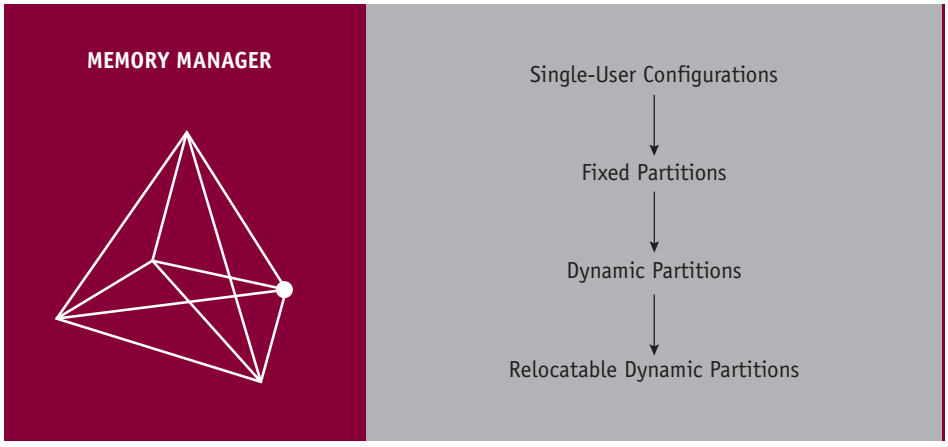
Chapter 2 Memory Management: Early Systems



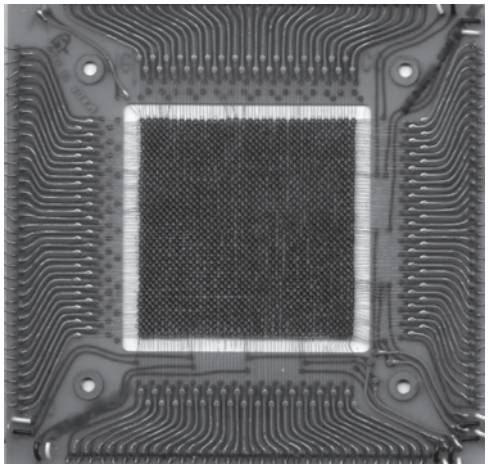
"Memory is the primary and fundamental power, without which there could be no other intellectual operation."

—Samuel Johnson *(1709–1784)*

Learning Objectives

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

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



(figure 2.1)
Main memory circuit from 1961 (before they became too small to see without magnification).

Courtesy of technikum29

The management of **main memory** is critical. In fact, from a historical perspective, the performance of the *entire* system has been directly dependent on two things: How much memory is available and how it is optimized while jobs are being processed. Pictured in Figure 2.1 is a main memory circuit from 1961. Since then, the physical size of memory units has become increasingly small and they are now available on small boards.

This chapter introduces the Memory Manager (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.

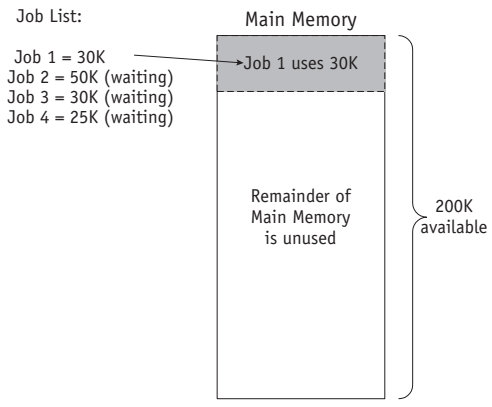
These early memory management schemes are seldom used by today's operating systems, but they are important to study because each one introduced fundamental concepts that helped memory management evolve, as shown in Chapter 3, "Memory Management: Virtual Memory," which discusses memory allocation strategies for Linux. Information on how other operating systems manage memory is presented in the memory management sections in Part Two of the text.

Let's start with the simplest memory management scheme—the one used in the earliest generations of computer systems.

Single-User Contiguous Scheme

The first memory allocation scheme worked like this: Each program to be processed was loaded in its entirety into memory and allocated as much contiguous space in memory as it needed, as shown in Figure 2.2. The key words here are *entirety* and *contiguous*. If the program was too large and didn't fit the available memory space, it couldn't be executed. And, although early computers were physically large, they had very little memory.

A single-user scheme supports one user on one computer running one job at a time. Sharing isn't possible.



