that is four pages long. To simplify the arithmetic, we've arbitrarily set the page size at 100 bytes. Job 1 is 350 bytes long and is being readied for execution.

Notice in Figure 3.1 that the last page (Page 3) is not fully utilized because the job is less than 400 bytes—the last page uses only 50 of the 100 bytes available. In fact, very few jobs perfectly fill all of the pages, so internal fragmentation is still a problem (but only in the last page of a job).

In Figure 3.1 (with seven free page frames), the operating system can accommodate jobs that vary in size from 1 to 700 bytes because they can be stored in the seven empty page frames. But a job that is larger than 700 bytes can't be accommodated until Job 1 ends its execution and releases the four page frames it occupies. And a job that is larger than 1100 bytes will never fit into the memory of this tiny system. Therefore, although



(figure 3.1)

Programs that are too long to fit on a single page are split into equal-sized pages that can be stored in free page frames. In this example, each page frame can hold 100 bytes. Job 1 is 350 bytes long and is divided among four page frames, leaving internal fragmentation in the last page frame. (The Page Map Table for this job is shown later in Table 3.2.)

paged memory allocation offers the advantage of noncontiguous storage, it still requires that the entire job be stored in memory during its execution.

Figure 3.1 uses arrows and lines to show how a job's pages fit into page frames in memory, but the Memory Manager uses tables to keep track of them. There are essentially three tables that perform this function: the Job Table, Page Map Table, and Memory Map Table. Although different operating systems may have different names for them, the tables provide the same service regardless of the names they are given. All three tables reside in the part of main memory that is reserved for the operating system.

As shown in Table 3.1, the **Job Table** (**JT**) contains two values for each active job: the size of the job (shown on the left) and the memory location where its Page Map Table is stored (on the right). For example, the first job has a job size of 400 located at 3096 in memory. The Job Table is a dynamic list that grows as jobs are loaded into the system and shrinks, as shown in (b) in Table 3.1, as they are later completed.

Job Table

Job Size	PMT Location
400	3096
200	3100
500	3150

(a)

Job Table

Job Size	PMT Location
400	3096
500	3150

(b)

Job Table

Job Size	PMT Location
400	3096
700	3100
500	3150

(c)

(table 3.1)

This section of the Job Table (a) initially has three entries, one for each job in progress. When the second job ends (b), its entry in the table is released and it is replaced (c) by information about the next job that is to be processed.

Each active job has its own **Page Map Table** (**PMT**), which contains the vital information for each page—the page number and its corresponding page frame memory address. Actually, the PMT includes only one entry per page. The page numbers are sequential (Page 0, Page 1, Page 2, through the last page), so it isn't necessary to list each page number in the PMT. The first entry in the PMT lists the page frame memory address for Page 0, the second entry is the address for Page 1, and so on.

The **Memory Map Table** (**MMT**) has one entry for each page frame listing its location and free/busy status.

At compilation time, every job is divided into pages. Using Job 1 from Figure 3.1, we can see how this works:

- Page 0 contains the first hundred bytes.
- Page 1 contains the second hundred bytes.

• Page 2 contains the third hundred bytes.
• Page 3 contains the last 50 bytes.

As you can see, the program has 350 bytes; but when they are stored, the system numbers them starting from 0 through 349. Therefore, the system refers to them as byte 0 through 349.

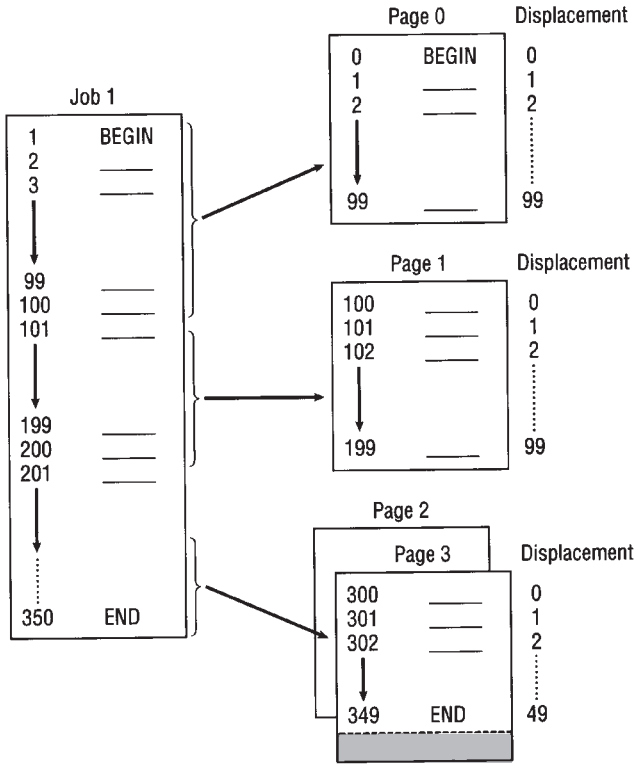
The **displacement**, or **offset**, of a byte (that is, how far away a byte is from the beginning of its page) is the factor used to locate that byte within its page frame. It is a relative factor.

In the simplified example shown in Figure 3.2, bytes 0, 100, 200, and 300 are the first bytes for pages 0, 1, 2, and 3, respectively, so each has a displacement of zero. Likewise, if the operating system needs to access byte 214, it can first go to page 2 and then go to byte 14 (the fifteenth line).

The first byte of each page has a displacement of zero, and the last byte, has a displacement of 99. So once the operating system finds the right page, it can access the correct bytes using its relative position within its page.

(figure 3.2)

Job 1 is 350 bytes long and is divided into four pages of 100 lines each.



In this example, it is easy for us to see intuitively that all numbers less than 100 will be on Page 0, all numbers greater than or equal to 100 but less than 200 will be on Page 1, and so on. (That is the advantage of choosing a fixed page size, such as 100 bytes.) The operating system uses an algorithm to calculate the page and displacement; it is a simple arithmetic calculation.

To find the address of a given program instruction, the byte number is divided by the page size, keeping the remainder as an integer. The resulting quotient is the page number, and the remainder is the displacement within that page. When it is set up as a long division problem, it looks like this:

```
                        page number
page size ) byte number to be located
            xxx
              xxx
              xxx
                displacement
```

For example, if we use 100 bytes as the page size, the page number and the displacement (the location within that page) of byte 214 can be calculated using long division like this:

```
          2
100 ) 214
        200
         14
```

The quotient (2) is the page number, and the remainder (14) is the displacement. So the byte is located on Page 2, 15 lines (Line 14) from the top of the page.

Let's try another example with a more common page size of 256 bytes. Say we are seeking the location of byte 384. When we divide 384 by 256, the result is 1.5. Therefore, the byte is located at the midpoint on the second page (Page 1).

```
          1.5
256 ) 384
```

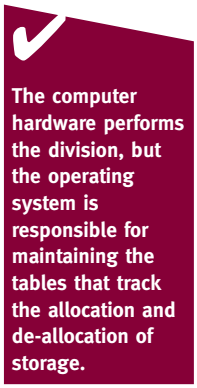
To find the line's exact location, multiply the page size (256) by the decimal (0.5) to discover that the line we're seeking is located on Line 129 of Page 1.

Using the concepts just presented, and using the same parameters from the first example, answer these questions:

1. Could the operating system (or the hardware) get a page number that is greater than 3 if the program was searching for byte 214?
2. If it did, what should the operating system do?
3. Could the operating system get a remainder of more than 99?
4. What is the smallest remainder possible?

Here are the answers:

1. No, not if the application program was written correctly.
2. Send an error message and stop processing the program (because the page is out of bounds).
3. No, not if it divides correctly.
4. Zero.



The computer hardware performs the division, but the operating system is responsible for maintaining the tables that track the allocation and de-allocation of storage.

This procedure gives the location of an instruction with respect to the job's pages. However, these pages are only relative; each page is actually stored in a page frame that can be located anywhere in available main memory. Therefore, the algorithm needs to be expanded to find the exact location of the byte in main memory. To do so, we need to correlate each of the job's pages with its page frame number using the Page Map Table.

For example, if we look at the PMT for Job 1 from Figure 3.1, we see that it looks like the data in Table 3.2.

(table 3.2)

Page Map Table for Job 1 in Figure 3.1.

Job Page Number	Page Frame Number
0	8
1	10
2	5
3	11

In the first division example, we were looking for an instruction with a displacement of 14 on Page 2. To find its exact location in memory, the operating system (or the hardware) has to perform the following four steps. (In actuality, the operating system identifies the lines, or data values and instructions, as addresses [bytes or words]. We refer to them here as lines to make it easier to explain.)

STEP 1 Do the arithmetic computation just described to determine the page number and displacement of the requested byte.

- Page number = the integer quotient from the division of the job space address by the page size
- Displacement = the remainder from the page number division

In this example, the computation shows that the page number is 2 and the displacement is 14.

STEP 2 Refer to this job's PMT (shown in Table 3.2) and find out which page frame contains Page 2. Page 2 is located in Page Frame 5.

STEP 3 Get the address of the beginning of the page frame by multiplying the page frame number (5) by the page frame size (100).

```
ADDR_PAGE_FRAME = PAGE_FRAME_NUM * PAGE_SIZE
ADDR_PAGE_FRAME = 5(100)
```

STEP 4 Now add the displacement (calculated in step 1) to the starting address of the page frame to compute the precise location in memory of the instruction:

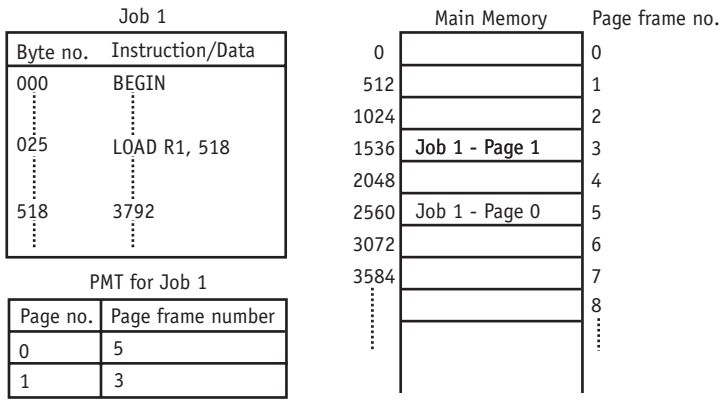
```
INSTR_ADDR_IN_MEM = ADDR_PAGE_FRAME + DISPL
INSTR_ADDR_IN_MEM = 500 + 14
```

The result of this maneuver tells us exactly where byte 14 is located in main memory.

Figure 3.3 shows another example and follows the hardware (and the operating system) as it runs an assembly language program that instructs the system to load into Register 1 the value found at byte 518.

In Figure 3.3, the page frame sizes in main memory are set at 512 bytes each and the page size is 512 bytes for this system. From the PMT we can see that this job has been divided into two pages. To find the exact location of byte 518 (where the system will find the value to load into Register 1), the system will do the following:

1. Compute the page number and displacement—the page number is 1, and the displacement is 6.
2. Go to the Page Map Table and retrieve the appropriate page frame number for Page 1. It is Page Frame 3.
3. Compute the starting address of the page frame by multiplying the page frame number by the page frame size: (3 * 512 = 1536).
4. Calculate the exact address of the instruction in main memory by adding the displacement to the starting address: (1536 + 6 = 1542). Therefore, memory address 1542 holds the value that should be loaded into Register 1.



