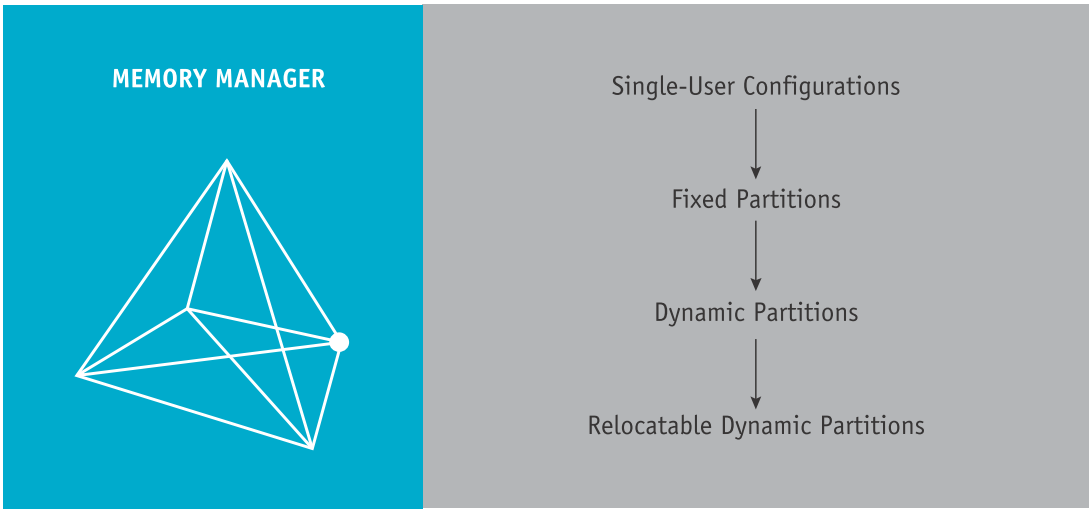
Chapter 2 Memory Management: Simple Systems



"Memory is the primary and fundamental power, without which there could be no other intellectual operation."

—Samuel Johnson (1709–1784)

Learning Objectives

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

- The basic functionality of the four memory allocation schemes presented in this chapter: single user, fixed partitions, dynamic partitions, and relocatable dynamic partitions
- Best-fit memory allocation as well as first-fit memory allocation
- How a memory list keeps track of available memory
- The importance of memory deallocation
- The importance of the bounds register in memory allocation schemes
- The role of compaction and how it can improve memory allocation efficiency

The management of **main memory** is critical. In fact, for many years, the performance of the *entire* system was directly dependent on two things: How much memory was available and how that memory was optimized while jobs were being processed.

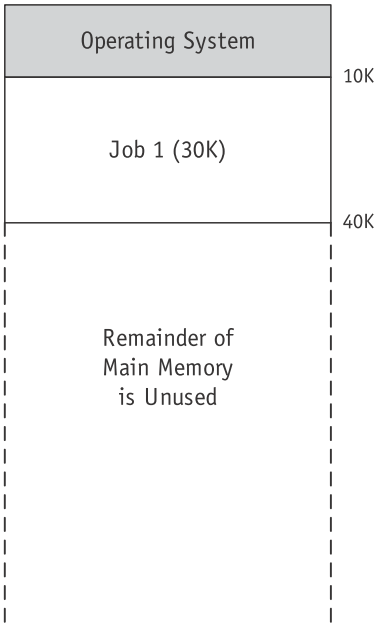
This chapter introduces the role of main memory (also known as random access memory or **RAM**, core memory, or primary storage) and four types of memory allocation schemes: single-user systems, fixed partitions, dynamic partitions, and relocatable dynamic partitions. Let's begin with the simplest memory management scheme—the one used with the simplest computer systems.

Single-User Contiguous Scheme

This memory allocation scheme works like this: before execution can begin, each job or program is loaded in its entirety into memory and allocated as much contiguous space in memory as it needs, as shown in Figure 2.1. The key words here are *entirety* and *contiguous*. If the program is too large to fit into the available memory space, it cannot begin execution.

A single-user scheme supports only one job at a time. It's not possible to share main memory.

This scheme demonstrates a significant limiting factor of all computers—they have only a finite amount of memory. If a program doesn't fit, then either the size of the main memory must be increased, or the program must be modified to fit, often by revising it to be smaller.



(figure 2.1)

Only one program can fit into memory at a time, even if there is room to accommodate other waiting jobs.

Single-user systems in a non-networked environment allocate, to each user, access to all available main memory for each job, and jobs are processed sequentially, one

after the other. To allocate memory, the amount of work required from the operating system's Memory Manager is minimal, as described in these steps:

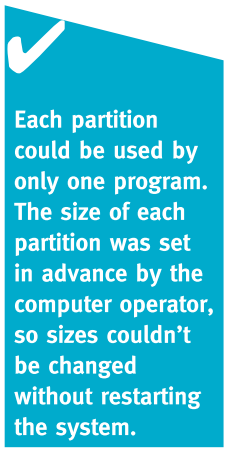
1. Evaluate the incoming process to see if it is small enough to fit into the available space. If it is, load it into memory; if not, reject it and evaluate the next incoming process,
2. Monitor the occupied memory space. When the resident process ends its execution and no longer needs to be in memory, make the entire amount of main memory space available and return to Step 1, evaluating the next incoming process.

An "Algorithm to Load a Job in a Single-User System" using pseudocode and demonstrating these steps can be found in Appendix A.

Once the program is entirely loaded into memory, it begins its execution and remains there until execution is complete, either by finishing its work or through the intervention of the operating system, such as when an error is detected.

One major problem with this type of memory allocation scheme is that it doesn't support multiprogramming (multiple jobs or processes occupying memory at the same time); it can handle only one at a time. When these single-user configurations were first made available commercially in the late 1940s and early 1950s, they were used in research institutions but proved unacceptable for the business community—it wasn't cost effective to spend almost $200,000 for a piece of equipment that could be used by only one person at a time. The next scheme introduced memory partitions.

Fixed Partitions



Each partition could be used by only one program. The size of each partition was set in advance by the computer operator, so sizes couldn't be changed without restarting the system.

The first attempt to allow for multiprogramming used **fixed partitions** (also known as static partitions) within main memory—each partition could be assigned to one job. A system with four partitions could hold four jobs in memory at the same time. One fact remained the same, however—these partitions were static, so the systems administrator had to turn off the entire system to reconfigure their sizes, and any job that couldn't fit into the largest partition could not be executed.

An important factor was introduced with this scheme: protection of the job's memory space. Once a partition was assigned to a job, the jobs in other memory partitions had to be prevented from invading its boundaries, either accidentally or intentionally. This problem of partition intrusion didn't exist in single-user contiguous allocation schemes because only one job was present in main memory at any given time—only the portion of main memory that held the operating system had to be protected. However, for the fixed partition allocation schemes, protection was mandatory for each partition in main memory. Typically this was the joint responsibility of the hardware of the computer and of the operating system.

The algorithm used to store jobs in memory requires a few more steps than the one used for a single-user system because the size of the job must be matched with the size of the available partitions to make sure it fits completely. ("An Algorithm to Load a Job in a Fixed Partition" is in Appendix A.) Remember, this scheme also required that the entire job be loaded into memory before execution could begin.

To do so, the Memory Manager could perform these steps in a two-partition system:

1. Check the incoming job's memory requirements. If it's greater than the size of the largest partition, reject the job and go to the next waiting job. If it's less than the largest partition, go to Step 2.
2. Check the job size against the size of the first available partition. If the job is small enough to fit, see if that partition is free. If it is available, load the job into that partition. If it's busy with another job, go to Step 3.
3. Check the job size against the size of the second available partition. If the job is small enough to fit, check to see if that partition is free. If it is available, load the incoming job into that partition. If not, go to Step 4.
4. Because neither partition is available now, place the incoming job in the waiting queue for loading at a later time. Return to Step 1 to evaluate the next incoming job.

This partition scheme is more flexible than the single-user scheme because it allows more than one program to be in memory at the same time. However, it still requires that the *entire* program be stored *contiguously* and *in memory* from the beginning to the end of its execution.

In order to allocate memory spaces to jobs, the Memory Manager must maintain a table which shows each memory partition's size, its **address**, its access restrictions, and its current status (free or busy). Table 2.1 shows a simplified version for the system illustrated in Figure 2.2. Note that Table 2.1 and the other tables in this chapter have been simplified. More detailed discussions of these tables and their contents are presented in Chapter 8, "File Management."