(figure 2.2) *One program fit in memory at a time. The remainder of memory was unused.*

This demonstrates a significant limiting factor of all computers—they have only a finite amount of memory and if a program doesn't fit, then either the size of the main memory must be increased or the program must be modified. It's usually modified by making it smaller or by using methods that allow program segments (partitions made to the program) to be overlaid. (To overlay is to transfer segments of a program from secondary storage into main memory for execution, so that two or more segments take turns occupying the same memory locations.)

Single-user systems in a nonnetworked environment work the same way. Each user is given access to all available main memory for each job, and jobs are processed sequentially, one after the other. To allocate memory, the operating system uses a simple algorithm (step-by-step procedure to solve a problem):

Algorithm to Load a Job in a Single-User System

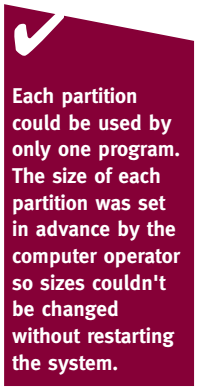
1 Store first memory location of program into base register (for memory protection)
2 Set program counter (it keeps track of memory space used by the program) equal to address of first memory location
3 Read first instruction of program
4 Increment program counter by number of bytes in instruction
5 Has the last instruction been reached?
  if yes, then stop loading program
  if no, then continue with step 6
6 Is program counter greater than memory size?
  if yes, then stop loading program
  if no, then continue with step 7
7 Load instruction in memory
8 Read next instruction of program
9 Go to step 4

Notice that the amount of work done by the operating system's Memory Manager is minimal, the code to perform the functions is straightforward, and the logic is quite simple. Only two hardware items are needed: a register to store the base **address** and an accumulator to keep track of the size of the program as it's being read into memory. Once the program is entirely loaded into memory, it remains there until execution is complete, either through normal termination or by intervention of the operating system.

One major problem with this type of memory allocation scheme is that it doesn't support multiprogramming or networking (both are discussed later in this text); it can handle only one job 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. Therefore, in the late 1950s and early 1960s a new scheme was needed to manage memory, which used partitions to take advantage of the computer system's resources by overlapping independent operations.

Fixed Partitions

The first attempt to allow for multiprogramming used **fixed partitions** (also called **static partitions**) within the main memory—one partition for each job. Because the size of each partition was designated when the system was powered on, each partition could only be reconfigured when the computer system was shut down, reconfigured, and restarted. Thus, once the system was in operation the partition sizes remained static.



A critical factor was introduced with this scheme: protection of the job's memory space. Once a partition was assigned to a job, no other job could be allowed to enter 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 so only the portion of the operating system residing in main memory had to be protected. However, for the fixed partition allocation schemes, protection was mandatory for each partition present in main memory. Typically this was the joint responsibility of the hardware of the computer and 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 partition to make sure it fits completely. Then, when a block of sufficient size is located, the status of the partition must be checked to see if it's available.

Algorithm to Load a Job in a Fixed Partition

```
1 Determine job's requested memory size
2 If job_size > size of largest partition
    Then reject the job
      print appropriate message to operator
      go to step 1 to handle next job in line
    Else
      continue with step 3
3 Set counter to 1
4 Do while counter <= number of partitions in memory
  If job_size > memory_partition_size(counter)
  Then counter = counter + 1
  Else
    If memory_partition_size(counter) = "free"
      Then load job into memory_partition(counter)
        change memory_partition_status(counter) to "busy"
        go to step 1 to handle next job in line
    Else
      counter = counter + 1
    End do
5 No partition available at this time, put job in waiting queue
6 Go to step 1 to handle next job in line
```

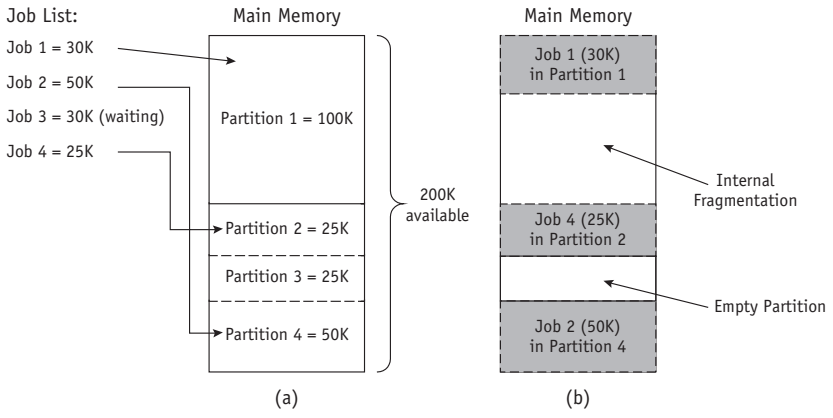
This partition scheme is more flexible than the single-user scheme because it allows several programs to be in memory at the same time. However, it still requires that the *entire* program be stored *contiguously* and *in memory* from the beginning to the end of its execution. In order to allocate memory spaces to jobs, the operating system's Memory Manager must keep a table, such as Table 2.1, which shows each memory partition size, its address, its access restrictions, and its current status (free or busy) for the system illustrated in Figure 2.3. (In Table 2.1 and the other tables in this chapter, K stands for kilobyte, which is 1,024 bytes. A more in-depth discussion of memory map tables is presented in Chapter 8, "File Management.")