(figure 3.3)

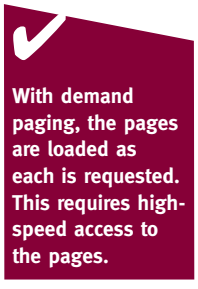
Job 1 with its Page Map Table. This snapshot of main memory shows the allocation of page frames to Job 1.

As you can see, this is a lengthy operation. Every time an instruction is executed, or a data value is used, the operating system (or the hardware) must translate the job space address, which is relative, into its physical address, which is absolute. This is called resolving the address, also called **address resolution**, or address translation. Of course, all of this processing is overhead, which takes processing capability away from the jobs waiting to be completed. However, in most systems the hardware does the paging, although the operating system is involved in dynamic paging, which will be covered later.

The advantage of a paging scheme is that it allows jobs to be allocated in noncontiguous memory locations so that memory is used more efficiently and more jobs can fit in the main memory (which is synonymous). However, there are disadvantages—overhead is increased and internal fragmentation is still a problem, although only in the last page of each job. The key to the success of this scheme is the size of the page. A page size that is too small will generate very long PMTs while a page size that is too large will result in excessive internal fragmentation. Determining the best page size is an important policy decision—there are no hard and fast rules that will guarantee optimal use of resources—and it is a problem we'll see again as we examine other paging alternatives. The best size depends on the actual job environment, the nature of the jobs being processed, and the constraints placed on the system.

Demand Paging

Demand paging introduced the concept of loading only a part of the program into memory for processing. It was the first widely used scheme that removed the restriction of having the entire job in memory from the beginning to the end of its processing. With demand paging, jobs are still divided into equally sized pages that initially reside in secondary storage. When the job begins to run, its pages are brought into memory only as they are needed.



With demand paging, the pages are loaded as each is requested. This requires high-speed access to the pages.

Demand paging takes advantage of the fact that programs are written sequentially so that while one section, or module, is processed all of the other modules are idle. Not all the pages are accessed at the same time, or even sequentially. For example:

- User-written error handling modules are processed only when a specific error is detected during execution. (For instance, they can be used to indicate to the operator that input data was incorrect or that a computation resulted in an invalid answer). If no error occurs, and we hope this is generally the case, these instructions are never processed and never need to be loaded into memory.

- Many modules are mutually exclusive. For example, if the input module is active (such as while a worksheet is being loaded) then the processing module is inactive. Similarly, if the processing module is active then the output module (such as printing) is idle.
- Certain program options are either mutually exclusive or not always accessible. This is easiest to visualize in menu-driven programs. For example, an application program may give the user several menu choices as shown in Figure 3.4. The system allows the operator to make only one selection at a time. If the user selects the first option then the module with the program instructions to move records to the file is the only one that is being used, so that is the only module that needs to be in memory at this time. The other modules all remain in secondary storage until they are called from the menu.
- Many tables are assigned a large fixed amount of address space even though only a fraction of the table is actually used. For example, a symbol table for an assembler might be prepared to handle 100 symbols. If only 10 symbols are used then 90 percent of the table remains unused.

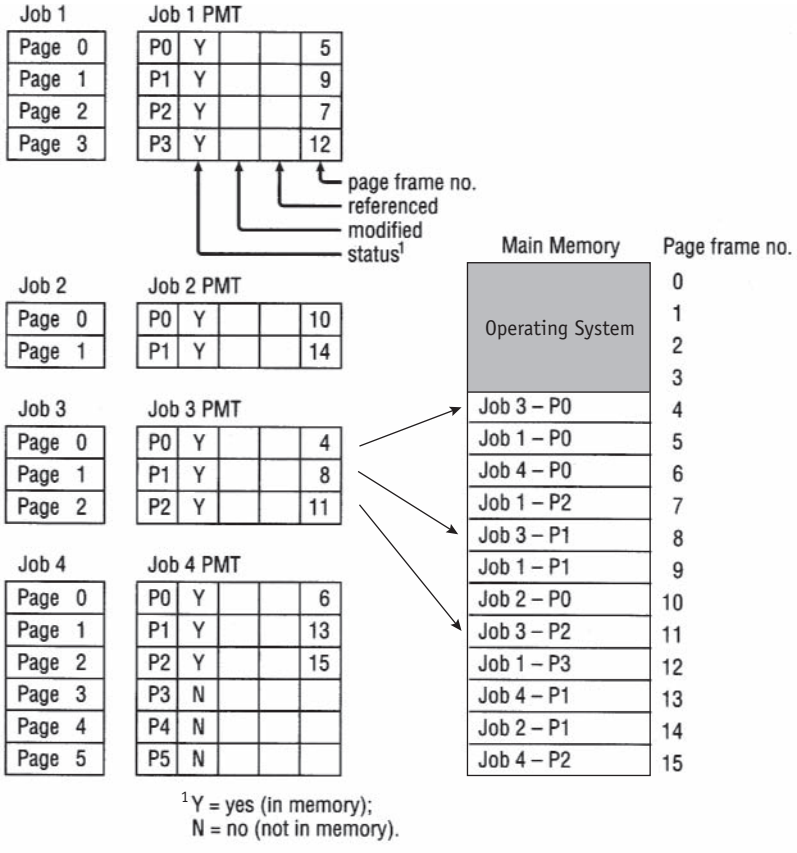
(figure 3.4)

When you choose one option from the menu of an application program such as this one, the other modules that aren't currently required (such as Help) don't need to be moved into memory immediately.

One of the most important innovations of demand paging was that it made virtual memory feasible. (Virtual memory will be discussed later in this chapter.) The demand paging scheme allows the user to run jobs with less main memory than is required if the operating system is using the paged memory allocation scheme described earlier. In fact, a demand paging scheme can give the appearance of an almost-infinite or nonfinite amount of physical memory when, in reality, physical memory is significantly less than infinite.

The key to the successful implementation of this scheme is the use of a high-speed direct access storage device (such as hard drives or flash memory) that can work directly with the CPU. That is vital because pages must be passed quickly from secondary storage to main memory and back again.

How and when the pages are passed (also called swapped) depends on predefined policies that determine when to make room for needed pages and how to do so. The operating system relies on tables (such as the Job Table, the Page Map Table, and the Memory Map Table) to implement the algorithm. These tables are basically the same as for paged memory allocation but with the addition of three new fields for each page



(figure 3.5)

Demand paging requires that the Page Map Table for each job keep track of each page as it is loaded or removed from main memory. Each PMT tracks the status of the page, whether it has been modified, whether it has been recently referenced, and the page frame number for each page currently in main memory. (Note: For this illustration, the Page Map Tables have been simplified. See Table 3.3 for more detail.)

in the PMT: one to determine if the page being requested is already in memory; a second to determine if the page contents have been modified; and a third to determine if the page has been referenced recently, as shown at the top of Figure 3.5.

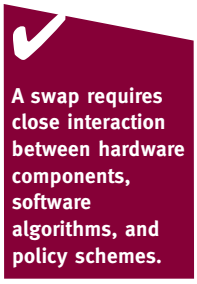
The first field tells the system where to find each page. If it is already in memory, the system will be spared the time required to bring it from secondary storage. It is faster for the operating system to scan a table located in main memory than it is to retrieve a page from a disk.

The second field, noting if the page has been modified, is used to save time when pages are removed from main memory and returned to secondary storage. If the contents of the page haven't been modified then the page doesn't need to be rewritten to secondary storage. The original, already there, is correct.

The third field, which indicates any recent activity, is used to determine which pages show the most processing activity, and which are relatively inactive. This information is used by several page-swapping policy schemes to determine which pages should

remain in main memory and which should be swapped out when the system needs to make room for other pages being requested.

For example, in Figure 3.5 the number of total job pages is 15, and the number of total available page frames is 12. (The operating system occupies the first four of the 16 page frames in main memory.)



A swap requires close interaction between hardware components, software algorithms, and policy schemes.

Assuming the processing status illustrated in Figure 3.5, what happens when Job 4 requests that Page 3 be brought into memory if there are no empty page frames available?

To move in a new page, a resident page must be swapped back into secondary storage. Specifically, that includes copying the resident page to the disk (if it was modified), and writing the new page into the empty page frame.

The hardware components generate the address of the required page, find the page number, and determine whether it is already in memory. The following algorithm makes up the hardware instruction processing cycle.

Hardware Instruction Processing Algorithm

```
1  Start processing instruction
2  Generate data address
3  Compute page number
4  If page is in memory
            Then
            get data and finish instruction
            advance to next instruction
            return to step 1
   Else
            generate page interrupt
            call page fault handler
   End if
```

The same process is followed when fetching an instruction.

When the test fails (meaning that the page is in secondary storage but not in memory), the operating system software takes over. The section of the operating system that resolves these problems is called the **page fault handler**. It determines whether there are empty page frames in memory so the requested page can be immediately copied from secondary storage. If all page frames are busy, the page fault handler must decide

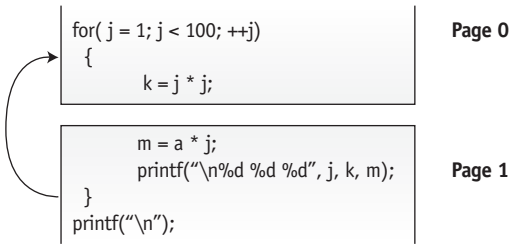
which page will be swapped out. (This decision is directly dependent on the predefined policy for page removal.) Then the swap is made.

Page Fault Handler Algorithm

```
1  If there is no free page frame
       Then
       select page to be swapped out using page removal algorithm
       update job's Page Map Table
       If content of page had been changed then
           write page to disk
       End if
   End if
2  Use page number from step 3 from the Hardware Instruction
   Processing Algorithm to get disk address where the requested page is
   stored (the File Manager, to be discussed in Chapter 8, uses the page
   number to get the disk address)
3  Read page into memory
4  Update job's Page Map Table
5  Update Memory Map Table
6  Restart interrupted instruction
```

Before continuing, three tables must be updated: the Page Map Tables for both jobs (the PMT with the page that was swapped out and the PMT with the page that was swapped in) and the Memory Map Table. Finally, the instruction that was interrupted is resumed and processing continues.

Although demand paging is a solution to inefficient memory utilization, it is not free of problems. When there is an excessive amount of **page swapping** between main memory and secondary storage, the operation becomes inefficient. This phenomenon is called **thrashing**. It uses a great deal of the computer's energy but accomplishes very little, and it is caused when a page is removed from memory but is called back shortly thereafter. Thrashing can occur across jobs, when a large number of jobs are vying for a relatively low number of free pages (the ratio of job pages to free memory page frames is high), or it can happen within a job—for example, in loops that cross page boundaries. We can demonstrate this with a simple example. Suppose the beginning of a loop falls at the bottom of a page and is completed at the top of the next page, as in the C program in Figure 3.6.



