CHAPTER 7
Physical Memory

7.1 PREPARING A PROGRAM FOR EXECUTION

Most application, OS, and utility programs are written in some user-oriented, high-level language, or less frequently, in assembly language. Each such program must undergo a series of transformation before it can be executed by the machine hardware.

7.1.1 Program Transformations

Figure 7-1 illustrates the stages of a program as it is being prepared for execution. The first step in this series of transformations is a translation of the source modules—the code as written by the programmer—into a machine-specific version in which all symbolic instructions, operands, and addresses are converted into numerical values. In the case of high-level languages this translation process is referred to as compilation. The term assembly is used with lower-level languages that simply provide a more convenient symbolic form of the underlying machine language. The resulting program is called the object module.

Software systems of any substantial size cannot be developed as one monolithic entity. Instead, modules reflecting different functions of the system are designed separately, possibly by different programmers, and then interconnected into one complex system using linking.

At the time a source program is translated into an object module, it contains external references, i.e., operand addresses referring to data or instructions within other modules.
These cannot be resolved at compile time. Instead, the compiler only records where such references occur in the program to facilitate the subsequent task of the linker. For that purpose, the compiler generates an external symbol table for each module, which lists all external references used in that module. The main task to be performed during linking is to resolve the external references. The resulting module is referred to as the load module. As shown in Figure 7-1, each source module is translated into the corresponding object module independently of other modules.

Note that resolving an external reference does not necessarily imply binding or assigning that reference to a physical location. It only implies that a common address space has been assigned to all modules linked together and that all references, internal and external, refer to locations within that common space. Thus, linking translates external symbolic references to corresponding numeric references. These can still be logical addresses, which must be translated into physical addresses before they can be accessed. When all external references are resolved before execution of the program begins, the linking is referred to as static. The linking is dynamic when external references are kept in their symbolic form until execution and are resolved during execution.

The final step in the series of transformations is the actual loading of the load module from secondary memory to main memory. The module can be transferred all at once or different parts of it can be transferred at different times during execution.

7.1.2 Logical-to-Physical Address Binding

Before a program can be executed, it must be loaded into main memory. This requires choosing the physical locations in which the program instruction and data will reside, and adjusting the program references to point to the appropriate locations. For example, store or load instructions must be adjusted such that their address fields refer to the current locations of the operands they are to access. Similarly, branch instructions must contain the current addresses of their destination instructions.

The assignment or binding of actual physical addresses to program instructions and data can take place at different points during a program life cycle. These include the time of program development, compilation, linking, loading, or execution. In the first
four cases, where the binding takes place prior to execution, the binding is called static. If the binding takes place when the program is already running, it is called dynamic.

Until the binding takes place, the programmer, the compiler, the linker, or possibly the loader must work within an assumed logical address space when creating or transforming a module. Typically, symbolic names are used by programmers to refer to data or branch locations, which the compiler, the linker, and the loader transform into numerical values, assuming a hypothetical starting address, such as zero. When the actual physical locations are assigned, all operand and branch addresses within the module must be adjusted according to this assignment. In other words, the program is “moved” from the logical to the physical address space. It is for this reason that the binding of logical addresses to physical address also is referred to as program relocation. In particular, a program is called relocatable or dynamically relocatable if it can be moved into different areas of memory without any transformations. Let us examine the issues of address binding and relocation in more detail.

Static Binding

The earliest possible moment at which physical memory address can be assigned is at the time of writing the program. Some assemblers and compilers permit the specification of an absolute (physical) address thus providing for this possibility. Such programming-time binding, however, is used relatively infrequently, for example, at the lowest kernel level of an OS, in many real-time and embedded systems, or when the programmer must control a special hardware component. In most other cases, symbolic names are used at the programming level, and must be translated into a physical address by the compiler and linker.

To perform the translation, the compiler must be given the starting address of the program. If this is the actual physical address, the resulting program can execute only when loaded into the specific preassigned memory space; it is not relocatable. Such early compile-time binding of programs to physical memory locations is very restrictive and thus rarely used. It is more practical to postpone the binding until at least the time of linking. At this time, all modules to be linked together are known, permitting both internal and external references to be resolved.

With link-time binding, the load module is still not relocatable, i.e., it must be loaded into main memory starting with the location assumed by the linker. Consequently, the task of the loader is simply to copy the load module into memory, without any further transformations. In this case, the linker and the loader may usefully be combined into one subsystem called the linking loader. This approach is used widely in smaller, single-user systems.

Separating the linker from the loader offers more flexibility in that the starting address need not be known at the time of linking. Only when the module is actually to be executed are physical addresses bound by relocating the complete load module. Hence, the load module may be loaded into a different region of memory without relinking each time. However, it is still not (dynamically) relocatable.

To permit the relocation of a module at the time of linking or loading, compilers and assemblers must indicate which operands and data variables are relocatable. In general, an operand can be a register name, an immediate operand (constant), an offset to a base register, an absolute memory address, or a relative memory address. Only relative memory addresses are relocatable; all others remain unchanged. Data variables are treated
in a similar manner. If they represent relative addresses, then they must be relocated; otherwise, they remain unchanged.