(table 2.1)

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

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

As each job terminates, the status of its memory partition is changed from busy to free so an incoming job can be assigned to that partition.



(figure 2.3)

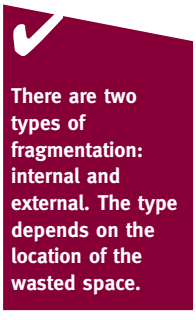
Main memory use during fixed partition allocation of Table 2.1. Job 3 must wait even though 70K of free space is available in Partition 1, where Job 1 only occupies 30K of the 100K available. The jobs are allocated space on the basis of "first available partition of required size."

The fixed partition scheme works well if all of the jobs run on the system are of the same 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 to be run on the system in the coming hours, days, or weeks. However, unless the operator can accurately predict the future, the sizes of the partitions are determined in an arbitrary fashion and they might be too small or too large for the jobs coming in.

There are significant consequences if the partition sizes are too small; larger jobs will be rejected if they're too big to fit into the largest partitions or will wait if the large partitions are busy. As a result, large jobs may have a longer turnaround time as they wait for free partitions of sufficient size or may never run.

On the other hand, if the partition sizes are too big, memory is wasted. If a job does not occupy the entire partition, the unused memory in the partition will remain idle; it can't be given to another job because each partition is allocated to only one job at a time. It's an indivisible unit. Figure 2.3 demonstrates one such circumstance.

This phenomenon of partial usage of fixed partitions and the coinciding creation of unused spaces within the partition is called **internal fragmentation**, and is a major drawback to the fixed partition memory allocation scheme.