(figure 3.6)

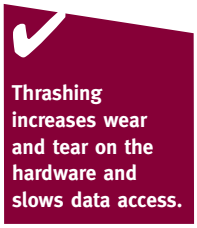
An example of demand paging that causes a page swap each time the loop is executed and results in thrashing. If only a single page frame is available, this program will have one page fault each time the loop is executed.

The situation in Figure 3.6 assumes there is only one empty page frame available. The first page is loaded into memory and execution begins, but after executing the last command on Page 0, the page is swapped out to make room for Page 1. Now execution can continue with the first command on Page 1, but at the "}" symbol Page 1 must be swapped out so Page 0 can be brought back in to continue the loop. Before this program is completed, swapping will have occurred 100 times (unless another page frame becomes free so both pages can reside in memory at the same time). A failure to find a page in memory is often called a **page fault** and this example would generate 100 page faults (and swaps).

In such extreme cases, the rate of useful computation could be degraded by a factor of 100. Ideally, a demand paging scheme is most efficient when programmers are aware of the page size used by their operating system and are careful to design their programs to keep page faults to a minimum; but in reality, this is not often feasible.

Page Replacement Policies and Concepts

As we just learned, the policy that selects the page to be removed, the **page replacement policy**, is crucial to the efficiency of the system, and the algorithm to do that must be carefully selected.



Several such algorithms exist and it is a subject that enjoys a great deal of theoretical attention and research. Two of the most well-known are first-in first-out and least recently used. The **first-in first-out (FIFO) policy** is based on the theory that the best page to remove is the one that has been in memory the longest. The **least recently used (LRU) policy** chooses the page least recently accessed to be swapped out.

To illustrate the difference between FIFO and LRU, let us imagine a dresser drawer filled with your favorite sweaters. The drawer is full, but that didn't stop you from buying a new sweater. Now you have to put it away. Obviously it won't fit in your

sweater drawer unless you take something out, but which sweater should you move to the storage closet? Your decision will be based on a sweater removal policy.

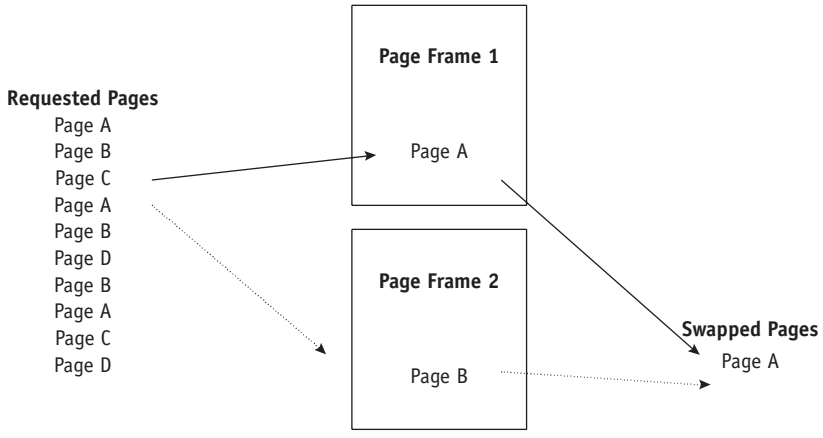
You could take out your oldest sweater (the one that was first in), figuring that you probably won't use it again—hoping you won't discover in the following days that it is your most used, most treasured possession. Or, you could remove the sweater that you haven't worn recently and has been idle for the longest amount of time (the one that was least recently used). It is readily identifiable because it is at the bottom of the drawer. But just because it hasn't been used recently doesn't mean that a once-a-year occasion won't demand its appearance soon.

What guarantee do you have that once you have made your choice you won't be trekking to the storage closet to retrieve the sweater you stored yesterday? You could become a victim of thrashing.

Which is the best policy? It depends on the weather, the wearer, and the wardrobe. Of course, one option is to get another drawer. For an operating system (or a computer), this is the equivalent of adding more accessible memory, and we will explore that option after we discover how to more effectively use the memory we already have.

First-In First-Out

The first-in first-out (FIFO) page replacement policy will remove the pages that have been in memory the longest. The process of swapping pages is illustrated in Figure 3.7.



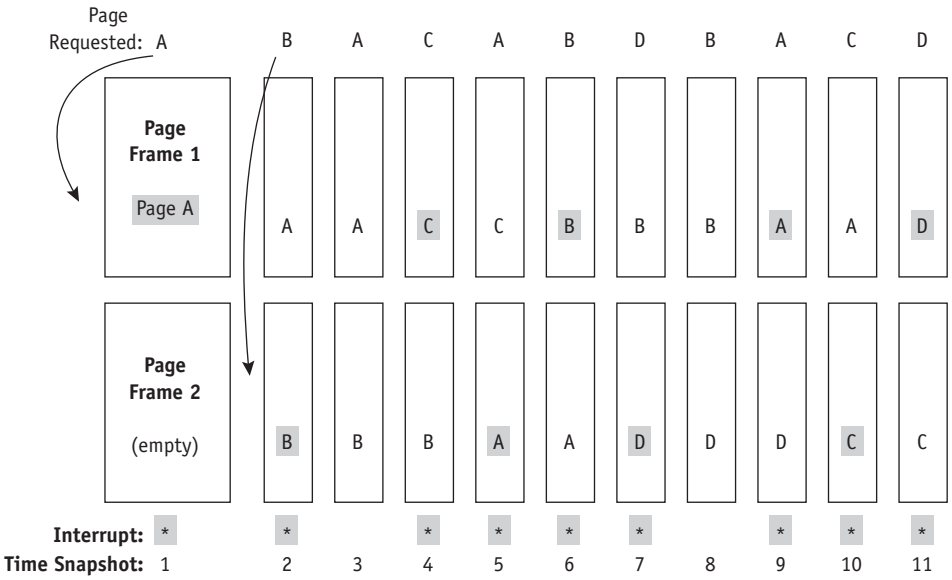
(figure 3.7)

The FIFO policy in action with only two page frames available. When the program calls for Page C, Page A must be moved out of the first page frame to make room for it, as shown by the solid lines. When Page A is needed again, it will replace Page B in the second page frame, as shown by the dotted lines. The entire sequence is shown in Figure 3.8.

Figure 3.8 shows how the FIFO algorithm works by following a job with four pages (A, B, C, D) as it is processed by a system with only two available page frames. Figure 3.8 displays how each page is swapped into and out of memory and marks each interrupt with an asterisk. We then count the number of page interrupts and compute the failure rate and the success rate. The job to be processed needs its pages in the following order: A, B, A, C, A, B, D, B, A, C, D.

When both page frames are occupied, each new page brought into memory will cause an existing one to be swapped out to secondary storage. A page interrupt, which we identify with an asterisk (*), is generated when a new page needs to be loaded into memory, whether a page is swapped out or not.

The efficiency of this configuration is dismal—there are 9 page interrupts out of 11 page requests due to the limited number of page frames available and the need for many new pages. To calculate the failure rate, we divide the number of interrupts by the number of page requests. The failure rate of this system is 9/11, which is 82 percent. Stated another way, the success rate is 2/11, or 18 percent. A failure rate this high is usually unacceptable.



(figure 3.8)

Using a FIFO policy, this page trace analysis shows how each page requested is swapped into the two available page frames. When the program is ready to be processed, all four pages are in secondary storage. When the program calls a page that isn't already in memory, a page interrupt is issued, as shown by the gray boxes and asterisks. This program resulted in nine page interrupts.

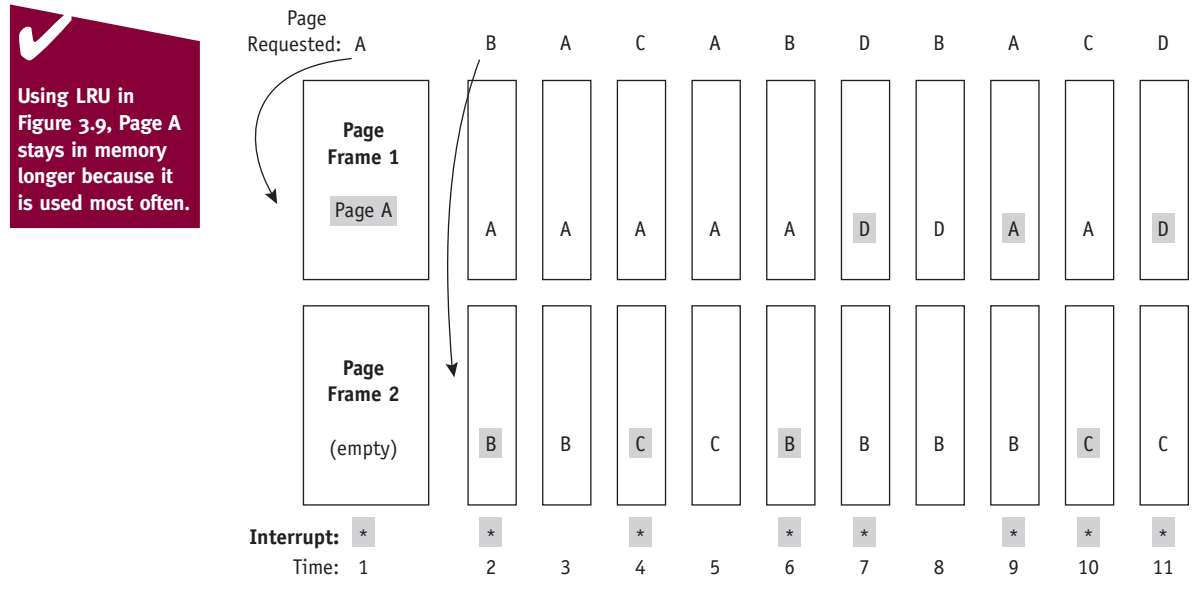
In Figure 3.8, using FIFO, Page A is swapped out when a newer page arrives even though it is used the most often.

We are not saying FIFO is bad. We chose this example to show how FIFO works, not to diminish its appeal as a swapping policy. The high failure rate here is caused by both the limited amount of memory available and the order in which pages are requested by the program. The page order can't be changed by the system, although the size of main memory can be changed; but buying more memory may not always be the best solution—especially when you have many users and each one wants an unlimited amount of memory. There is no guarantee that buying more memory will always result in better performance; this is known as the **FIFO anomaly**, which is explained later in this chapter.

Least Recently Used

The least recently used (LRU) page replacement policy swaps out the pages that show the least amount of recent activity, figuring that these pages are the least likely to be used again in the immediate future. Conversely, if a page is used, it is likely to be used again soon; this is based on the theory of locality, which will be explained later in this chapter.

To see how it works, let us follow the same job in Figure 3.8 but using the LRU policy. The results are shown in Figure 3.9. To implement this policy, a queue of the requests is kept in FIFO order, a time stamp of when the job entered the system is saved, or a mark in the job's PMT is made periodically.



Using LRU in Figure 3.9, Page A stays in memory longer because it is used most often.