Partition Size	Memory Address	Access	Partition Status
100K	200K	Job 1	Busy
25K	300K	Job 4	Busy
25K	325K		Free
50K	350K	Job 2	Busy

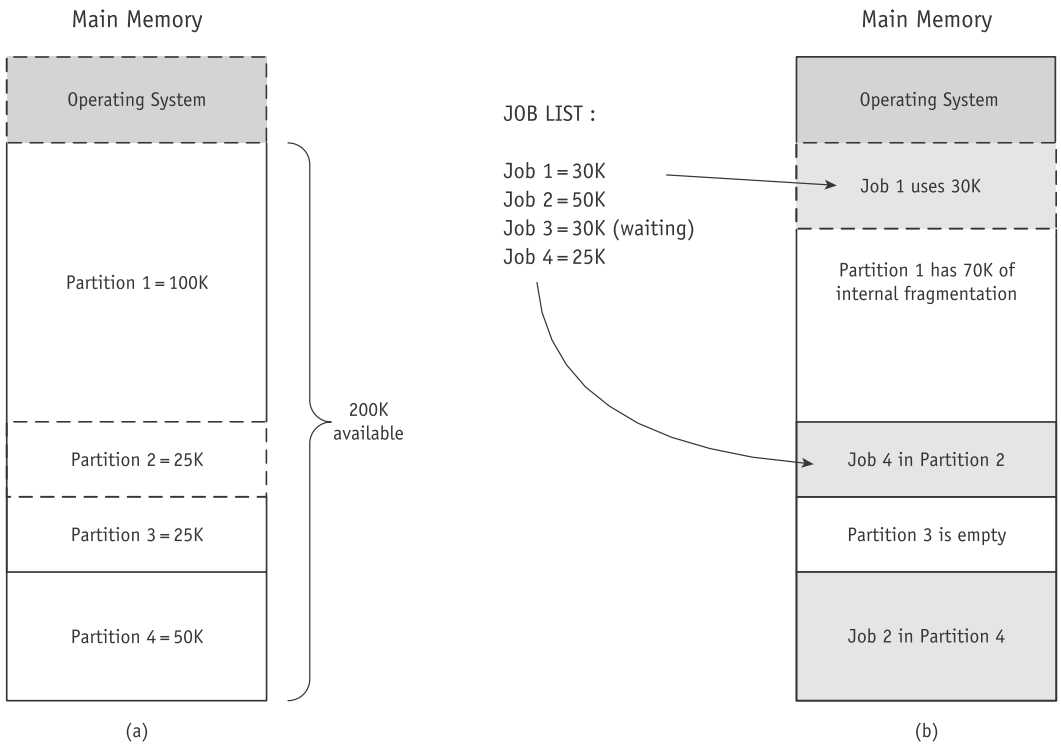
(table 2.1)

A simplified fixed-partition memory table with the free partition shaded.

When each resident job terminates, the status of its memory partition is changed from busy to free to make it available to an incoming job.

(figure 2.2)

As the jobs listed in Table 2.1 are loaded into the four fixed partitions, Job 3 must wait even though Partition 1 has 70K of available memory. Jobs are allocated space on the basis of "first available partition of required size."



The fixed partition scheme works well if all of the jobs that run on the system are of similar size or if the sizes are known ahead of time and don't vary between reconfigurations. Ideally, that would require accurate, advance knowledge of all the jobs waiting to be run on the system in the coming hours, days, or weeks. However, under most circumstances, the operator chooses partition sizes in an arbitrary fashion, and thus not all incoming jobs fit in them.

There are significant consequences if the partition sizes are too small; large jobs will need to wait if the large partitions are already booked, and they will be rejected if they're too big to fit into the largest partition.

On the other hand, if the partition sizes are too big, memory is wasted. Any job that occupies less than the entire partition (the vast majority will do so) will cause unused memory in the partition to remain idle. Remember that each partition is an indivisible unit that can be allocated to only one job at a time. Figure 2.3 demonstrates one such circumstance: Job 3 is kept waiting because it's too big for Partition 3, even though there is more than enough unused space for it in Partition 1, which is claimed by Job 1.

This phenomenon of less-than-complete use of memory space in a fixed partition is called **internal fragmentation** (because it's inside a partition) and is a major drawback to this memory allocation scheme.

Jay W. Forrester (1918–)

Jay Forrester led the groundbreaking MIT Whirlwind computer project (1947-1953) that developed the first practical and reliable, high-speed random access memory for digital computers. At the time it was called "coincident current magnetic core storage." This invention became the standard memory device for digital computers and provided the foundation for main memory development through the 1970s. Forrester is the recipient of numerous awards, including the IEEE Computer Pioneer Award (1982), the U.S. National Medal of Technology and Innovation (1989), and induction into the Operational Research Hall of Fame (2006).

For more information:
http://www.computerhistory.org/fellowawards/hall/bios/Jay,Forrester/

Received the IEEE Computer Pioneer Award: "For exceptional advances in the digital computer through his invention and application of the magnetic-core random-access memory, employing coincident current addressing."

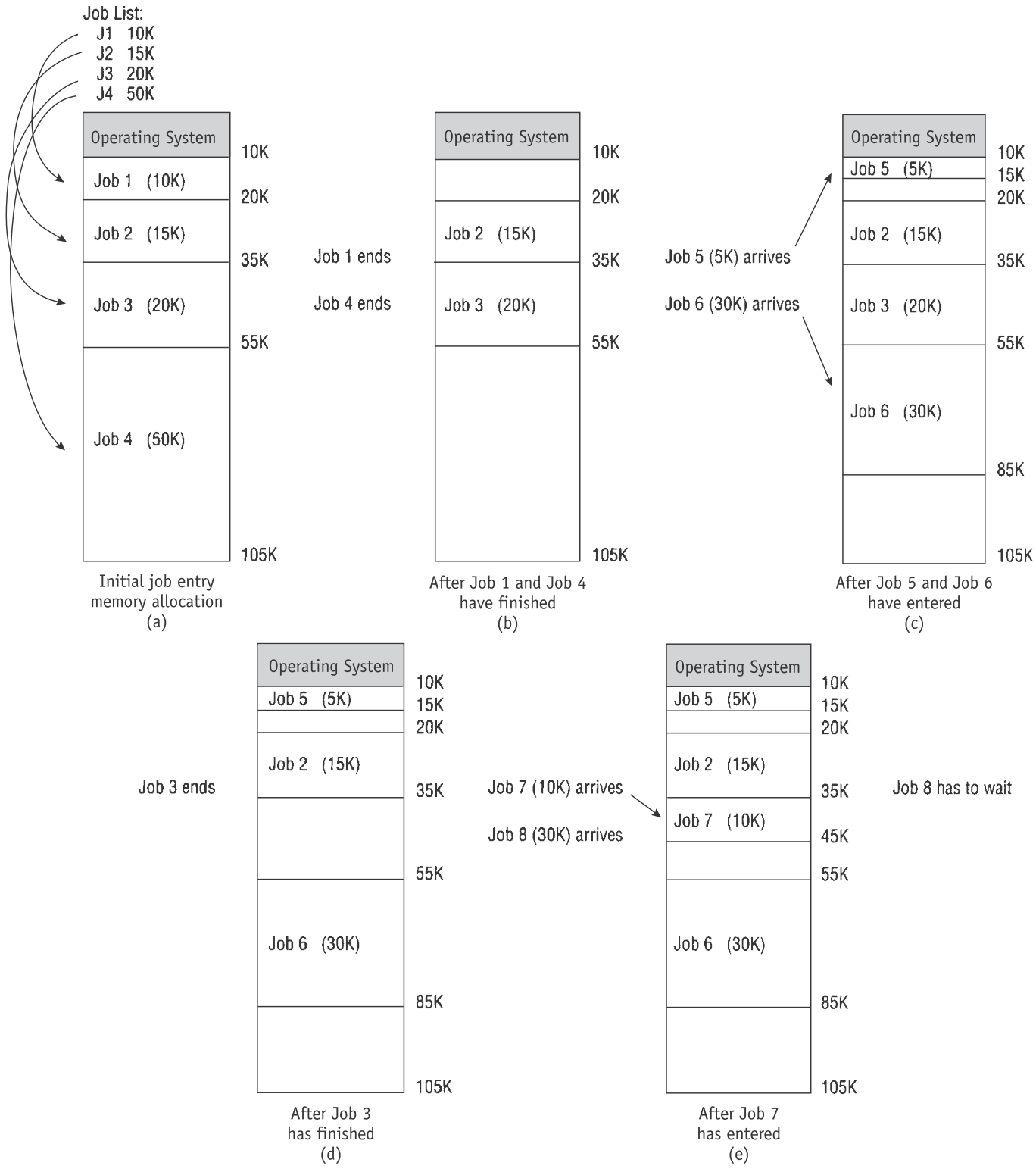
Dynamic Partitions

With the introduction of the **dynamic partition** allocation scheme, memory is allocated to an incoming job in one contiguous block, and each job is given only as much memory as it requests when it is loaded for processing. Although this is a significant improvement over fixed partitions because memory is no longer wasted inside each partition, it introduces another problem.