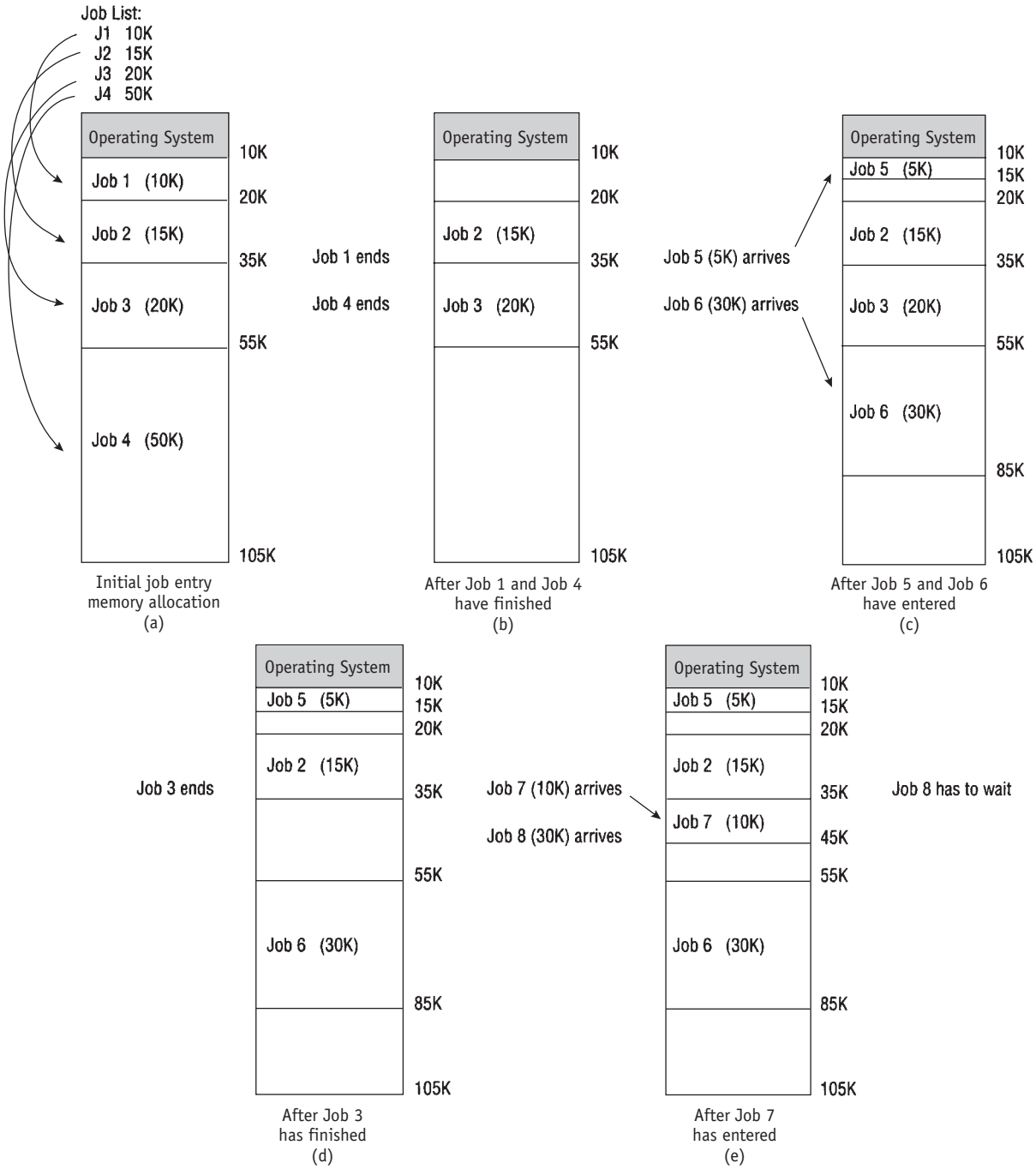


There are two types of fragmentation: internal and external. The type depends on the location of the wasted space.

Dynamic Partitions

With **dynamic partitions**, available memory is still kept in contiguous blocks but jobs are given only as much memory as they request when they are loaded for processing. Although this is a significant improvement over fixed partitions because memory isn't wasted within the partition, it doesn't entirely eliminate the problem.

As shown in Figure 2.4, a dynamic partition scheme fully utilizes memory when the first jobs are loaded. But as new jobs enter the system that are not the same size as those that



(figure 2.4)

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.

just vacated memory, they are fit into the available spaces on a priority basis. Figure 2.4 demonstrates **first-come, first-served** priority. Therefore, the subsequent allocation of memory creates fragments of free memory between blocks of allocated memory. This problem is called **external fragmentation** and, like internal fragmentation, lets memory go to waste.

In the last snapshot, (e) in Figure 2.4, there are three free partitions of 5K, 10K, and 20K—35K in all—enough to accommodate Job 8, which only requires 30K. However they are not contiguous and, because the jobs are loaded in a contiguous manner, this scheme forces Job 8 to wait.

Before we go to the next allocation scheme, let's examine how the operating system keeps track of the free sections of memory.

Best-Fit Versus 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.

These partitions may be allocated on the basis of **first-fit memory allocation** (first partition fitting the requirements) or **best-fit memory allocation** (least wasted space, the smallest partition fitting the requirements). For both schemes, the Memory Manager organizes the memory lists of the free and used partitions (free/busy) either by size or by location. The best-fit allocation method keeps the free/busy lists in order by size, smallest to largest. The first-fit method keeps the free/busy lists organized by memory locations, 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 in making the allocation.

To understand the trade-offs, imagine that you've turned your collection of books into a lending library. Let's say you have books of all shapes and sizes, and let's also say that there's a continuous stream of people taking books out and bringing them back—someone's always waiting. It's clear that you'll always be busy, and that's good, but you never have time to rearrange the bookshelves.

You need a system. Your shelves have fixed partitions with a few tall spaces for oversized books, several shelves for paperbacks, 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.

So as each book is removed from its 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. 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 of allocation.

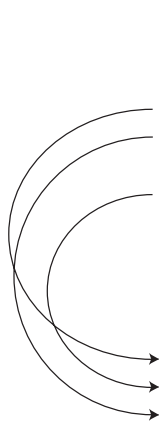
If the lists are organized by size, you're optimizing your shelf space—as books arrive, you'll be able to put them in the spaces that fit them best. This is a best-fit scheme. If a paperback is returned, you'll place it on a shelf with the other paperbacks or at least with other small books. Similarly, 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. This is 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—even an encyclopedia rack can be used if it's close to your desk because you are optimizing the time it takes you to reshelve the books.