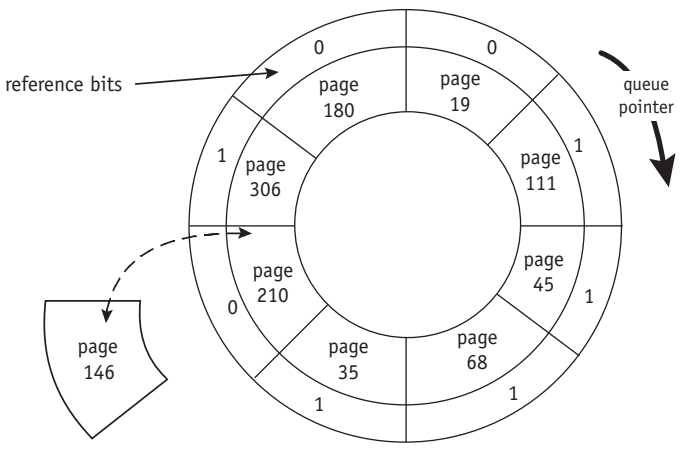
(figure 3.9)

Memory management using an LRU page removal policy for the program shown in Figure 3.8. Throughout the program, 11 page requests are issued, but they cause only 8 page interrupts.

The efficiency of this configuration is only slightly better than with FIFO. Here, there are 8 page interrupts out of 11 page requests, so the failure rate is 8/11, or 73 percent. In this example, an increase in main memory by one page frame would increase the success rate of both FIFO and LRU. However, we can't conclude on the basis of only one example that one policy is better than the others. In fact, LRU is a stack algorithm removal policy, which means that an increase in memory will never cause an increase in the number of page interrupts.

On the other hand, it has been shown that under certain circumstances adding more memory can, in rare cases, actually cause an increase in page interrupts when using a FIFO policy. As noted before, it is called the FIFO anomaly. But although it is an unusual occurrence, the fact that it exists coupled with the fact that pages are removed regardless of their activity (as was the case in Figure 3.8) has removed FIFO from the most favored policy position it held in some cases.

A variation of the LRU page replacement algorithm is known as the **clock page replacement policy** because it is implemented with a circular queue and uses a pointer to step through the reference bits of the active pages, simulating a clockwise motion. The algorithm is paced according to the computer's **clock cycle**, which is the time span between two ticks in its system clock. The algorithm checks the reference bit for each page. If the bit is one (indicating that it was recently referenced), the bit is reset to zero and the bit for the next page is checked. However, if the reference bit is zero (indicating that the page has not recently been referenced), that page is targeted for removal. If all the reference bits are set to one, then the pointer must cycle through the entire circular queue again giving each page a second and perhaps a third or fourth chance. Figure 3.10 shows a circular queue containing the reference bits for eight pages currently in memory. The pointer indicates the page that would be considered next for removal. Figure 3.10 shows what happens to the reference bits of the pages that have



(figure 3.10)

A circular queue, which contains the page number and its reference bit. The pointer seeks the next candidate for removal and replaces page 210 with a new page, 146.

been given a second chance. When a new page, 146, has to be allocated to a page frame, it is assigned to the space that has a reference bit of zero, the space previously occupied by page 210.

A second variation of LRU uses an 8-bit reference byte and a bit-shifting technique to track the usage of each page currently in memory. When the page is first copied into memory, the leftmost bit of its reference byte is set to 1; and all bits to the right of the one are set to zero, as shown in Figure 3.11. At specific time intervals of the clock cycle, the Memory Manager shifts every page's reference bytes to the right by one bit, dropping their rightmost bit. Meanwhile, each time a page is referenced, the leftmost bit of its reference byte is set to 1.

This process of shifting bits to the right and resetting the leftmost bit to 1 when a page is referenced gives a history of each page's usage. For example, a page that has not been used for the last eight time ticks would have a reference byte of 00000000, while one that has been referenced once every time tick will have a reference byte of 11111111.

When a page fault occurs, the LRU policy selects the page with the smallest value in its reference byte because that would be the one least recently used. Figure 3.11 shows how the reference bytes for six active pages change during four snapshots of usage. In (a), the six pages have been initialized; this indicates that all of them have been referenced once. In (b), pages 1, 3, 5, and 6 have been referenced again (marked with 1), but pages 2 and 4 have not (now marked with 0 in the leftmost position). In (c), pages 1, 2, and 4 have been referenced. In (d), pages 1, 2, 4, and 6 have been referenced. In (e), pages 1 and 4 have been referenced.

As shown in Figure 3.11, the values stored in the reference bytes are not unique: page 3 and page 5 have the same value. In this case, the LRU policy may opt to swap out all of the pages with the smallest value, or may select one among them based on other criteria such as FIFO, priority, or whether the contents of the page have been modified.

Other page removal algorithms, MRU (most recently used) and LFU (least frequently used), are discussed in exercises at the end of this chapter.

(figure 3.11)

Notice how the reference bit for each page is updated with every time tick. Arrows (a) through (e) show how the initial bit shifts to the right with every tick of the clock.

Page Number	Time Snapshot 0	Time Snapshot 1	Time Snapshot 2	Time Snapshot 3	Time Snapshot 4
1	10000000	11000000	11100000	11110000	11111000
2	10000000	01000000	10100000	11010000	01101000
3	10000000	11000000	01100000	00110000	00011000
4	10000000	01000000	10100000	11010000	11101000
5	10000000	11000000	01100000	00110000	00011000
6	10000000	11000000	01100000	10110000	01011000
	(a)	(b)	(c)	(d)	(e)

The Mechanics of Paging

Before the Memory Manager can determine which pages will be swapped out, it needs specific information about each page in memory—information included in the Page Map Tables.

Each PMT must track each page's status, modifications, and references. It does so with three bits, each of which can be either 0 or 1.

For example, in Figure 3.5, the Page Map Table for Job 1 included three bits: the status bit, the referenced bit, and the modified bit (these were the three middle columns: the two empty columns and the Y/N column representing "in memory"). But the representation of the table shown in Figure 3.5 was simplified for illustration purposes. It actually looks something like the one shown in Table 3.3.

Page	Status Bit	Referenced Bit	Modified Bit	Page Frame
0	1	1	1	5
1	1	0	0	9
2	1	0	0	7
3	1	1	0	12

(table 3.3)

Page Map Table for Job 1 shown in Figure 3.5.

As we said before, the status bit indicates whether the page is currently in memory. The referenced bit indicates whether the page has been called (referenced) recently. This bit is important because it is used by the LRU algorithm to determine which pages should be swapped out.

The modified bit indicates whether the contents of the page have been altered and, if so, the page must be rewritten to secondary storage when it is swapped out before its page frame is released. (A page frame with contents that have not been modified can be overwritten directly, thereby saving a step.) That is because when a page is swapped into memory it isn't removed from secondary storage. The page is merely copied—the original remains intact in secondary storage. Therefore, if the page isn't altered while it is in main memory (in which case the modified bit remains unchanged, zero), the page needn't be copied back to secondary storage when it is swapped out of memory—the page that is already there is correct. However, if modifications were made to the page, the new version of the page must be written over the older version—and that takes time.

Each bit can be either 0 or 1 as shown in Table 3.4.

(table 3.4) *The meaning of the bits used in the Page Map Table.*

Status Bit		Modified Bit		Referenced Bit	
Value	**Meaning**	**Value**	**Meaning**	**Value**	**Meaning**
0	not in memory	0	not modified	0	not called
1	resides in memory	1	was modified	1	was called

The status bit for all pages in memory is 1. A page must be in memory before it can be swapped out so all of the candidates for swapping have a 1 in this column. The other two bits can be either 0 or 1, so there are four possible combinations of the referenced and modified bits as shown in Table 3.5.

(table 3.5) *Four possible combinations of modified and referenced bits and the meaning of each.*

	Modified	Referenced	Meaning
Case 1	0	0	Not modified AND not referenced
Case 2	0	1	Not modified BUT was referenced
Case 3	1	0	Was modified BUT not referenced [impossible?]
Case 4	1	1	Was modified AND was referenced

The FIFO algorithm uses only the modified and status bits when swapping pages, but the LRU looks at all three before deciding which pages to swap out.

Which page would the LRU policy choose first to swap? Of the four cases described in Table 3.5, it would choose pages in Case 1 as the ideal candidates for removal because they've been neither modified nor referenced. That means they wouldn't need to be rewritten to secondary storage, and they haven't been referenced recently. So the pages with zeros for these two bits would be the first to be swapped out.

What is the next most likely candidate? The LRU policy would choose Case 3 next because the other two, Case 2 and Case 4, were recently referenced. The bad news is that Case 3 pages have been modified, so it will take more time to swap them out. By process of elimination, then we can say that Case 2 is the third choice and Case 4 would be the pages least likely to be removed.

You may have noticed that Case 3 presents an interesting situation: apparently these pages have been modified without being referenced. How is that possible? The key lies in how the referenced bit is manipulated by the operating system. When the pages are brought into memory, they are all usually referenced at least once and that means that all of the pages soon have a referenced bit of 1. Of course the LRU algorithm would be defeated if every page indicated that it had been referenced. Therefore, to make sure the referenced bit actually indicates *recently* referenced, the operating system periodically resets it to 0. Then, as the pages are referenced during processing, the bit is changed from 0 to 1 and the LRU policy is able to identify which pages actually are frequently referenced. As you can imagine, there is one brief instant, just after the bits are reset, in which all of the pages (even the active pages) have reference bits of 0 and are vulnerable. But as processing continues, the most-referenced pages soon have their bits reset to 1, so the risk is minimized.

The Working Set

One innovation that improved the performance of demand paging schemes was the concept of the **working set**. A job's working set is the set of pages residing in memory that can be accessed directly without incurring a page fault.

When a user requests execution of a program, the first page is loaded into memory and execution continues as more pages are loaded: those containing variable declarations, others containing instructions, others containing data, and so on. After a while, most programs reach a fairly stable state and processing continues smoothly with very few additional page faults. At this point the job's working set is in memory, and the program won't generate many page faults until it gets to another phase requiring a different set of pages to do the work—a different working set.

Of course, it is possible that a poorly structured program could require that every one of its pages be in memory before processing can begin.

Fortunately, most programs are structured, and this leads to a **locality of reference** during the program's execution, meaning that during any phase of its execution the program references only a small fraction of its pages. For example, if a job is executing a loop then the instructions within the loop are referenced extensively while those outside the loop aren't used at all until the loop is completed—that is locality of reference. The same applies to sequential instructions, subroutine calls (within the subroutine), stack implementations, access to variables acting as counters or sums, or multidimensional variables such as arrays and tables (only a few of the pages are needed to handle the references).

It would be convenient if all of the pages in a job's working set were loaded into memory at one time to minimize the number of page faults and to speed up processing, but that is easier said than done. To do so, the system needs definitive answers to some

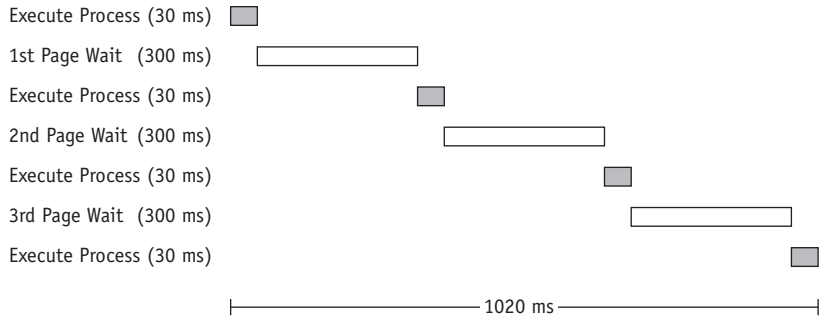
difficult questions: How many pages comprise the working set? What is the maximum number of pages the operating system will allow for a working set?