There are two types of fragmentation: internal and external. The type depends on the location of the wasted space.

It works well when the first jobs are loaded. As shown in Figure 2.4, a dynamic partition scheme allocates memory efficiently as each of the first few jobs are loaded, but when those jobs finish and new jobs enter the system (which are not the same size as those that just vacated memory), the newer jobs are allocated space in the available partition spaces on a priority basis. Figure 2.4 demonstrates first-come, first-served priority—that is, each job is loaded into the first available partition. Therefore, the subsequent allocation of memory creates fragments of free memory *between* partitions of allocated memory. This problem is called **external fragmentation** and, like internal fragmentation, allows memory to be wasted.

In Figure 2.3, notice the difference between the first snapshot (a), where the entire amount of main memory is used efficiently, and the last snapshot, (e), where there are



(figure 2.3)

Main memory use during dynamic partition allocation. Five snapshots (a-e) of main memory as eight jobs are submitted for processing and allocated space on the basis of "first come, first served." Job 8 has to wait (e) even though there's enough free memory between partitions to accommodate it.

three free partitions of 5K, 10K, and 20K—35K in all—enough to accommodate Job 8, which requires only 30K. However, because the three memory blocks are separated by partitions, Job 8 cannot be loaded in a contiguous manner. Therefore, this scheme forces Job 8 to wait.

Before we go to the next allocation scheme, let's examine two ways that an operating system can allocate free sections of memory.

Best-Fit and First-Fit Allocation

For both fixed and dynamic memory allocation schemes, the operating system must keep lists of each memory location, noting which are free and which are busy. Then, as new jobs come into the system, the free partitions must be allocated fairly, according to the policy adopted by the programmers who designed and wrote the operating system.

Memory partitions may be allocated on the basis of first-fit memory allocation or best-fit memory allocation. For both schemes, the Memory Manager keeps detailed lists of the free and busy sections of memory either by size or by location. The **best-fit allocation method keeps** the free/busy lists in order by size, from smallest to largest. The **first-fit allocation method** keeps the free/busy lists organized by memory locations, from low-order memory to high-order memory. Each has advantages depending on the needs of the particular allocation scheme. Best-fit usually makes the best use of memory space; first-fit is faster.

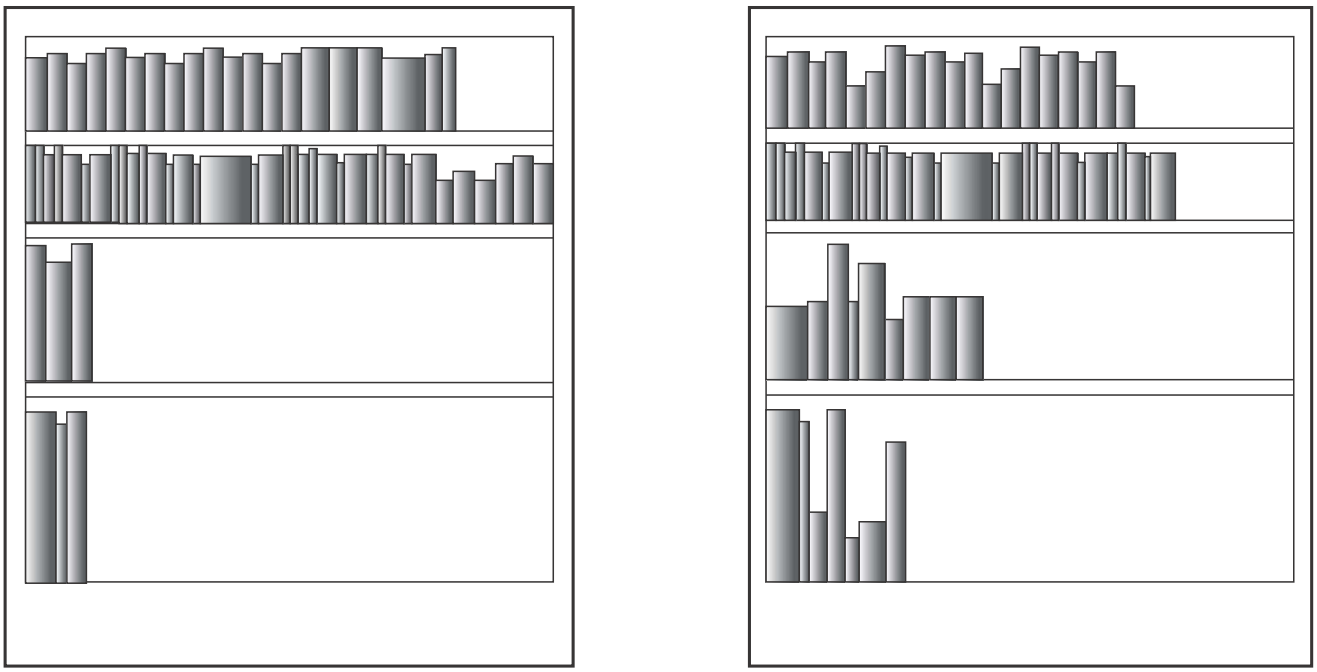
To understand the trade-offs, imagine that you are in charge of a small local library. You have books of all shapes and sizes, and there's a continuous stream of people taking books out and bringing them back—someone is always waiting. It's clear that your library is a popular place and you'll always be busy, without any time to rearrange the books on the shelves.

You need a system. Your shelves have fixed partitions with a few tall spaces for oversized books, several shelves for paperbacks and DVDs, and lots of room for textbooks. You'll need to keep track of which spaces on the shelves are full and where you have spaces for more. For the purposes of our example, we'll keep two lists: a free list showing all the available spaces, and a busy list showing all the occupied spaces. Each list will include the size and location of each space.

As each book is removed from its place on the shelf, you'll update both lists by removing the space from the busy list and adding it to the free list. Then as your books are returned and placed back on a shelf, the two lists will be updated again.

There are two ways to organize your lists: by size or by location. If they're organized by size, the spaces for the smallest books are at the top of the list and those for the

largest are at the bottom, as shown in Figure 2.4. When they're organized by location, the spaces closest to your lending desk are at the top of the list and the areas farthest away are at the bottom. Which option is best? It depends on what you want to optimize: space or speed.



(figure 2.4)

The bookshelf on the left uses the best-fit allocation. The one on the right is first-fit allocation, where some small items occupy space that could hold larger ones.

If the lists are organized by size, you're optimizing your shelf space—as books arrive, you'll be able to put them in the spaces that fit them best. This is a best-fit scheme. If a paperback is returned, you'll place it on a shelf with the other paperbacks or at least with other small books. Similarly, oversized books will be shelved with other large books. Your lists make it easy to find the smallest available empty space where the book can fit. The disadvantage of this system is that you're wasting time looking for the best space. Your other customers have to wait for you to put each book away, so you won't be able to process as many customers as you could with the other kind of list.

In the second case, a list organized by shelf location, you're optimizing the time it takes you to put books back on the shelves by using a first-fit scheme. This system ignores the size of the book that you're trying to put away. If the same paperback book arrives, you can quickly find it an empty space. In fact, any nearby empty space will suffice if it's large enough—and if it's close to your desk. In this case you are optimizing the time it takes you to shelve the books.

This is a fast method of shelving books, and if speed is important, it's the better of the two alternatives. However, it isn't a good choice if your shelf space is limited or if many large books are returned, because large books must wait for the large spaces. If all of your large spaces are filled with small books, the customers returning large books must wait until a suitable space becomes available. (Eventually you'll need time to rearrange the books and compact your collection.)

Figure 2.5 shows how a large job can have problems with a first-fit memory allocation list. Jobs 1, 2, and 4 are able to enter the system and begin execution; Job 3 has to wait even though there would be more than enough room to accommodate it if all of the fragments of memory were added together. First-fit offers fast allocation, but it isn't always efficient. (On the other hand, the same job list using a best-fit scheme would use memory more efficiently, as shown in Figure 2.6. In this particular case, a best-fit scheme would yield better memory utilization.)