Of course, this is a fast method of shelving books, and if speed is important it's the best 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, if all of the fragments of memory were added together, there would be more than enough room to accommodate it. 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:

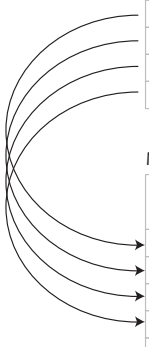
Memory location	Memory block size	Job number	Job size	Status	Internal fragmentation
10240	30K	J1	10K	Busy	20K
40960	15K	J4	10K	Busy	5K
56320	50K	J2	20K	Busy	30K
107520	20K			Free	
Total Available:	115K	Total Used:	40K		

(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 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 moves out of the loop to fetch 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.



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
40960	15K	J1	10K	Busy	5K
107520	20K	J2	20K	Busy	None
10240	30K	J3	30K	Busy	None
56320	50K	J4	10K	Busy	40K
Total Available:	115K	Total Used:	70K		

(figure 2.6)

Best-fit free scheme. Job 1 is allocated to the closest-fitting 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.

The algorithms for best-fit and first-fit are very different. Here's how first-fit is implemented:

First-Fit Algorithm

```
1 Set counter to 1
2 Do while counter <= number of blocks in memory
    If job_size > memory_size(counter)
      Then counter = counter + 1
    Else
      load job into memory_size(counter)
      adjust free/busy memory lists
      go to step 4
  End do
3 Put job in waiting queue
4 Go fetch next job
```

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 algorithm 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 a request is made using the first-fit algorithm. (Note: All values are in decimal notation unless otherwise indicated.)

Before Request		After Request	
Beginning Address	**Memory Block Size**	**Beginning Address**	**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 algorithm for best-fit is slightly more complex because the goal is to find the smallest memory block into which the job will fit:

Best-Fit Algorithm

```
1 Initialize memory_block(0) = 99999
2 Compute initial_memory_waste = memory_block(0) – job_size
3 Initialize subscript = 0
4 Set counter to 1
5 Do while counter <= number of blocks in memory
     If job_size > memory_size(counter)
        Then counter = counter + 1
     Else
        memory_waste = memory_size(counter) – job_size
   If initial_memory_waste > memory_waste
        Then subscript = counter
        initial_memory_waste = memory_waste
        counter = counter + 1
   End do
6 If subscript = 0
   Then put job in waiting queue
   Else
        load job into memory_size(subscript)
        adjust free/busy memory lists
7 Go fetch next job
```

One of the problems with the best-fit algorithm is that the entire table must be searched before the 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 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.

(table 2.3)

These two snapshots of memory show the status of each memory block before and after a request is made using the best-fit algorithm.

Before Request		After Request	
Beginning Address	**Memory Block Size**	**Beginning Address**	**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

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 smallest block that's 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).

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. Note that while the best-fit resulted in a better fit, it also resulted (and does so in the general case) in a smaller free space (5 spaces), which is known as a sliver.

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, which allocates the largest free available block to the new job. Worst-fit is the opposite of best-fit. Although it's a good way to explore the theory of memory allocation, it might not be the best choice for an actual system.

In recent years, access times have become so fast that the scheme that saves the more valuable resource, memory space, may be the best in some cases. Research continues to focus on finding the optimum allocation scheme. This includes optimum page size—a fixed allocation scheme we will cover in the next chapter, which is the key to improving the performance of the best-fit allocation scheme.

Deallocation

Until now, we've considered only the problem of how memory blocks are allocated, but eventually there comes a time when memory space must be released, or **deallocated**.

For a fixed partition system, the process is quite straightforward. When the job is completed, the Memory Manager resets the status of the memory block where the job was stored 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 because the algorithm 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

The deallocation algorithm must be prepared for all three eventualities with a set of nested conditionals. The following algorithm is based on the fact that memory locations are listed using a lowest-to-highest address scheme. The algorithm would have to be modified to accommodate a different organization of memory locations. In this algorithm, *job_size* is the amount of memory being released by the terminating job, and *beginning_address* is the location of the first instruction for the job.

Algorithm to Deallocate Memory Blocks

```
If job_location is adjacent to one or more free blocks
    Then
    If job_location is between two free blocks
        Then merge all three blocks into one block
        memory_size(counter-1) = memory_size(counter-1) + job_size
        + memory_size(counter+1)
        set status of memory_size(counter+1) to null entry
    Else
        merge both blocks into one
        memory_size(counter-1) = memory_size(counter-1) + job_size
Else
    search for null entry in free memory list
    enter job_size and beginning_address in the entry slot
    set its status to "free"
```

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.

(table 2.4)

This is the original free list before deallocation for Case 1. The asterisk indicates the free memory block that's adjacent to the soon-to-be-free memory block.

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.

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

(table 2.5)

Case 1. This is the free list after deallocation. The asterisk indicates the location where changes were made to the free memory block.