The second question is particularly important in networked or time-sharing systems, which regularly swap jobs (or pages of jobs) into memory and back to secondary storage to accommodate the needs of many users. The problem is this: every time a job is reloaded back into memory (or has pages swapped), it has to generate several page faults until its working set is back in memory and processing can continue. It is a time-consuming task for the CPU, which can't be processing jobs during the time it takes to process each page fault, as shown in Figure 3.12.

One solution adopted by many paging systems is to begin by identifying each job's working set and then loading it into memory in its entirety before allowing execution to begin. This is difficult to do before a job is executed but can be identified as its execution proceeds.

In a time-sharing or networked system, this means the operating system must keep track of the size and identity of every working set, making sure that the jobs destined for processing at any one time won't exceed the available memory. Some operating systems use a variable working set size and either increase it when necessary (the job requires more processing) or decrease it when necessary. This may mean that the number of jobs in memory will need to be reduced if, by doing so, the system can ensure the completion of each job and the subsequent release of its memory space.

We have looked at several examples of demand paging memory allocation schemes. Demand paging had two advantages. It was the first scheme in which a job was no



(figure 3.12)

Time line showing the amount of time required to process page faults for a single program. The program in this example takes 120 milliseconds (ms) to execute but an additional 900 ms to load the necessary pages into memory. Therefore, job turnaround is 1020 ms.

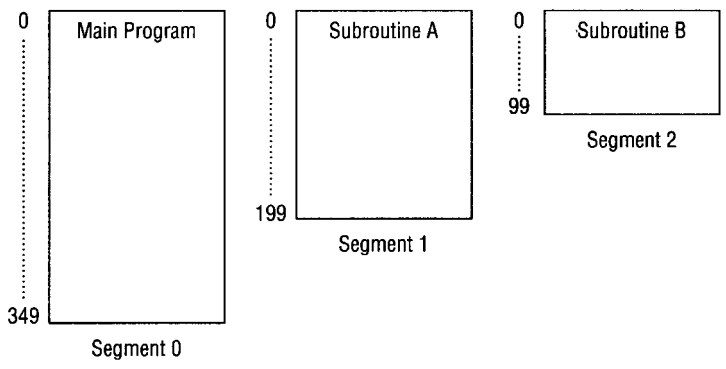
longer constrained by the size of physical memory and it introduced the concept of virtual memory. The second advantage was that it utilized memory more efficiently than the previous schemes because the sections of a job that were used seldom or not at all (such as error routines) weren't loaded into memory unless they were specifically requested. Its disadvantage was the increased overhead caused by the tables and the page interrupts. The next allocation scheme built on the advantages of both paging and dynamic partitions.

Segmented Memory Allocation

The concept of segmentation is based on the common practice by programmers of structuring their programs in modules—logical groupings of code. With **segmented memory allocation**, each job is divided into several **segments** of different sizes, one for each module that contains pieces that perform related functions. Segmented memory allocation was designed to reduce page faults that resulted from having a segment's loop split over two or more pages. A **subroutine** is an example of one such logical group. This is fundamentally different from a paging scheme, which divides the job into several pages all of the same size, each of which often contains pieces from more than one program module.

A second important difference is that main memory is no longer divided into page frames because the size of each segment is different—some are large and some are small. Therefore, as with the dynamic partitions discussed in Chapter 2, memory is allocated in a dynamic manner.

When a program is compiled or assembled, the segments are set up according to the program's structural modules. Each segment is numbered and a **Segment Map Table (SMT)** is generated for each job; it contains the segment numbers, their lengths, access rights, status, and (when each is loaded into memory) its location in memory. Figures 3.13 and 3.14 show the same job, Job 1, composed of a main program and two subroutines, together with its Segment Map Table and actual main memory allocation.



(figure 3.13)

Segmented memory allocation. Job 1 includes a main program, Subroutine A, and Subroutine B. It is one job divided into three segments.

The Segment Map Table functions the same way as a Page Map Table but manages segments instead of pages.

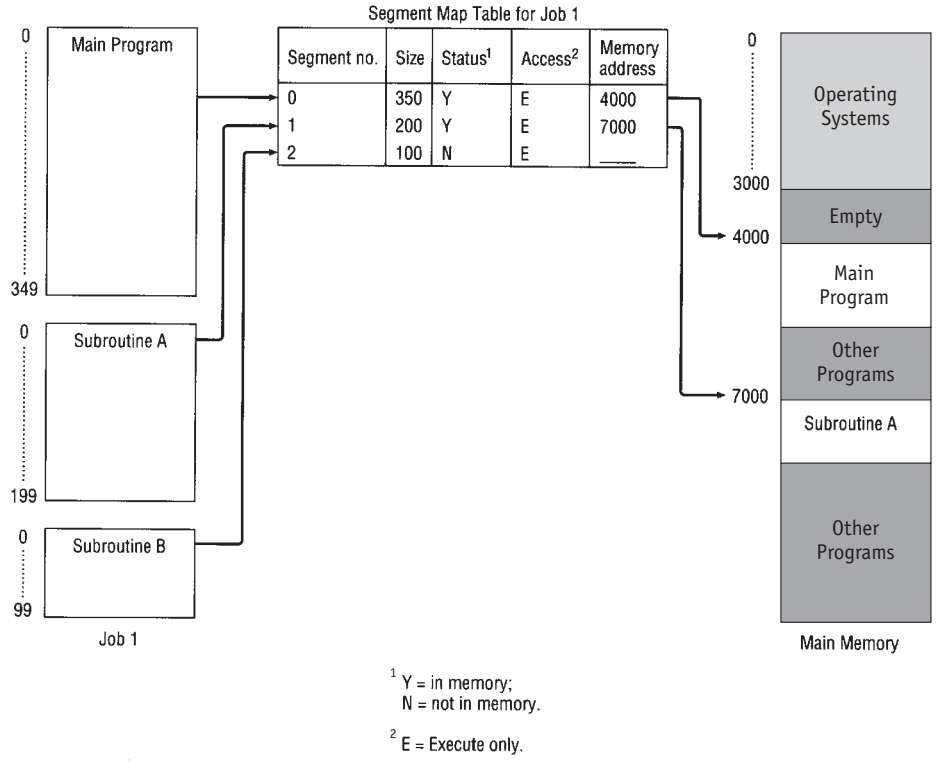
As in demand paging, the referenced, modified, and status bits are used in segmentation and appear in the SMT but they aren't shown in Figures 3.13 and 3.14.

The Memory Manager needs to keep track of the segments in memory. This is done with three tables combining aspects of both dynamic partitions and demand paging memory management:

- The Job Table lists every job being processed (one for the whole system).
- The Segment Map Table lists details about each segment (one for each job).
- The Memory Map Table monitors the allocation of main memory (one for the whole system).

Like demand paging, the instructions within each segment are ordered sequentially, but the segments don't need to be stored contiguously in memory. We only need to know where each segment is stored. The contents of the segments themselves are contiguous in this scheme.

To access a specific location within a segment, we can perform an operation similar to the one used for paged memory management. The only difference is that we work with



(figure 3.14)

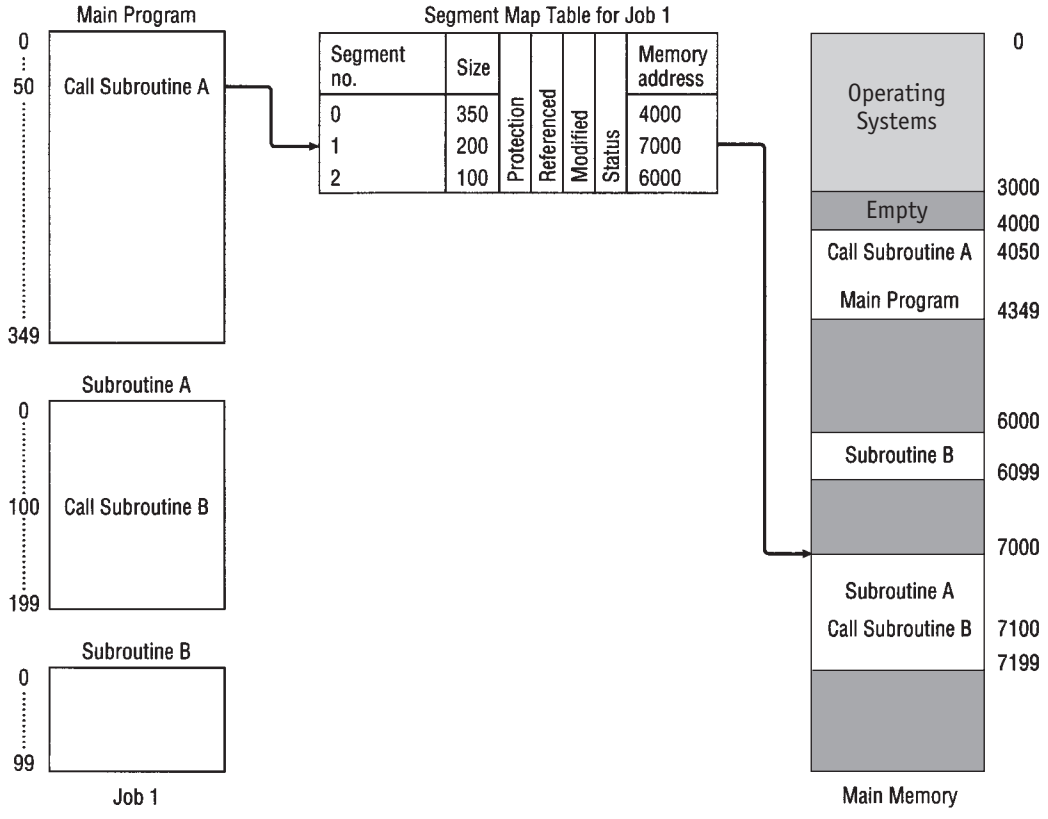
The Segment Map Table tracks each segment for Job 1.

segments instead of pages. The addressing scheme requires the segment number and the displacement within that segment; and because the segments are of different sizes, the displacement must be verified to make sure it isn't outside of the segment's range.

In Figure 3.15, Segment 1 includes all of Subroutine A so the system finds the beginning address of Segment 1, address 7000, and it begins there.

If the instruction requested that processing begin at byte 100 of Subroutine A (which is possible in languages that support multiple entries into subroutines) then, to locate that item in memory, the Memory Manager would need to add 100 (the displacement) to 7000 (the beginning address of Segment 1). Its code could look like this:

```
ACTUAL_MEM_LOC = BEGIN_MEM_LOC + DISPLACEMENT
```



(figure 3.15)

During execution, the main program calls Subroutine A, which triggers the SMT to look up its location in memory.