Job List:

Job number	Memory requested
J1	10K
J2	20K
J3	30K*
J4	10K

Memory List:

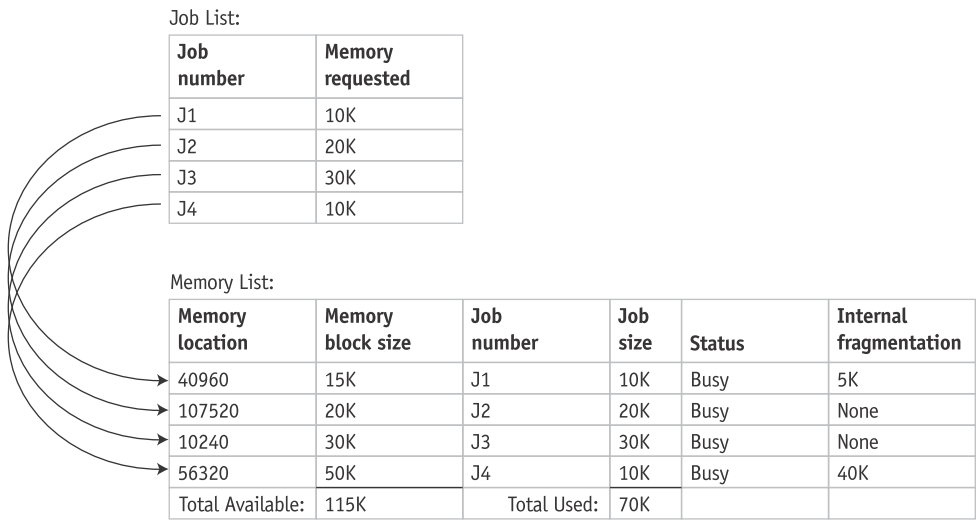
Memory location	Memory block size	Job number	Job size	Status	Internal fragmentation
10240	30K	J1	10K	Busy	20K
40960	15K	J4	10K	Busy	5K
56320	50K	J2	20K	Busy	30K
107520	20K			Free	
Total Available:	115K	Total Used:	40K		

(figure 2.5)

Using a first-fit scheme, Job 1 claims the first available space. Job 2 then claims the first partition large enough to accommodate it, but by doing so it takes the last block large enough to accommodate Job 3. Therefore, Job 3 (indicated by the asterisk) must wait until a large block becomes available, even though there's 75K of unused memory space (internal fragmentation). Notice that the memory list is ordered according to memory location.

Memory use has been increased, but the memory allocation process takes more time. What's more, while internal fragmentation has been diminished, it hasn't been completely eliminated.

The first-fit algorithm (included in Appendix A) assumes that the Memory Manager keeps two lists, one for free memory blocks and one for busy memory blocks. The operation consists of a simple loop that compares the size of each job to the size of each memory block until a block is found that's large enough to fit the job. Then the job is stored into that block of memory, and the Memory Manager fetches the next job from the entry queue. If the entire list is searched in vain, then the job is placed into a waiting queue. The Memory Manager then fetches the next job and repeats the process.



(figure 2.6)

Best-fit free scheme. Job 1 is allocated to the closestfitting free partition, as are Job 2 and Job 3. Job 4 is allocated to the only available partition although it isn't the best-fitting one. In this scheme, all four jobs are served without waiting. Notice that the memory list is ordered according to memory size. This scheme uses memory more efficiently but it's slower to implement.

In Table 2.2, a request for a block of 200 spaces has just been given to the Memory Manager. (The spaces may be words, bytes, or any other unit the system handles.) Using the first-fit scheme and starting from the top of the list, the Memory Manager locates the first block of memory large enough to accommodate the job, which is at location 6785. The job is then loaded, starting at location 6785 and occupying the next 200 spaces. The next step is to adjust the free list to indicate that the block of free memory now starts at location 6985 (not 6785 as before) and that it contains only 400 spaces (not 600 as before).

(table 2.2)

These two snapshots of memory show the status of each memory block before and after 200 spaces are allocated at address 6785, using the first-fit algorithm. (Note: All values are in decimal notation unless otherwise indicated.)

Status Before Request		Status After Request	
Beginning Address	**Free Memory Block Size**	**Beginning Address**	**Free Memory Block Size**
4075	105	4075	105
5225	5	5225	5
6785	600	*6985	400
7560	20	7560	20
7600	205	7600	205
10250	4050	10250	4050
15125	230	15125	230
24500	1000	24500	1000

The best-fit algorithm is slightly more complex because the goal is to find the smallest memory block into which the job will fit. One of the problems with it is that the entire table must be searched before an allocation can be made because the memory blocks are physically stored in sequence according to their location in memory (and not by memory block sizes as shown in Figure 2.6). The system could execute an

algorithm to continuously rearrange the list in ascending order by memory block size, but that would add more overhead and might not be an efficient use of processing time in the long run.

The best-fit algorithm (included in Appendix A) is illustrated showing only the list of free memory blocks. Table 2.3 shows the free list before and after the best-fit block has been allocated to the same request presented in Table 2.2.

Status Before Request		Status After Request	
Beginning Address	**Free Memory Block Size**	**Beginning Address**	**Free Memory Block Size**
4075	105	4075	105
5225	5	5225	5
6785	600	6785	600
7560	20	7560	20
7600	205	*7800	5
10250	4050	10250	4050
15125	230	15125	230
24500	1000	24500	1000

(table 2.3)

These two snapshots of memory show the status of each memory block before and after 200 spaces are allocated at address 7600, using the best-fit algorithm.

In Table 2.3, a request for a block of 200 spaces has just been given to the Memory Manager. Using the best-fit algorithm and starting from the top of the list, the Memory Manager searches the entire list and locates a block of memory starting at location 7600, which is the block that's the smallest one that's also large enough to accommodate the job. The choice of this block minimizes the wasted space (only 5 spaces are wasted, which is less than in the four alternative blocks). The job is then stored, starting at location 7600 and occupying the next 200 spaces. Now the free list must be adjusted to show that the block of free memory starts at location 7800 (not 7600 as before) and that it contains only 5 spaces (not 205 as before). But it doesn't eliminate wasted space. Note that while the best-fit resulted in a better fit, it also resulted (as it does in the general case) in a smaller free space (5 spaces), which is known as a sliver.

Which is best—first-fit or best-fit? For many years there was no way to answer such a general question because performance depends on the job mix. In recent years, access times have become so fast that the scheme that saves the more valuable resource, memory space, may be the best. Research continues to focus on finding the optimum allocation scheme.

In the exercises at the end of this chapter, two other hypothetical allocation schemes are explored: next-fit, which starts searching from the last allocated block for the next available block when a new job arrives; and worst-fit (the opposite of best-fit), which allocates the largest free available block to the new job. Although it's a good way to explore the theory of memory allocation, it might not be the best choice for a real system.

Deallocation

Until now, we've considered only the problem of how memory blocks are allocated, but eventually there comes a time for the release of memory space, called **deallocation**.

When memory is deallocated, it creates an opportunity for external fragmentation.

For a fixed partition system, the process is quite straightforward. When the job is completed, the Memory Manager immediately deallocates it by resetting the status of the entire memory block from "busy" to "free." Any code—for example, binary values with 0 indicating free and 1 indicating busy—may be used, so the mechanical task of deallocating a block of memory is relatively simple.

A dynamic partition system uses a more complex algorithm (shown in Appendix A) because it tries to combine free areas of memory whenever possible. Therefore, the system must be prepared for three alternative situations:

- Case 1: When the block to be deallocated is adjacent to another free block
- Case 2: When the block to be deallocated is between two free blocks
- Case 3: When the block to be deallocated is isolated from other free blocks

Case 1: Joining Two Free Blocks

Table 2.4 shows how deallocation occurs in a dynamic memory allocation system when the job to be deallocated is next to one free memory block.

After deallocation, the free list looks like the one shown in Table 2.5.

(table 2.4)

This is the original free list before deallocation for Case 1. The asterisk indicates the free memory block (of size 5) that's adjacent to the soon-to-be-free memory block (of size 200) that's shaded.

Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
7560	20	Free
(7600)	(200)	(Busy)[1]
*7800	5	Free
10250	4050	Free
15125	230	Free
24500	1000	Free

[1]Although the numbers in parentheses don't appear in the free list, they've been inserted here for clarity. The job size is 200 and its beginning location is 7600.

Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
7560	20	Free
*7600	205	Free
10250	4050	Free
15125	230	Free
24500	1000	Free

(table 2.5)

Case 1. This is the free list after deallocation. The shading indicates the location where changes were made to indicate the free memory block (of size 205).

Using the deallocation algorithm, the system sees that the memory to be released is next to a free memory block, which starts at location 7800. Therefore, the list must be changed to reflect the starting address of the new free block, 7600, which was the address of the first instruction of the job that just released this block. In addition, the memory block size for this new free space must be changed to show its new size, which is the combined total of the two free partitions (200 + 5).

Case 2: Joining Three Free Blocks

When the deallocated memory space is between two free memory blocks, the process is similar, as shown in Table 2.6.

Using the deallocation algorithm, the system learns that the memory to be deallocated is between two free blocks of memory. Therefore, the sizes of the three free partitions (20 + 20 + 205) must be combined and the total stored with the smallest beginning address, 7560.

Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
*7560	20	Free
(7580)	(20)	(Busy)[1]
*7600	205	Free
10250	4050	Free
15125	230	Free
24500	1000	Free

[1] Although the numbers in parentheses don't appear in the free list, they have been inserted here for clarity.

(table 2.6)

Case 2. This is the Free List before deallocation. The asterisks indicate the two free memory blocks that are adjacent to the soon-to-be-free memory block.

Because the entry at location 7600 has been combined with the previous entry, we must empty out this entry. We do that by changing the status to **null entry**, with no beginning address and no memory block size, as indicated by an asterisk in Table 2.7. This negates the need to rearrange the list at the expense of memory.

(table 2.7)

Case 2. The free list after a job has released memory. The revised entry is shaded.

Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
7560	245	Free
*		(null entry)
10250	4050	Free
15125	230	Free
24500	1000	Free

Case 3: Deallocating an Isolated Block

The third alternative is when the space to be deallocated is isolated from all other free areas. For this example, we need to know more about how the busy memory list is configured. To simplify matters, let's look at the busy list for the memory area between locations 7560 and 10250. Remember that, starting at 7560, there's a free memory block of 245, so the busy memory area includes everything from location 7805 (7560 + 245) to 10250, which is the address of the next free block. The free list and busy list are shown in Table 2.8 and Table 2.9.

(table 2.8)

Case 3. Original free list before deallocation. The soon-to-be-free memory block (at location 8805) is not adjacent to any blocks that are already free.

Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
7560	245	Free
		(null entry)
10250	4050	Free
15125	230	Free
24500	1000	Free

Beginning Address	Memory Block Size	Status
7805	1000	Busy
*8805	445	Busy
9250	1000	Busy

(table 2.9)

Case 3. Busy memory list before deallocation. The job to be deallocated is of size 445 and begins at location 8805. The asterisk indicates the soon-to-be-free memory block.

Using the deallocation algorithm, the system learns that the memory block to be released is not adjacent to any free blocks of memory; instead it is between two other busy areas. Therefore, the system must search the table for a null entry.

The scheme presented in this example creates null entries in both the busy and the free lists during the process of allocation or deallocation of memory. An example of a null entry occurring as a result of deallocation was presented in Case 2. A null entry in the busy list occurs when a memory block between two other busy memory blocks is returned to the free list, as shown in Table 2.10. This mechanism ensures that all blocks are entered in the lists according to the beginning address of their memory location from smallest to largest.

Beginning Address	Memory Block Size	Status
7805	1000	Busy
*		(null entry)
9250	1000	Busy

(table 2.10)

Case 3. This is the busy list after the job has released its memory. The asterisk indicates the new null entry in the busy list.

When the null entry is found, the beginning memory location of the terminating job is entered in the beginning address column, the job size is entered under the memory block size column, and the status is changed from "null entry" to free to indicate that a new block of memory is available, as shown in Table 2.11.

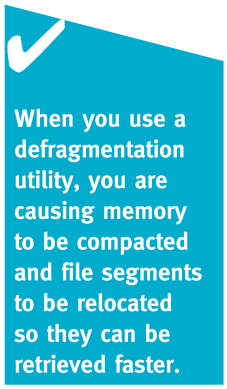
Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
7560	245	Free
*8805	445	Free
10250	4050	Free
15125	230	Free
24500	1000	Free

(table 2.11)

Case 3. This is the free list after the job has released its memory. The asterisk indicates the new free block entry replacing the null entry.

Relocatable Dynamic Partitions

All of the memory allocation schemes described thus far shared some unacceptable fragmentation characteristics that had to be resolved as the number of waiting jobs became unwieldy and demand increased to use all the slivers of memory often left unused.



The solution to both problems was the development of **relocatable dynamic partitions**. With this memory allocation scheme, the Memory Manager relocates programs to gather together all of the empty blocks and compact them to make one block of memory large enough to accommodate some or all of the jobs waiting to get in.

The **compaction of memory**, sometimes referred to as memory defragmentation, is performed by the operating system to reclaim fragmented space. Remember our earlier example of the lending library? If you stopped lending books for a few moments and rearranged the books in the most effective order, you would be compacting your collection. But this demonstrates its disadvantage—this is an overhead process that can take place only while everything else waits.

Compaction isn't an easy task. Most or all programs in memory must be relocated so they're contiguous, and then every address, and every reference to an address, within each program must be adjusted to account for the program's new location in memory. However, all other values within the program (such as data values) must be left alone. In other words, the operating system must distinguish between addresses and data values, and these distinctions are not obvious after the program has been loaded into memory.

To appreciate the complexity of **relocation**, let's look at a typical program. Remember, all numbers are stored in memory as binary values (ones and zeros), and in any given program instruction it's not uncommon to find addresses as well as data values. For example, an assembly language program might include the instruction to add the integer 1 to I. The source code instruction looks like this:

```
ADDI I, 1
```

However, after it has been translated into actual code it could be represented like this (for readability purposes the values are represented here in octal code, not binary code):

```
000007 271 01 0 00 000001
```

Question: Which elements are addresses and which are instruction codes or data values? It's not obvious at first glance. In fact, the address is the number on the left (000007). The instruction code is next (271), and the data value is on the right (000001). Therefore, if this instruction is relocated 200 places, then the address would

be adjusted (added to or subtracted) by 200, but the instruction code and data value would not be.

The operating system can tell the function of each group of digits by its location in the line and the operation code. However, if the program is to be moved to another place in memory, each address must be identified, or flagged. This becomes particularly important when the program includes loop sequences, decision sequences, and branching sequences, as well as data references. If, by chance, every address was not adjusted by the same value, the program would branch to the wrong section of the program or to a part of another program, or it would reference the wrong data.

The program shown in Figure 2.7 and Figure 2.8 shows how the operating system flags the addresses so that they can be adjusted if and when a program is relocated.

Internally, the addresses are marked with a special symbol (indicated in Figure 2.8 by apostrophes) so the Memory Manager will know to adjust them by the value stored in the relocation register. All of the other values (data values) are not marked and won't be changed after relocation. Other numbers in the program, those indicating instructions, registers, or constants used in the instruction, are also left alone.

Figure 2.9 illustrates what happens to a program in memory during compaction and relocation.

This discussion of compaction raises three questions:

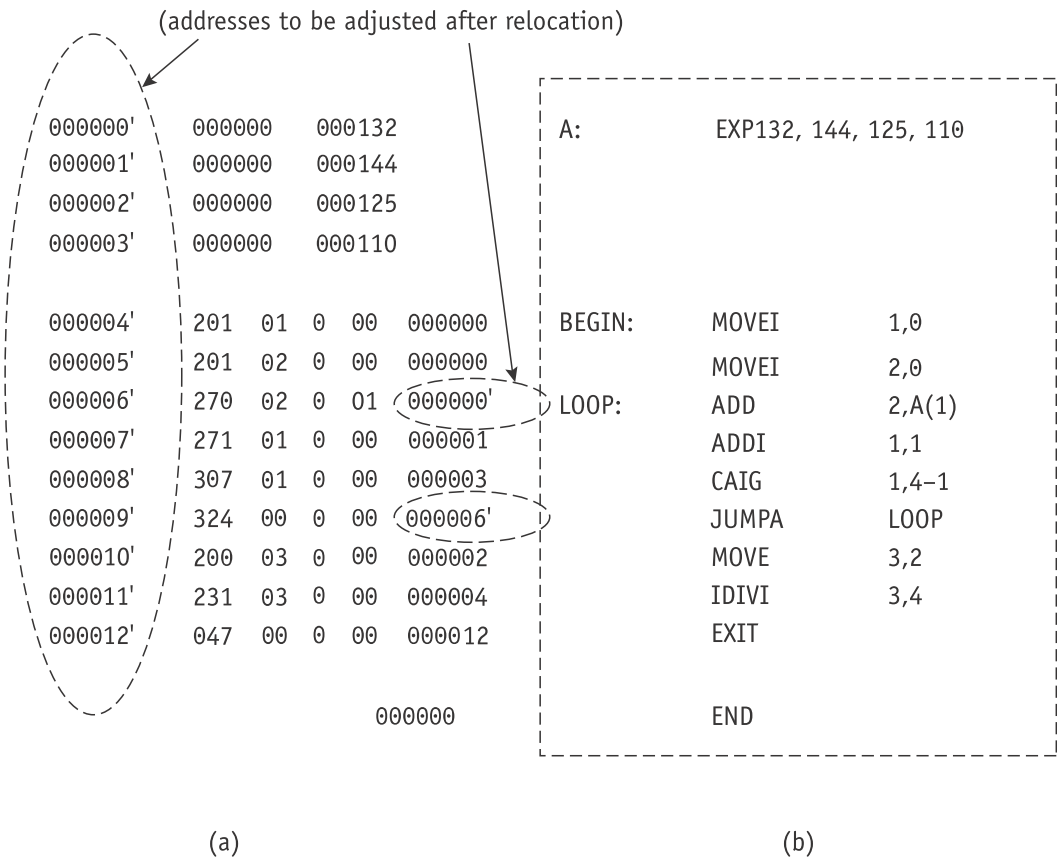
1. What goes on behind the scenes when relocation and compaction take place?
2. What keeps track of how far each job has moved from its original storage area?
3. What lists have to be updated?

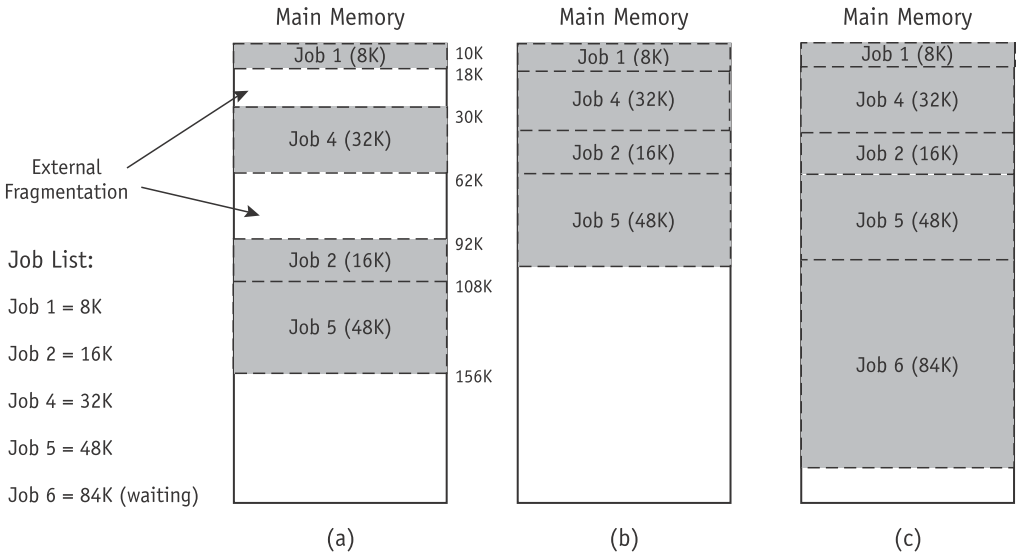
The last question is easiest to answer. After relocation and compaction, both the free list and the busy list are updated. The free list is changed to show the partition for

```
A       EXP 132, 144, 125, 110      ;the data values
BEGIN:  MOVEI           1,0         ;initialize register 1
        MOVEI           2,0         ;initialize register 2
LOOP:   ADD             2,A(1)      ;add (A + reg 1) to reg 2
        ADDI            1,1         ;add 1 to reg 1
        CAIG            1,4-1       ;is register 1 > 4-1?
        JUMPA           LOOP        ;if not, go to Loop
        MOVE            3,2         ;if so, move reg 2 to reg 3
        IDIVI           3,4         ;divide reg 3 by 4,
                                    ;remainder to register 4
        EXIT                        ;end
        END
```

(figure 2.7)

An assembly language program that performs a simple incremental operation. This is what the programmer submits to the assembler. The commands are shown on the left and the comments explaining each command are shown on the right after the semicolons.





(figure 2.8)

The original assembly language program after it has been processed by the assembler, shown on the right (a). To run the program, the assembler translates it into machine readable code (b) with all addresses marked by a special symbol (shown here as an apostrophe) to distinguish addresses from data values. All addresses (and no data values) must be adjusted after relocation.

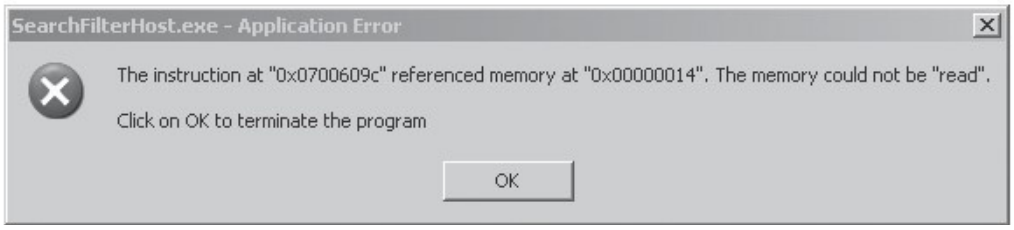
(figure 2.9)

Three snapshots of memory before and after compaction with the operating system occupying the first 10K of memory. When Job 6 arrives requiring 84K, the initial memory layout in (a) shows external fragmentation totaling 96K of space. Immediately after compaction (b), external fragmentation has been eliminated, making room for Job 6 which, after loading, is shown in (c).

the new block of free memory: the one formed as a result of compaction that will be located in memory starting after the last location used by the last job. The busy list is changed to show the new locations for all of the jobs already in progress that were relocated. Each job will have a new address except for those that were already residing at the lowest memory locations. For example, if the entry in the busy list starts at location 7805 (as shown in Table 2.9) and is relocated 762 spaces, then the location would be changed to 7043 (but the memory block size would not change—only its location).

To answer the other two questions we must learn more about the hardware components of a computer—specifically the registers. Special-purpose registers are used to help with the relocation. In some computers, two special registers are set aside for this purpose: the bounds register and the relocation register.

The **bounds register** is used to store the highest (or lowest, depending on the specific system) location in memory accessible by each program. This ensures that during execution, a program won't try to access memory locations that don't belong to it—that is, those that are out of bounds. The result of one such instance is reflected in the error message shown in Figure 2.10.



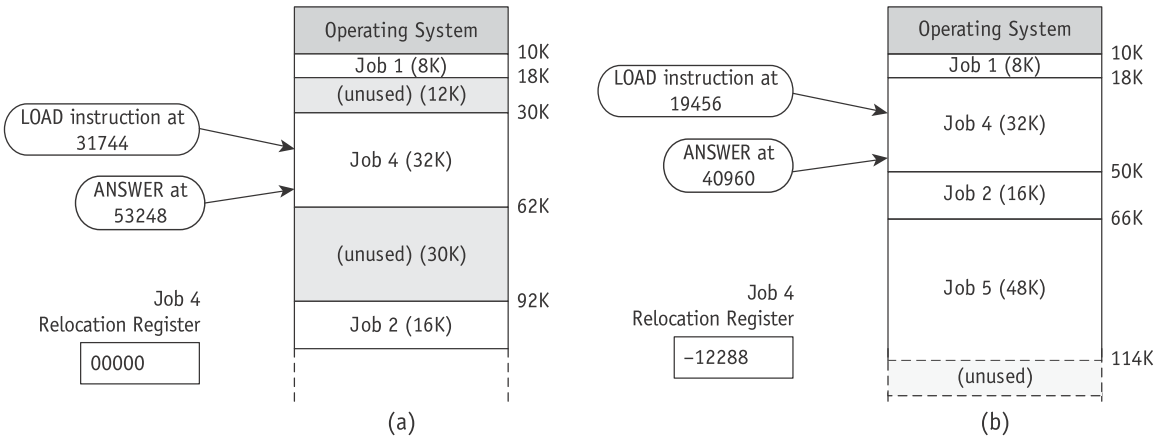
(figure 2.10)

This message indicates that a program attempted to access memory space that is out of bounds. This was a fatal error resulting in the program's termination.