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

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 original 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.

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

(table 2.7)

Case 2. The free list after a job has released memory.

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 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

(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.

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

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 a 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

Both of the fixed and dynamic memory allocation schemes described thus far shared some unacceptable fragmentation characteristics that had to be resolved before the number of jobs waiting to be accepted became unwieldy. In addition, there was a growing need to use all the slivers of memory often left over.

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 garbage collection or defragmentation, is performed by the operating system to reclaim fragmented sections of the memory space. Remember our earlier example of the makeshift 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 disad-

When you use a defragmentation utility, you are compacting memory and relocating file segments so they can be retrieved faster.

vantage—it's an overhead process, so that while compaction is being done everything else must wait.

Compaction isn't an easy task. First, every program 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 the distinctions are not obvious once 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, 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 look like this (for readability purposes the values are represented here in octal code, not binary code):

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

It's not immediately obvious which elements are addresses and which are instruction codes or data values. 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).

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. So later the amount of memory locations by which the program has been displaced must be added to (or subtracted from) all of the original addresses in the program.

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 section of another program, or it would reference the wrong data.

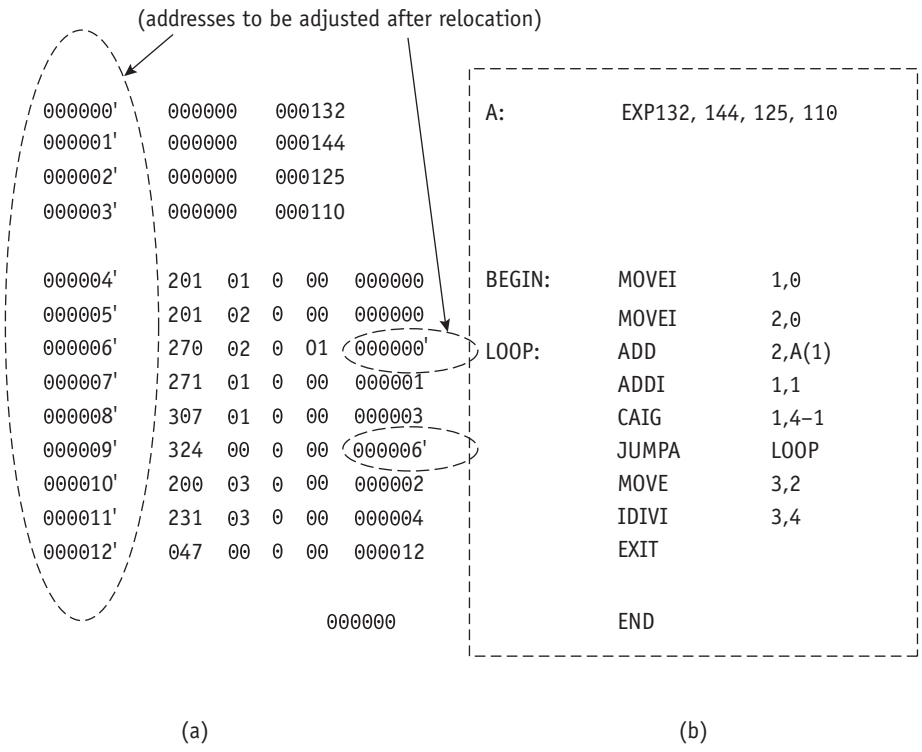
The program in Figure 2.7 and Figure 2.8 shows how the operating system flags the addresses so that they can be adjusted if and when a program is relocated.