Can the displacement be larger than the size of the segment? No, not if the program is coded correctly; however, accidents do happen and the Memory Manager must always guard against this possibility by checking the displacement against the size of the segment, verifying that it is not out of bounds.

To access a location in memory, when using either paged or segmented memory management, the address is composed of two values: the page or segment number and the displacement. Therefore, it is a two-dimensional addressing scheme:

```
SEGMENT_NUMBER & DISPLACEMENT
```

The disadvantage of any allocation scheme in which memory is partitioned dynamically is the return of external fragmentation. Therefore, recompaction of available memory is necessary from time to time (if that schema is used).

As you can see, there are many similarities between paging and segmentation, so they are often confused. The major difference is a conceptual one: pages are physical units that are invisible to the user's program and consist of fixed sizes; segments are logical units that are visible to the user's program and consist of variable sizes.

Segmented/Demand Paged Memory Allocation

The **segmented/demand paged memory allocation** scheme evolved from the two we have just discussed. It is a combination of segmentation and demand paging, and it offers the logical benefits of segmentation, as well as the physical benefits of paging. The logic isn't new. The algorithms used by the demand paging and segmented memory management schemes are applied here with only minor modifications.

This allocation scheme doesn't keep each segment as a single contiguous unit but subdivides it into pages of equal size, smaller than most segments, and more easily manipulated than whole segments. Therefore, many of the problems of segmentation (compaction, external fragmentation, and secondary storage handling) are removed because the pages are of fixed length.

This scheme, illustrated in Figure 3.16, requires four tables:

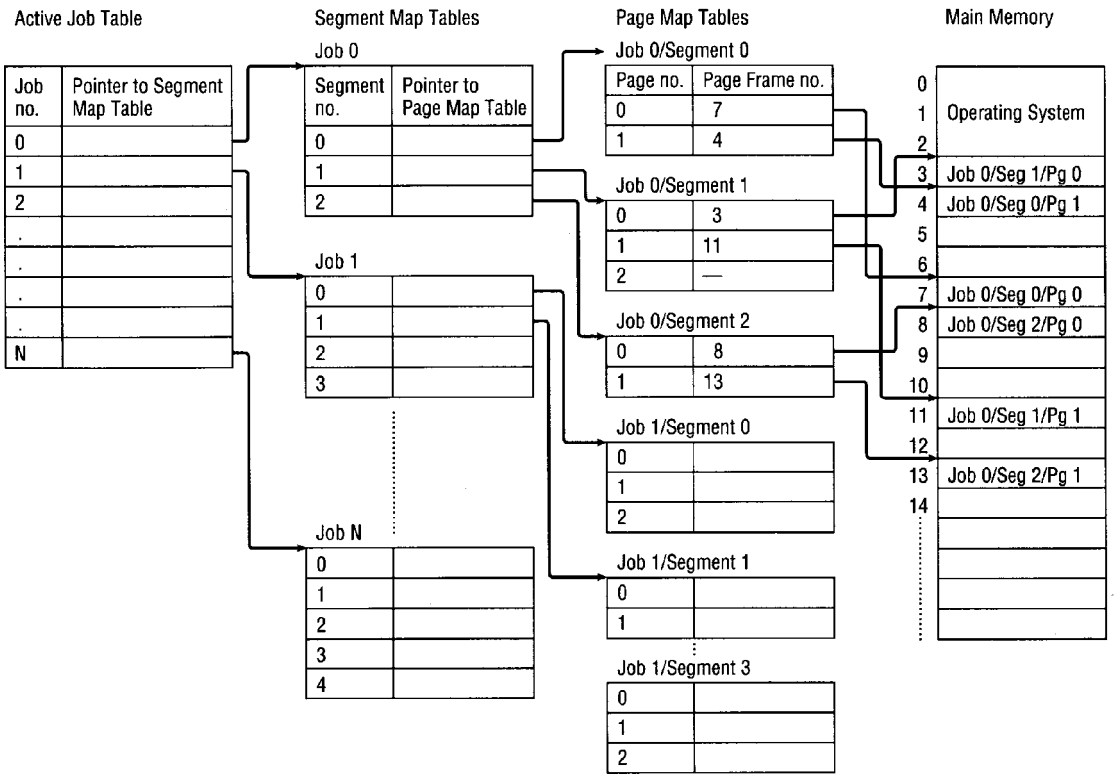
- The Job Table lists every job in process (one for the whole system).
- The Segment Map Table lists details about each segment (one for each job).
- The Page Map Table lists details about every page (one for each segment).
- The Memory Map Table monitors the allocation of the page frames in main memory (one for the whole system).

Note that the tables in Figure 3.16 have been simplified. The SMT actually includes additional information regarding protection (such as the authority to read, write, execute, and delete parts of the file), as well as which users have access to that segment (user only, group only, or everyone—some systems call these access categories owner, group, and world, respectively). In addition, the PMT includes the status, modified, and referenced bits.

To access a location in memory, the system must locate the address, which is composed of three entries: segment number, page number within that segment, and displacement within that page. It is a three-dimensional addressing scheme:

```
SEGMENT_NUMBER & PAGE_NUMBER & DISPLACEMENT
```

The major disadvantages of this memory allocation scheme are the overhead required for the extra tables and the time required to reference the segment table and the



(figure 3.16)

How the Job Table, Segment Map Table, Page Map Table, and main memory interact in a segment/paging scheme.

page table. To minimize the number of references, many systems use associative memory to speed up the process.

Associative memory is a name given to several registers that are allocated to each job that is active. Their task is to associate several segment and page numbers belonging to the job being processed with their main memory addresses. These associative registers reside in main memory, and the exact number of registers varies from system to system.

To appreciate the role of associative memory, it is important to understand how the system works with segments and pages. In general, when a job is allocated to the CPU, its Segment Map Table is loaded into main memory while the Page Map Tables are loaded only as needed. As pages are swapped between main memory and secondary storage, all tables are updated.

Here is a typical procedure: when a page is first requested, the job's SMT is searched to locate its PMT; then the PMT is loaded and searched to determine the page's location in memory. If the page isn't in memory, then a page interrupt is issued, the page is brought into memory, and the table is updated. (As the example indicates, loading the PMT can cause a page interrupt, or fault, as well.) This process is just as tedious as it sounds, but it gets easier. Since this segment's PMT (or part of it) now resides in memory, any other requests for pages within this segment can be quickly accommodated because there is no need to bring the PMT into memory. However, accessing these tables (SMT and PMT) is time-consuming.

The two searches through associative memory and segment/page map tables take place at the same time.

That is the problem addressed by associative memory, which stores the information related to the most-recently-used pages. Then when a page request is issued, two searches begin—one through the segment and page tables and one through the contents of the associative registers.

If the search of the associative registers is successful, then the search through the tables is stopped (or eliminated) and the address translation is performed using the information in the associative registers. However, if the search of associative memory fails, no time is lost because the search through the SMTs and PMTs had already begun (in this schema). When this search is successful and the main memory address from the PMT has been determined, the address is used to continue execution of the program and the reference is also stored in one of the associative registers. If all of the associative registers are full, then an LRU (or other) algorithm is used and the least-recently-referenced associative register is used to hold the information on this requested page.

For example, a system with eight associative registers per job will use them to store the SMT and PMT for the last eight pages referenced by that job. When an address needs to be translated from segment and page numbers to a memory location, the system will

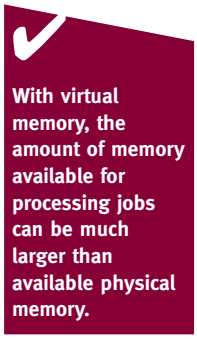
look first in the eight associative registers. If a match is found, the memory location is taken from the associative register; if there is no match, then the SMTs and PMTs will continue to be searched and the new information will be stored in one of the eight registers as a result.

If a job is swapped out to secondary storage during its execution, then all of the information stored in its associative registers is saved, as well as the current PMT and SMT, so the displaced job can be resumed quickly when the CPU is reallocated to it. The primary advantage of a large associative memory is increased speed. The disadvantage is the high cost of the complex hardware required to perform the parallel searches. In some systems the searches do not run in parallel, but the search of the SMT and PMT follows the search of the associative registers.

Virtual Memory

Demand paging made it possible for a program to execute even though only a part of a program was loaded into main memory. In effect, virtual memory removed the restriction imposed on maximum program size. This capability of moving pages at will between main memory and secondary storage gave way to a new concept appropriately named **virtual memory**. Even though only a portion of each program is stored in memory, it gives users the appearance that their programs are being completely loaded in main memory during their entire processing time—a feat that would require an incredible amount of main memory.

Until the implementation of virtual memory, the problem of making programs fit into available memory was left to the users. In the early days, programmers had to limit the size of their programs to make sure they fit into main memory; but sometimes that wasn't possible because the amount of memory allocated to them was too small to get the job done. Clever programmers solved the problem by writing tight programs wherever possible. It was the size of the program that counted most—and the instructions for these tight programs were nearly impossible for anyone but their authors to understand or maintain. The useful life of the program was limited to the employment of its programmer.



During the second generation, programmers started dividing their programs into sections that resembled working sets, really segments, originally called roll in/roll out and now called **overlays**. The program could begin with only the first overlay loaded into memory. As the first section neared completion, it would instruct the system to lay the second section of code over the first section already in memory. Then the second section would be processed. As that section finished, it would call in the third section to be overlaid, and so on until the program was finished. Some programs had multiple overlays in main memory at once.

Although the swapping of overlays between main memory and secondary storage was done by the system, the tedious task of dividing the program into sections was done by the programmer. It was the concept of overlays that suggested paging and segmentation and led to virtual memory, which was then implemented through demand paging and segmentation schemes. These schemes are compared in Table 3.6.

(table 3.6)

Comparison of the advantages and disadvantages of virtual memory with paging and segmentation.

Virtual Memory with Paging	Virtual Memory with Segmentation
Allows internal fragmentation within page frames	Doesn't allow internal fragmentation
Doesn't allow external fragmentation	Allows external fragmentation
Programs are divided into equal-sized pages	Programs are divided into unequal-sized segments that contain logical groupings of code
The absolute address is calculated using page number and displacement	The absolute address is calculated using segment number and displacement
Requires PMT	Requires SMT

Segmentation allowed for sharing program code among users. This means that the shared segment contains: (1) an area where unchangeable code (called **reentrant code**) is stored, and (2) several data areas, one for each user. In this schema users share the code, which cannot be modified, and can modify the information stored in their own data areas as needed without affecting the data stored in other users' data areas.

Before virtual memory, sharing meant that copies of files were stored in each user's account. This allowed them to load their own copy and work on it at any time. This kind of sharing created a great deal of unnecessary system cost—the I/O overhead in loading the copies and the extra secondary storage needed. With virtual memory, those costs are substantially reduced because shared programs and subroutines are loaded on demand, satisfactorily reducing the storage requirements of main memory (although this is accomplished at the expense of the Memory Map Table).