The **relocation register** contains the value that must be added to each address referenced in the program so that the system will be able to access the correct memory addresses after relocation. If a program isn't relocated, the value stored in the program's relocation register is zero. Figure 2.11 illustrates what happens during relocation by using the relocation register (all values are shown in decimal form).

Originally, Job 4 was loaded into memory starting at memory location 30K. (1K equals 1,024 bytes. Therefore, the exact starting address is: 30 * 1024 = 30720.) It required a block of memory of 32K (or 32 * 1024 = 32,768) addressable locations. Therefore, when it was originally loaded, the job occupied the space from memory location 30720 to memory location 63488-1.

Now, suppose that within the program, at memory location 31744, there's this instruction: LOAD 4, ANSWER. (This assembly language command asks that the data value



(figure 2.11)

Contents of the relocation register for Job 4 before the job's relocation and compaction (a) and after (b).

known as ANSWER be loaded into Register 4 for later computation. In this example, Register 4 is a working/computation register, which is distinct from either the relocation or the bounds registers mentioned earlier.) Similarly, the variable known as ANSWER, the value 37, is stored at memory location 53248 before compaction and relocation.

After relocation, Job 4 resides at a new starting memory address of 18K (to be exact, it is 18432, which is 18 * 1024). The job still has the same number of addressable locations, but those locations are in a different physical place in memory. Job 4 now occupies memory from location 18432 to location 51200-1 and, thanks to the relocation register, all of the addresses will be adjusted accordingly when the program is executed.

What does the relocation register contain? In this example, it contains the value –12288. As calculated previously, 12288 is the size of the free block that has been moved toward the high addressable end of memory where it can be combined with other free blocks of memory. The sign is negative because Job 4 has been moved back, closer to the low addressable end of memory, as shown at the top of Figure 2.10(b).

If the addresses were not adjusted by the value stored in the relocation register, then even though memory location 31744 is still part of Job 4's accessible set of memory locations, it would not contain the LOAD command. Not only that, but the other location, 53248, would be out of bounds. The instruction that was originally at 31744 has been moved to location 19456. That's because all of the instructions in this program have been moved back by 12K (12 * 1024 = 12,288), which is the size of the free block. Therefore, location 53248 has been displaced by –12288 and ANSWER, the data value 37, is now located at address 40960.

However, the precise LOAD instruction that was part of the original job has not been changed. The instructions inside the job are not revised in any way. And the original address where ANSWER had been stored, 53248, remains unchanged in the program no matter how many times Job 4 is relocated. Before the instruction is executed, the true address is computed by adding the value stored in the relocation register to the address found at that instruction. By changing only the value in the relocation register each time the job is moved in memory, this technique allows every job address to be properly adjusted.

In effect, by compacting and relocating, the Memory Manager optimizes the use of memory and thus improves throughput—an important measure of system performance. An unfortunate side effect is that this memory allocation scheme requires more overhead than with the previous schemes. The crucial factor here is the timing of the compaction—when and how often it should be done. There are three options.

- One approach is to perform compaction when a certain percentage of memory becomes busy—say 75 percent. The disadvantage of this approach is that the system would incur unnecessary overhead if no jobs were waiting to use the remaining 25 percent.
- A second approach is to compact memory only when there are jobs waiting to get in. This would entail constant checking of the entry queue, which might result in unnecessary overhead and thus slow down the processing of jobs already in the system.
- A third approach is to compact memory after a prescribed amount of time has elapsed. If the amount of time chosen is too small, however, then the system will spend more time on compaction than on processing. If it's too large, too many jobs will congregate in the waiting queue and the advantages of compaction are lost.

Each option has its good and bad points. The best choice for any system is decided by the operating system designers who, based on the job mix and other factors, try to optimize both processing time and memory use while keeping overhead as low as possible.

Conclusion

Four memory management techniques were presented in this chapter: single-user systems, fixed partitions, dynamic partitions, and relocatable dynamic partitions. Although they have differences, they all share the requirement that the entire program (1) be loaded into memory, (2) be stored contiguously, and (3) remain in memory until the entire job is completed. Consequently, each puts severe restrictions on the size of executable jobs because each one can be only as large as the biggest partition in memory. These schemes laid the groundwork for more complex memory management techniques that allowed users to interact more directly and more often with the system.

In the next chapter, we examine several memory allocation schemes that had two new things in common. First, they removed the requirement that jobs be loaded into contiguous memory locations. This allowed jobs to be subdivided so they could be loaded anywhere in memory (if space allowed) as long as the pieces were linked to each other in some way. The second requirement no longer needed was that the entire job reside in memory throughout its execution. By eliminating these mandates, programs were able to grow in size and complexity, opening the door to virtual memory.

Key Terms

address: a number that designates a particular memory location.

best-fit memory allocation: a main memory allocation scheme that considers all free blocks and selects for allocation the one that will result in the least amount of wasted space.

bounds register: a register used to store the highest location in memory legally accessible by each program.

compaction of memory: the process of collecting fragments of available memory space into contiguous blocks by relocating programs and data in a computer's memory.

deallocation: the process of freeing an allocated resource, whether memory space, a device, a file, or a CPU.

dynamic partitions: a memory allocation scheme in which jobs are given as much memory as they request when they are loaded for processing, thus creating their own partitions in main memory.

external fragmentation: a situation in which the dynamic allocation of memory creates unusable fragments of free memory between blocks of busy, or allocated, memory.

first-fit memory allocation: a main memory allocation scheme that searches from the beginning of the free block list and selects for allocation the first block of memory large enough to fulfill the request.

fixed partitions: a memory allocation scheme in which main memory is sectioned with one partition assigned to each job.

internal fragmentation: a situation in which a partition is only partially used by the program; the remaining space within the partition is unavailable to any other job and is therefore wasted.

main memory: the unit that works directly with the CPU and in which the data and instructions must reside in order to be processed. Also called *random access memory (RAM)*, *primary storage*, or *internal memory*.

null entry: an empty entry in a list.

RAM: another term for *main memory*.

relocatable dynamic partitions: a memory allocation scheme in which the system relocates programs in memory to gather together all empty blocks and compact them to make one block of memory that's large enough to accommodate some or all of the jobs waiting for memory.

relocation: (1) the process of moving a program from one area of memory to another; or (2) the process of adjusting address references in a program, by either software or hardware means, to allow the program to execute correctly when loaded in different sections of memory.

relocation register: a register that contains the value that must be added to each address referenced in the program so that the Memory Manager will be able to access the correct memory addresses.

Interesting Searches

For more background on a few of the topics discussed in this chapter, begin a search with these terms.

- Core Memory Technology
- technikum29 Museum of Computer and Communication Technology
- How RAM Works
- Static vs. Dynamic Partitions
- Internal vs. External Fragmentation

Exercises

Research Topics

A. Identify a computer (a desktop or laptop computer) that can accommodate an upgrade or addition of internal memory. Research the brand or type of memory chip that could be used in that system. Do not perform the installation. Instead, list the steps you would take to install the memory. Be sure to cite your sources.

B. For a platform of your choice, research the growth in the size of main memory from the time the platform was developed to the present day. Create a chart showing milestones in memory growth and the approximate date. Choose one from laptops, desktops, midrange computers, and mainframes. Be sure to mention the organization that performed the research and cite your sources.

Exercises

1. Describe a present-day computing environment that might use each of the memory allocation schemes (single user, fixed, dynamic, and relocatable dynamic) described in the chapter. Defend your answer by describing the advantages and disadvantages of the scheme in each case.
2. How often should memory compaction/relocation be performed? Describe the advantages and disadvantages of performing it even more often than recommended.
3. Using your own words, explain the role of the bounds register.
4. Describe in your own words the role of the relocation register.
5. Give an example of computing circumstances that would favor first-fit allocation over best-fit. Explain your answer.
6. Given the following information:

Job List:		**Memory Block List:**	
Job Number	**Memory Requested**	**Memory Block**	**Memory Block Size**
Job A	690K	Block 1	900K (low-order memory)
Job B	275K	Block 2	910K
Job C	760K	Block 3	300K (high-order memory)

   a. Use the first-fit algorithm to indicate which memory blocks are allocated to each of the three arriving jobs.
   b. Use the best-fit algorithm to indicate which memory blocks are allocated to each of the three arriving jobs.