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

```
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.

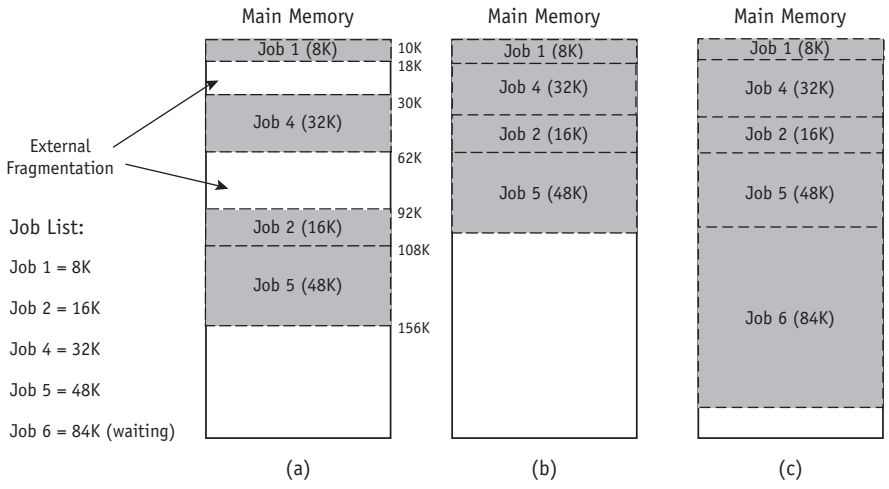


(figure 2.8)

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.

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.



(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).

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?

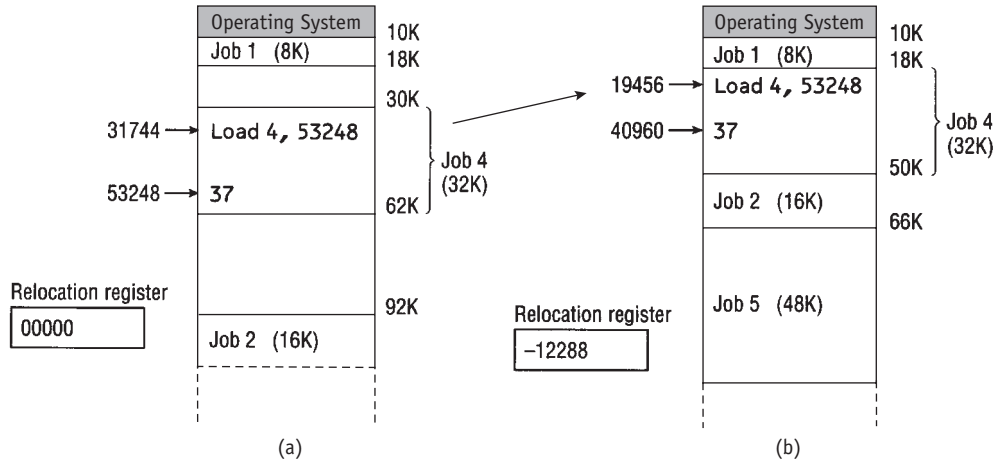
The last question is easiest to answer. After relocation and compaction, both the free list and the busy list are updated. The free list is changed to show the partition for the new block of free memory: the one formed as a result of compaction that will be located in memory starting after the last location 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.

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 **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 the program isn't relocated, the value stored in the program's relocation register is zero.

Figure 2.10 illustrates what happens during relocation by using the relocation register (all values are shown in decimal form).



(figure 2.10)

Contents of relocation register and close-up of Job 4 memory area (a) before relocation and (b) after relocation and compaction.

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 = 30,720.) 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 an instruction that looks like this:

```
LOAD 4, ANSWER
```

This assembly language command asks that the data value known as ANSWER be loaded into Register 4 for later computation. ANSWER, the value 37, is stored at memory location 53248. (In this example, Register 4 is a working/computation register, which is distinct from either the relocation or the bounds register.)

After relocation, Job 4 has been moved to a new starting memory address of 18K (actually 18 * 1024 = 18,432). Of course, the job still has its 32K addressable locations, so it now occupies memory from location 18432 to location 51200-1 and, thanks to the relocation register, all of the addresses will be adjusted accordingly.

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 forward toward the high addressable end 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).

However, the program instruction (LOAD 4, ANSWER) has not been changed. The original address 53248 where ANSWER had been stored remains the same in the program no matter how many times it is relocated. Before the instruction is executed, however, the true address must be computed by adding the value stored in the relocation register to the address found at that instruction. If the addresses are not adjusted by the value stored in the relocation register, then even though memory location 31744 is still part of the job's accessible set of memory locations, it would not contain the LOAD command. Not only that, but location 53248 is now 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.

In effect, by compacting and relocating, the Memory Manager optimizes the use of memory and thus improves throughput—one of the measures of system performance. An unfortunate side effect is that more overhead is incurred than with the two previous memory allocation 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 do it 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 slow down the processing of jobs already in the system.

A third approach is to do it 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.

As you can see, each option has its good and bad points. The best choice for any system is decided by the operating system designer who, based on the job mix and other

factors, tries 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. They have three things in common: They all require that the entire program (1) be loaded into memory, (2) be stored contiguously, and (3) remain in memory until the job is completed.

Consequently, each puts severe restrictions on the size of the jobs because they can only be as large as the biggest partitions in memory.

These schemes were sufficient for the first three generations of computers, which processed jobs in batch mode. Turnaround time was measured in hours, or sometimes days, but that was a period when users expected such delays between the submission of their jobs and pick up of output. As you'll see in the next chapter, a new trend emerged during the third-generation computers of the late 1960s and early 1970s: Users were able to connect directly with the central processing unit via remote job entry stations, loading their jobs from online terminals that could interact more directly with the system. New methods of memory management were needed to accommodate them.

We'll see that the memory allocation schemes that followed had two new things in common. First, programs didn't have to be stored in contiguous memory locations—they could be divided into segments of variable sizes or pages of equal size. Each page, or segment, could be stored wherever there was an empty block big enough to hold it. Second, not all the pages, or segments, had to reside in memory during the execution of the job. These were significant advances for system designers, operators, and users alike.

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: the process of collecting fragments of available memory space into contiguous blocks by moving programs and data in a computer's memory or disk. Also called *garbage collection*.

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 come first served (FCFS): a nonpreemptive process scheduling policy that handles jobs according to their arrival time; the first job in the READY queue is processed first.

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 off, with portions assigned to each job.

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

kilobyte (K): a unit of memory or storage space equal to 1,024 bytes or 2^{10} bytes.

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.

relocatable dynamic partitions: a memory allocation scheme in which the system relocates programs in memory to gather together all of the 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 it will be able to access the correct memory addresses after relocation.

static partitions: another term for *fixed partitions*.

Interesting Searches

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

Exercises

Research Topics

A. Three different number systems (in addition to the familiar base-10 system) are commonly used in computer science. Create a column of integers 1 through 30. In the next three columns show how each value is represented using the binary, octal, and hex number systems. Identify when and why each of the each three numbering systems is used. Cite your sources.

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

Exercises

1. Explain the fundamental differences between internal fragmentation and external fragmentation. For each of the four memory management systems explained in this chapter (single user, fixed, dynamic, and relocatable dynamic), identify which one causes each type of fragmentation.
2. Which type of fragmentation is reduced by compaction? Explain your answer.
3. How often should relocation be performed? Explain your answer.
4. Imagine an operating system that does not perform memory deallocation. Name at least three unfortunate outcomes that would result and explain your answer.
5. Compare and contrast a fixed partition system and a dynamic partition system.
6. Compare and contrast a dynamic partition system and a relocatable dynamic partition system.

7. Given the following information:

Job list:

Job Number	Memory Requested	Memory Block	Memory Block Size
Job 1	690 K	Block 1	900 K (low-order memory)
Job 2	275 K	Block 2	910 K
Job 3	760 K	Block 3	300 K (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. Given the following information:

Job list:

Job Number	Memory Requested	Memory Block	Memory Block Size
Job 1	275 K	Block 1	900 K (low-order memory)
Job 2	920 K	Block 2	910 K
Job 3	690 K	Block 3	300 K (high-order memory)

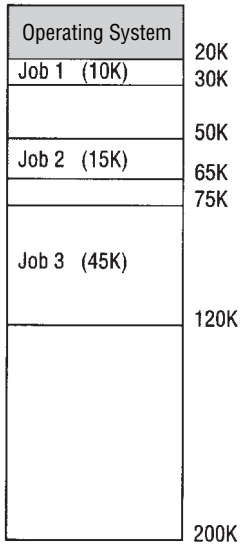
   a. Use the best-fit algorithm to indicate which memory blocks are allocated to each of the three arriving jobs.
   b. Use the first-fit algorithm to indicate which memory blocks are allocated to each of the three arriving jobs.

9. Next-fit is an allocation algorithm that keeps track of the partition that was allocated previously (last) and starts searching from that point on when a new job arrives.
   a. Are there any advantages of the next-fit algorithm? If so, what are they?
   b. How would it compare to best-fit and first-fit for the conditions given in Exercise 7?
   c. How would it compare to best-fit and first-fit for the conditions given in Exercise 8?

10. Worst-fit is an allocation algorithm that allocates the largest free block to a new job. This is the opposite of the best-fit algorithm.
   a. Are there any advantages of the worst-fit algorithm? If so, what are they?
   b. How would it compare to best-fit and first-fit for the conditions given in Exercise 7?
   c. How would it compare to best-fit and first-fit for the conditions given in Exercise 8?

Advanced Exercises

11. 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?
12. To compact memory in some systems, some people suggest that all jobs in memory 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.
13. Given the memory configuration in Figure 2.11, 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?
    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?



(figure 2.11)

Memory configuration for Exercise 13.

Programming Exercises

14. 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 (as opposed to "virtual" storage, which will be covered in the following chapter) computer 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 table on the previous page.

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, etc.)
    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—who 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.

This page intentionally left blank

15. Suppose your system (as explained in Exercise 14) now has a "spooler" (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.

16. Suppose your spooler (as described in Exercise 14) 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.