The use of virtual memory requires cooperation between the Memory Manager (which tracks each page or segment) and the processor hardware (which issues the interrupt and resolves the virtual address). For example, when a page is needed that is not already in memory, a page fault is issued and the Memory Manager chooses a page frame, loads the page, and updates entries in the Memory Map Table and the Page Map Tables.

Virtual memory works well in a multiprogramming environment because most programs spend a lot of time waiting—they wait for I/O to be performed; they wait for

pages to be swapped in or out; and in a time-sharing environment, they wait when their time slice is up (their turn to use the processor is expired). In a multiprogramming environment, the waiting time isn't lost, and the CPU simply moves to another job.

Virtual memory has increased the use of several programming techniques. For instance, it aids the development of large software systems because individual pieces can be developed independently and linked later on.

Virtual memory management has several advantages:

- A job's size is no longer restricted to the size of main memory (or the free space within main memory).
- Memory is used more efficiently because the only sections of a job stored in memory are those needed immediately, while those not needed remain in secondary storage.
- It allows an unlimited amount of multiprogramming, which can apply to many jobs, as in dynamic and static partitioning, or many users in a time-sharing environment.
- It eliminates external fragmentation and minimizes internal fragmentation by combining segmentation and paging (internal fragmentation occurs in the program).
- It allows the sharing of code and data.
- It facilitates dynamic linking of program segments.

The advantages far outweigh these disadvantages:

- Increased processor hardware costs.
- Increased overhead for handling paging interrupts.
- Increased software complexity to prevent thrashing.

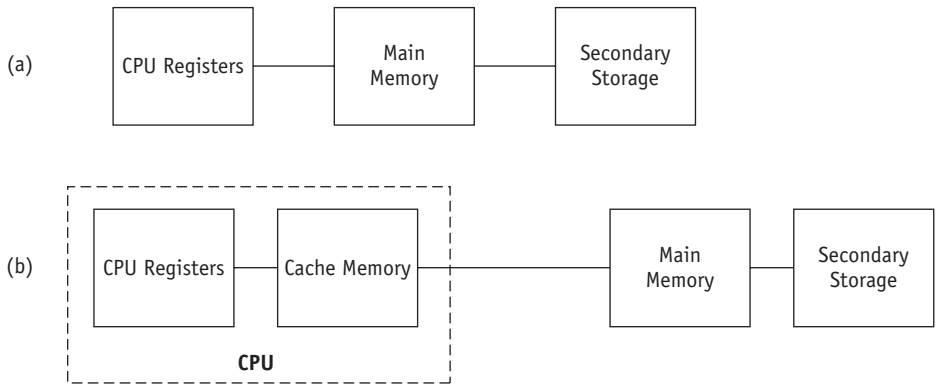
Cache Memory

Caching is based on the idea that the system can use a small amount of expensive high-speed memory to make a large amount of slower, less-expensive memory work faster than main memory.

Because the cache is small in size (compared to main memory), it can use faster, more expensive memory chips and can be five to ten times faster than main memory and match the speed of the CPU. Therefore, when frequently used data or instructions are stored in cache memory, memory access time can be cut down significantly and the CPU can execute instructions faster, thus raising the overall performance of the computer system.

(figure 3.17)

Comparison of (a) the traditional path used by early computers between main memory and the CPU and (b) the path used by modern computers to connect the main memory and the CPU via cache memory.



As shown in Figure 3.17(a), the original architecture of a computer was such that data and instructions were transferred from secondary storage to main memory and then to special-purpose registers for processing, increasing the amount of time needed to complete a program. However, because the same instructions are used repeatedly in most programs, computer system designers thought it would be more efficient if the system would not use a complete memory cycle every time an instruction or data value is required. Designers found that this could be done if they placed repeatedly used data in general-purpose registers instead of in main memory, but they found that this technique required extra work for the programmer. Moreover, from the point of view of the system, this use of general-purpose registers was not an optimal solution because those registers are often needed to store temporary results from other calculations, and because the amount of instructions used repeatedly often exceeds the capacity of the general-purpose registers.

To solve this problem, computer systems automatically store data in an intermediate memory unit called **cache memory**. This adds a middle layer to the original hierarchy. Cache memory can be thought of as an intermediary between main memory and the special-purpose registers, which are the domain of the CPU, as shown in Figure 3.17(b).

A typical microprocessor has two levels of caches: Level 1 (L1) and Level 2 (L2). Information enters the processor through the bus interface unit, which immediately sends one copy to the L2 cache, which is an integral part of the microprocessor and is directly connected to the CPU. A second copy is sent to a pair of L1 caches, which are built directly into the CPU. One of these L1 caches is designed to store instructions, while the other stores data to be used by the instructions. If an instruction needs more data, it is put on hold while the processor looks for it first in the data L1 cache, and then in the larger L2 cache before looking for it in main memory.

Because the L2 cache is an integral part of the microprocessor, data moves two to four times faster between the CPU and the L2 than between the CPU and main memory.

To understand the relationship between main memory and cache memory, consider the relationship between the size of the Web and the size of your private bookmark file. If main memory is the Web and cache memory is your private bookmark file where you collect your most frequently used Web addresses, then your bookmark file is small and may contain only 0.00001 percent of all the addresses in the Web; but the chance that you will soon visit a Web site that is in your bookmark file is high. Therefore, the purpose of your bookmark file is to keep your most recently accessed addresses so you can access them quickly, just as the purpose of cache memory is to keep handy the most recently accessed data and instructions so that the CPU can access them repeatedly without wasting time.

The movement of data, or instructions, from main memory to cache memory uses a method similar to that used in paging algorithms. First, cache memory is divided into blocks of equal size called slots. Then, when the CPU first requests an instruction or data from a location in main memory, the requested instruction and several others around it are transferred from main memory to cache memory where they are stored in one of the free slots. Moving a block at a time is based on the principle of locality of reference, which states that it is very likely that the next CPU request will be physically close to the one just requested. In addition to the block of data transferred, the slot also contains a label that indicates the main memory address from which the block was copied. When the CPU requests additional information from that location in main memory, cache memory is accessed first; and if the contents of one of the labels in a slot matches the address requested, then access to main memory is not required.

The algorithm to execute one of these "transfers from main memory" is simple to implement, as follows:

Main Memory Transfer Algorithm

1 CPU puts the address of a memory location in the Memory Address Register and requests data or an instruction to be retrieved from that address

2 A test is performed to determine if the block containing this address is already in a cache slot:

If YES, transfer the information to the CPU register – DONE

If NO: Access main memory for the block containing the requested address

Allocate a free cache slot to the block

Perform these in parallel: Transfer the information to CPU

Load the block into slot

DONE

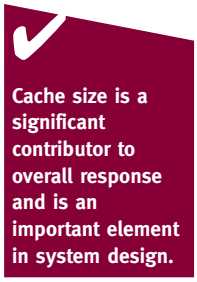
This algorithm becomes more complicated if there aren't any free slots, which can occur because the size of cache memory is smaller than that of main memory, which means that individual slots cannot be permanently allocated to blocks. To address this contingency, the system needs a policy for block replacement, which could be one similar to those used in page replacement.

When designing cache memory, one must take into consideration the following four factors:

- *Cache size.* Studies have shown that having any cache, even a small one, can substantially improve the performance of the computer system.
- *Block size.* Because of the principle of locality of reference, as block size increases, the ratio of number of references found in the cache to the total number of references will be high.
- *Block replacement algorithm.* When all the slots are busy and a new block has to be brought into the cache, a block that is least likely to be used in the near future should be selected for replacement. However, as we saw in paging, this is nearly impossible to predict. A reasonable course of action is to select a block that has not been used for a long time. Therefore, LRU is the algorithm that is often chosen for block replacement, which requires a hardware mechanism to specify the least recently used slot.
- *Rewrite policy.* When the contents of a block residing in cache are changed, it must be written back to main memory before it is replaced by another block. A rewrite policy must be in place to determine when this writing will take place. On the one hand, it could be done every time that a change occurs, which would increase the number of memory writes, increasing overhead. On the other hand, it could be done only when the block is replaced or the process is finished, which would minimize overhead but would leave the block in main memory in an inconsistent state. This would create problems in multiprocessor environments and in cases where I/O modules can access main memory directly.

The optimal selection of cache size and replacement algorithm can result in 80 to 90 percent of all requests being in the cache, making for a very efficient memory system. This measure of efficiency, called the cache hit ratio (h), is used to determine the performance of cache memory and represents the percentage of total memory requests that are found in the cache:

$$\text{HitRatio} = \frac{\text{number of requests found in the cache}}{\text{total number of requests}} * 100$$



For example, if the total number of requests is 10, and 6 of those are found in cache memory, then the hit ratio is 60 percent.

```
HitRatio = (6 / 10) * 100 = 60%
```

On the other hand, if the total number of requests is 100, and 9 of those are found in cache memory, then the hit ratio is only 9 percent.

```
HitRatio = (9 / 100) * 100 = 9%
```

Another way to measure the efficiency of a system with cache memory, assuming that the system always checks the cache first, is to compute the average memory access time using the following formula:

```
AvgMemAccessTime = AvgCacheAccessTime + (1 – h) * AvgMainMemAccTime
```

For example, if we know that the average cache access time is 200 nanoseconds (nsec) and the average main memory access time is 1000 nsec, then a system with a hit ratio of 60 percent will have an average memory access time of 600 nsec:

```
AvgMemAccessTime = 200 + (1 - 0.60) * 1000 = 600 nsec
```

A system with a hit ratio of 9 percent will show an average memory access time of 1110 nsec:

```
AvgMemAccessTime = 200 + (1 - 0.09) * 1000 = 1110 nsec
```

Conclusion

The Memory Manager has the task of allocating memory to each job to be executed, and reclaiming it when execution is completed.

Each scheme we discussed in Chapters 2 and 3 was designed to address a different set of pressing problems; but, as we have seen, when some problems were solved, others were created. Table 3.7 shows how memory allocation schemes compare.

(table 3.7)

Comparison of the memory allocation schemes discussed in Chapters 2 and 3.

Scheme	Problem Solved	Problem Created	Changes in Software
Single-user contiguous		Job size limited to physical memory size; CPU often idle	None
Fixed partitions	Idle CPU time	Internal fragmentation; Job size limited to partition size	Add Processor Scheduler; Add protection handler
Dynamic partitions	Internal fragmentation	External fragmentation	None
Relocatable dynamic partitions	Internal fragmentation	Compaction overhead; Job size limited to physical memory size	Compaction algorithm
Paged	Need for compaction	Memory needed for tables; Job size limited to physical memory size; Internal fragmentation returns	Algorithms to handle Page Map Tables
Demand paged	Job size no longer limited to memory size; More efficient memory use; Allows large-scale multiprogramming and time-sharing	Larger number of tables; Possibility of thrashing; Overhead required by page interrupts; Necessary paging hardware	Page replacement algorithm; Search algorithm for pages in secondary storage
Segmented	Internal fragmentation	Difficulty managing variable-length segments in secondary storage; External fragmentation	Dynamic linking package; Two-dimensional addressing scheme
Segmented/ demand paged	Large virtual memory; Segment loaded on demand	Table handling overhead; Memory needed for page and segment tables	Three-dimensional addressing scheme

The Memory Manager is only one of several managers that make up the operating system. Once the jobs are loaded into memory using a memory allocation scheme, the Processor Manager must allocate the processor to process each job in the most efficient manner possible. We will see how that is done in the next chapter.