7. Given the following information:

Job List:		**Memory Block List:**	
Job Number	**Memory Requested**	**Memory Block**	**Memory Block Size**
Job A	57K	Block 1	900K (low-order memory)
Job B	920K	Block 2	910K
Job C	50K	Block 3	200K
Job D	701K	Block 4	300K (high-order memory)

   a. Use the best-fit algorithm to indicate which memory blocks are allocated to each of the three arriving jobs.
   b. Use the first-fit algorithm to indicate which memory blocks are allocated to each of the three arriving jobs.

8. Next-fit is an allocation algorithm that starts by using the first-fit algorithm but keeps track of the partition that was last allocated. Instead of restarting the search with Block 1, it starts searching from the most

recently allocated block when a new job arrives. Using the following information:

Job List:		Memory Block List:	
Job Number	**Memory Requested**	**Memory Block**	**Memory Block Size**
Job A	590K	Block 1	100K (low-order memory)
Job B	50K	Block 2	900K
Job C	275K	Block 3	280K
Job D	460K	Block 4	600K (high-order memory)

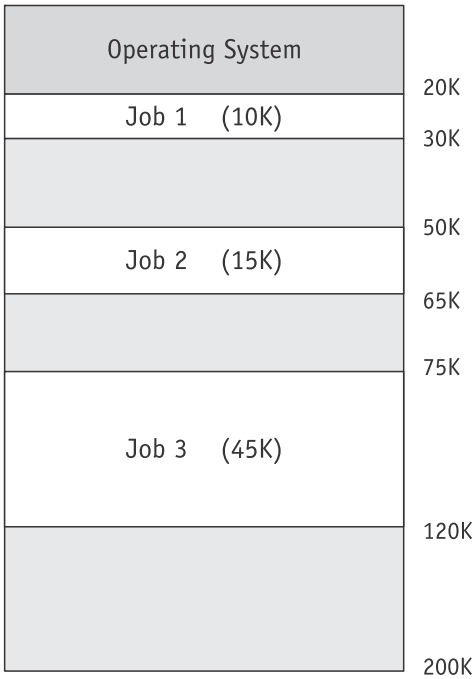
Indicate which memory blocks are allocated to each of the three arriving jobs, and explain in your own words what advantages the next-fit algorithm could offer.

9. Worst-fit is an allocation algorithm that allocates the largest free block to a new job. It is the opposite of the best-fit algorithm. Compare it to next-fit conditions given in Exercise 8 and explain in your own words what advantages the worst-fit algorithm could offer.
10. Imagine an operating system that cannot perform memory deallocation. Name at least three effects on overall system performance that might result and explain your answer.
11. In a system using the relocatable dynamic partitions scheme, given the following situation (and using decimal form): Job Q is loaded into memory starting at memory location 42K.
    a. Calculate the exact starting address for Job Q in bytes.
    b. If the memory block has 3K in fragmentation, calculate the size of the memory block.
    c. Is the resulting fragmentation internal or external? Explain your reasoning.
12. In a system using the relocatable dynamic partitions scheme, given the following situation (and using decimal form): Job W is loaded into memory starting at memory location 5000K.
    a. Calculate the exact starting address for Job W in bytes.
    b. If the memory block has 3K in fragmentation, calculate the size of the memory block.
    c. Is the resulting fragmentation internal or external? Explain your reasoning.
13. If the relocation register holds the value -83968, was the relocated job moved toward the lower or higher addressable end of main memory? By how many kilobytes was it moved? Explain your conclusion.
14. In a system using the fixed partitions memory allocation scheme, given the following situation (and using decimal form): After Job J is loaded into a partition of size 50K, the resulting fragmentation is 7168 bytes. Perform the following:
    a. What is the size of Job J in bytes?
    b. What type of fragmentation is caused?

15. In a system using the dynamic partition memory allocation scheme, given the following situation (and using decimal form): After Job C of size 70K is loaded into a partition resulting in 7K of fragmentation, calculate the size (in bytes) of its partition and identify the type of fragmentation that is caused. Explain your answer.

Advanced Exercises

16. The relocation example presented in the chapter implies that compaction is done entirely in memory, without secondary storage. Can all free sections of memory be merged into one contiguous block using this approach? Why or why not?
17. In this chapter we described one way to compact memory. Some people suggest an alternate method: all jobs could be copied to a secondary storage device and then reloaded (and relocated) contiguously into main memory, thus creating one free block after all jobs have been recopied into memory. Is this viable? Could you devise a better way to compact memory? Write your algorithm and explain why it is better.
18. Given the memory configuration in Figure 2.12, answer the following questions. At this point, Job 4 arrives requesting a block of 100K.
    a. Can Job 4 be accommodated? Why or why not?
    b. If relocation is used, what are the contents of the relocation registers for Job 1, Job 2, and Job 3 after compaction?
    c. What are the contents of the relocation register for Job 4 after it has been loaded into memory?



(figure 2.12)

Memory configuration for Exercise 18.

    d. An instruction that is part of Job 1 was originally loaded into memory location 22K. What is its new location after compaction?
    e. An instruction that is part of Job 2 was originally loaded into memory location 55K. What is its new location after compaction?
    f. An instruction that is part of Job 3 was originally loaded into memory location 80K. What is its new location after compaction?
    g. If an instruction was originally loaded into memory location 110K, what is its new location after compaction?

Programming Exercises

19. Here is a long-term programming project. Use the information that follows to complete this exercise.

Job List

Job Stream Number	Time	Job Size
1	5	5760
2	4	4190
3	8	3290
4	2	2030
5	2	2550
6	6	6990
7	8	8940
8	10	740
9	7	3930
10	6	6890
11	5	6580
12	8	3820
13	9	9140
14	10	420
15	10	220
16	7	7540
17	3	3210
18	1	1380
19	9	9850
20	3	3610
21	7	7540
22	2	2710
23	8	8390
24	5	5950
25	10	760

Memory List

Memory Block	Size
1	9500
2	7000
3	4500
4	8500
5	3000
6	9000
7	1000
8	5500
9	1500
10	500

At one large batch-processing computer installation, the management wants to decide what storage placement strategy will yield the best possible performance. The installation runs a large real storage computer (as opposed to "virtual" storage, which is covered in Chapter 3) under fixed partition multiprogramming. Each user program runs in a single group of contiguous storage locations. Users state their storage requirements and time units for CPU usage on their Job Control Card (it used to, and still does, work this way, although cards may not be used). The operating system allocates to each user the appropriate partition and starts up the user's job. The job remains in memory until completion. A total of 50,000 memory locations are available, divided into blocks as indicated in the previous table.

a. Write (or calculate) an event-driven simulation to help you decide which storage placement strategy should be used at this installation. Your program would use the job stream and memory partitioning as indicated previously. Run the program until all jobs have been executed with the memory as is (in order by address). This will give you the first-fit type performance results.

b. Sort the memory partitions by size and run the program a second time; this will give you the best-fit performance results. For both parts a. and b., you are investigating the performance of the system using a typical job stream by measuring:

    1. Throughput (how many jobs are processed per given time unit)
    2. Storage utilization (percentage of partitions never used, percentage of partitions heavily used, and so on)
    3. Waiting queue length
    4. Waiting time in queue
    5. Internal fragmentation

Given that jobs are served on a first-come, first-served basis:

c. Explain how the system handles conflicts when jobs are put into a waiting queue and there are still jobs entering the system—which job goes first?

d. Explain how the system handles the "job clocks," which keep track of the amount of time each job has run, and the "wait clocks," which keep track of how long each job in the waiting queue has to wait.

e. Since this is an event-driven system, explain how you define "event" and what happens in your system when the event occurs.

f. Look at the results from the best-fit run and compare them with the results from the first-fit run. Explain what the results indicate about the performance of the system for this job mix and memory organization. Is one method of partitioning better than the other? Why or why not? Could you recommend one method over the other given your sample run? Would this hold in all cases? Write some conclusions and recommendations.

20. Suppose your system (as explained in Exercise 19) now has a "spooler" (a storage area in which to temporarily hold jobs), and the job scheduler can choose which will be served from among 25 resident jobs. Suppose also that the first-come, first-served policy is replaced with a "faster-job, first-served" policy. This would require that a sort by time be performed on the job list before running the program. Does this make a difference in the results? Does it make a difference in your analysis? Does it make a difference in your conclusions and recommendations? The program should be run twice to test this new policy with both best-fit and first-fit.
21. Suppose your spooler (as described in Exercise 19) replaces the previous policy with one of "smallest-job, first-served." This would require that a sort by job size be performed on the job list before running the program. How do the results compare to the previous two sets of results? Will your analysis change? Will your conclusions change? The program should be run twice to test this new policy with both best-fit and first-fit.