Key Terms

address resolution: the process of changing the address of an instruction or data item to the address in main memory at which it is to be loaded or relocated.

associative memory: the name given to several registers, allocated to each active process, whose contents associate several of the process segments and page numbers with their main memory addresses.

cache memory: a small, fast memory used to hold selected data and to provide faster access than would otherwise be possible.

clock cycle: the elapsed time between two ticks of the computer's system clock.

clock page replacement policy: a variation of the LRU policy that removes from main memory the pages that show the least amount of activity during recent clock cycles.

demand paging: a memory allocation scheme that loads a program's page into memory at the time it is needed for processing.

displacement: in a paged or segmented memory allocation environment, the difference between a page's relative address and the actual machine language address. Also called offset.

FIFO anomaly: an unusual circumstance through which adding more page frames causes an increase in page interrupts when using a FIFO page replacement policy.

first-in first-out (FIFO) policy: a page replacement policy that removes from main memory the pages that were brought in first.

Job Table (JT): a table in main memory that contains two values for each active job—the size of the job and the memory location where its page map table is stored.

least recently used (LRU) policy: a page-replacement policy that removes from main memory the pages that show the least amount of recent activity.

locality of reference: behavior observed in many executing programs in which memory locations recently referenced, and those near them, are likely to be referenced in the near future.

Memory Map Table (MMT): a table in main memory that contains as many entries as there are page frames and lists the location and free/busy status for each one.

offset: *see* displacement.

page: a fixed-size section of a user's job that corresponds in size to page frames in main memory.

page fault: a type of hardware interrupt caused by a reference to a page not residing in memory. The effect is to move a page out of main memory and into secondary storage so another page can be moved into memory.

page fault handler: the part of the Memory Manager that determines if there are empty page frames in memory so that the requested page can be immediately copied from secondary storage, or determines which page must be swapped out if all page frames are busy. Also known as a page interrupt handler.

page frame: an individual section of main memory of uniform size into which a single page may be loaded without causing external fragmentation.

Page Map Table (PMT): a table in main memory with the vital information for each page including the page number and its corresponding page frame memory address.

page replacement policy: an algorithm used by virtual memory systems to decide which page or segment to remove from main memory when a page frame is needed and memory is full.

page swapping: the process of moving a page out of main memory and into secondary storage so another page can be moved into memory in its place.

paged memory allocation: a memory allocation scheme based on the concept of dividing a user's job into sections of equal size to allow for noncontiguous program storage during execution.

reentrant code: code that can be used by two or more processes at the same time; each shares the same copy of the executable code but has separate data areas.

sector: a division in a disk's track, sometimes called a "block." The tracks are divided into sectors during the formatting process.

segment: a variable-size section of a user's job that contains a logical grouping of code.

Segment Map Table (SMT): a table in main memory with the vital information for each segment including the segment number and its corresponding memory address.

segmented/demand paged memory allocation: a memory allocation scheme based on the concept of dividing a user's job into logical groupings of code and loading them into memory as needed to minimize fragmentation.

segmented memory allocation: a memory allocation scheme based on the concept of dividing a user's job into logical groupings of code to allow for noncontiguous program storage during execution.

subroutine: also called a "subprogram," a segment of a program that can perform a specific function. Subroutines can reduce programming time when a specific function is required at more than one point in a program.

thrashing: a phenomenon in a virtual memory system where an excessive amount of page swapping back and forth between main memory and secondary storage results in higher overhead and little useful work.

virtual memory: a technique that allows programs to be executed even though they are not stored entirely in memory.

working set: a collection of pages to be kept in main memory for each active process in a virtual memory environment.

Interesting Searches

- Memory Card Suppliers
- Virtual Memory
- Working Set
- Cache Memory
- Thrashing

Exercises

Research Topics

A. The sizes of pages and page frames are often identical. Search academic sources to discover typical page sizes, what factors are considered by operating system developers when establishing these sizes, and whether or not hardware considerations are important. Cite your sources.

B. Core memory consists of the CPU and arithmetic logic unit but not the attached cache memory. On the Internet or using academic sources, research the design of multi-core memory and identify the roles played by cache memory Level 1 and Level 2. Does the implementation of cache memory on multi-core chips vary from one manufacturer to another? Explain and cite your sources.

Exercises

1. Compare and contrast internal fragmentation and external fragmentation. Explain the circumstances where one might be preferred over the other.
2. Describe how the function of the Page Map Table differs in paged vs. segmented/demand paging memory allocation.
3. Describe how the operating system detects thrashing. Once thrashing is detected, explain what the operating system can do to stop it.
4. Given that main memory is composed of three page frames for public use and

that a seven-page program (with pages a, b, c, d, e, f, g) requests pages in the following order:

a, b, a, c, d, a, e, f, g, c, b, g

a. Using the FIFO page removal algorithm, do a page trace analysis indicating page faults with asterisks (*). Then compute the failure and success ratios.

b. Increase the size of memory so it contains four page frames for public use. Using the same page requests as above and FIFO, do another page trace analysis and compute the failure and success ratios.

c. Did the result correspond with your intuition? Explain.

5. Given that main memory is composed of three page frames for public use and that a program requests pages in the following order:

a, d, b, a, f, b, e, c, g, f, b, g

a. Using the FIFO page removal algorithm, perform a page trace analysis indicating page faults with asterisks (*). Then compute the failure and success ratios.

b. Using the LRU page removal algorithm, perform a page trace analysis and compute the failure and success ratios.

c. Which is better? Why do you think it is better? Can you make general statements from this example? Why or why not?

6. Let us define "most-recently-used" (MRU) as a page removal algorithm that removes from memory the most recently used page. Perform a page trace analysis using three page frames and the page requests from the previous exercise. Compute the failure and success ratios and explain why you think MRU is, or is not, a viable memory allocation system.

7. By examining the reference bits for the six pages shown in Figure 3.11, identify which of the six pages was referenced most often as of the last time snapshot [shown in (e)]. Which page was referenced least often? Explain your answer.

8. To implement LRU, each page needs a referenced bit. If we wanted to implement a least frequently used (LFU) page removal algorithm, in which the page that was used the least would be removed from memory, what would we need to add to the tables? What software modifications would have to be made to support this new algorithm?

9. Calculate the cache Hit Ratio using the formula presented at the end of this chapter assuming that the total number of requests is 2056 and 1209 of those requests are found in the cache.

10. Assuming a hit ratio of 67 percent, calculate the Average Memory Access Time using the formula presented in this chapter if the Average Cache Access Time is 200 nsec and the Average Main Memory Access Time is 500 nsec.

11. Assuming a hit ratio of 31 percent, calculate the Average Memory Access Time using the formula presented in this chapter if the Average Cache Access Time is 125 nsec and the Average Main Memory Access Time is 300 nsec.
12. Using a paged memory allocation system with a page size of 2,048 bytes and an identical page frame size, and assuming the incoming data file is 25,600, calculate how many pages will be created by the file. Calculate the size of any resulting fragmentation. Explain whether this situation will result in internal fragmentation, external fragmentation, or both.

Advanced Exercises

13. Given that main memory is composed of four page frames for public use, use the following table to answer all parts of this problem:

Page Frame	Time When Loaded	Time When Last Referenced	Referenced Bit	Modified Bit
0	126	279	0	0
1	230	280	1	0
2	120	282	1	1
3	160	290	1	1

    a. The contents of which page frame would be swapped out by FIFO?
    b. The contents of which page frame would be swapped out by LRU?
    c. The contents of which page frame would be swapped out by MRU?
    d. The contents of which page frame would be swapped out by LFU?

14. Given three subroutines of 700, 200, and 500 words each, if segmentation is used then the total memory needed is the sum of the three sizes (if all three routines are loaded). However, if paging is used then some storage space is lost because subroutines rarely fill the last page completely, and that results in internal fragmentation. Determine the total amount of wasted memory due to internal fragmentation when the three subroutines are loaded into memory using each of the following page sizes:
    a. 100 words
    b. 600 words
    c. 700 words
    d. 900 words

15. Given the following Segment Map Tables for two jobs:

SMT for Job 1

Segment Number	Memory Location
0	4096
1	6144
2	9216
3	2048
4	7168

SMT for Job 2

Segment number	Memory location
0	2048
1	6144
2	9216

a. Which segments, if any, are shared between the two jobs?

b. If the segment now located at 7168 is swapped out and later reloaded at 8192, and the segment now at 2048 is swapped out and reloaded at 1024, what would the new segment tables look like?

Programming Exercises

16. This problem studies the effect of changing page sizes in a demand paging system.

The following sequence of requests for program words is taken from a 460-word program: 10, 11, 104, 170, 73, 309, 185, 245, 246, 434, 458, 364. Main memory can hold a total of 200 words for this program and the page frame size will match the size of the pages into which the program has been divided.

Calculate the page numbers according to the page size, divide by the page size, and the quotient gives the page number. The number of page frames in memory is the total number, 200, divided by the page size. For example, in problem (a) the page size is 100, which means that requests 10 and 11 are on Page 0, and requests 104 and 170 are on Page 1. The number of page frames is two.

a. Find the success frequency for the request list using a FIFO replacement algorithm and a page size of 100 words (there are two page frames).

b. Find the success frequency for the request list using a FIFO replacement algorithm and a page size of 20 words (10 pages, 0 through 9).

c. Find the success frequency for the request list using a FIFO replacement algorithm and a page size of 200 words.

d. What do your results indicate? Can you make any general statements about what happens when page sizes are halved or doubled?

e. Are there any overriding advantages in using smaller pages? What are the offsetting factors? Remember that transferring 200 words of information takes less than twice as long as transferring 100 words because of the way secondary storage devices operate (the transfer rate is higher than the access [search/find] rate).

f. Repeat (a) through (c) above, using a main memory of 400 words. The size of each page frame will again correspond to the size of the page.

g. What happened when more memory was given to the program? Can you make some general statements about this occurrence? What changes might you expect to see if the request list was much longer, as it would be in real life?

h. Could this request list happen during the execution of a real program? Explain.

i. Would you expect the success rate of an actual program under similar conditions to be higher or lower than the one in this problem?

17. Given the following information for an assembly language program:

Job size = 3126 bytes

Page size = 1024 bytes

instruction at memory location 532:	`Load 1, 2098`
instruction at memory location 1156:	`Add 1, 2087`
instruction at memory location 2086:	`Sub 1, 1052`
data at memory location 1052:	`015672`
data at memory location 2098:	`114321`
data at memory location 2087:	`077435`

a. How many pages are needed to store the entire job?

b. Compute the page number and displacement for each of the byte addresses where the data is stored. (Remember that page numbering starts at zero).

c. Determine whether the page number and displacements are legal for this job.

d. Explain why the page number and/or displacements may not be legal for this job.

e. Indicate what action the operating system might take when a page number or displacement is not legal.