Assuming that an object module is to be loaded into contiguous memory starting at location $x$ and it has been translated relative to some assumed location $y$, then all references within the program must be incremented by $k = x - y$, which is referred to as the relocation constant. In most cases, $y$ is assumed zero, and the relocation constant $k$ is simply the module’s starting address, $x$.

**EXAMPLE: Program Transformation Under Static Binding**

Figure 7-2 shows the successive transformation of an internal and an external reference as a program is prepared for execution. The source module $S$ contains an assignment statement, $i = \ldots$, and a function call, $f()$. The compiler is given zero as the starting address for the object module. Assuming that the integer variable, $i$, resides at the offset 20, the store instruction produced by the compiler as part of the assignment statement will have 20 as its operand address. The address of the function $f$ is still unknown at this time, so it remains in its symbolic form. For simplicity, the figure shows $f$ as the address of the branch instruction. In reality, the symbolic name $f$ would be kept in an **eternal symbol table**, and it would point to the corresponding branch instruction.

The external references are resolved at the time of linking, when the starting addresses of all object modules are known to the linker. Let’s assume that the function $f$ is 100 words long and that it precedes $S$ in the linking sequence. The linker, assuming zero as the starting address for the load module, adjusts the addresses as shown in the figure. That means, the operand of the store instruction is incremented by 100, which is the relocation constant for the module $S$. The external reference to $f$ is set to zero, the starting address of function $f$.

At the time of loading, the physical starting address for the load module is known. Assuming this address to be 1000, the figure shows the final modification of the code. For both the store and branch instruction, the operand is incremented by 1000—the relocation constant for the load module.

**FIGURE 7-2. Transformations of a sample program under static binding.**
It is worth noting that, in the above example, both the linker and the loader must modify operand addresses. In other words, all modules are relocated twice, once by the linker and a second time by the loader. With a linking loader, the two phases would have been combined, resulting in a single relocation.

**Dynamic Binding**

The previous section described static address binding, where all address transformations are completed before the program starts executing. More flexibility is gained when the binding occurs at runtime, i.e., immediately preceding each memory reference. This delays the binding of the program to the machine until the latest possible moment.

With such dynamic binding, the loader does not need to transform the program in any way. All memory references are still the logical addresses set up by the compiler and the linker, but they must be translated into a physical addresses before being presented to the memory. This translation is performed each time an instruction is fetched from memory and each time an operand is read from or written to memory. Since this occurs at runtime as part of each instruction execution cycle, it must be supported by hardware for reasons of efficiency.

Figure 7-3a illustrates the principles of dynamic relocation. The processor generates a logical address whenever it must fetch an instruction or access a memory operand. This address passes through a translation mechanism, called *address_map*, which performs the logical-to-physical address mapping. The resulting physical address is then presented to the memory, which either reads or writes the corresponding data item, based on the type of memory operation. Thus the *address_map* mechanism implements the following address translation function:

\[
\text{physical\_address} = \text{address\_map}(\text{logical\_address})
\]

Depending on the type of memory management supported by the system, the *address_map* can be implemented entirely in hardware, or it may require complex interaction between parts of the OS and specialized hardware support.

**FIGURE 7-3.** Dynamic address translation with (a) general address map; and (b) relocation register.
The simplest form of dynamic address binding is to use a special purpose register, called the **relocation register (RR)**, which contains the starting memory address of the currently running program. The contents of this register is then simply added to every logical address before it is presented to the memory. Formally, the **address map** then performs the following function:

\[
\text{physical\_address} = \text{logical\_address} + \text{RR}
\]

Figure 7-3b illustrates the use of a RR.

In systems using multiprogramming, the content of the RRs becomes part of each process control block. When a context switch is performed, the RRs of the currently running process are saved along with the program counter and other registers, and the register contents of the process to be restarted are restored from the values saved in its control block. Thus, the process resumes its computation at the point of its last interruption and using the same values it had at that point in time.

The main advantage of dynamic address binding is that programs may be swapped in and out of different areas of memory easily, without having to change any of their operand addresses. Only the program’s address mapping must be changed, e.g., the starting address in a relocation register.

**EXAMPLE: Program Transformation Under Dynamic Binding**

Figure 7-4 shows the transformations of a program in a system with dynamic binding. The tasks performed by the compiler and linker are the same as in Figure 7-2. The loader, however, does not change the program in any way when it loads it into memory. That means, the store instruction still contains the address zero, and the branch instruction contains the address 120. The necessary relocation constant 1000 is added to these addresses by the **address map** mechanism each time the instructions are executed.

![FIGURE 7-4. Transformations of a sample program under dynamic binding.](image)

### 7.2 MEMORY PARTITIONING SCHEMES

Given that main memory remains a critical resource in most computer systems, we must address two important problems resulting from this limitation. One is that the size of many applications simply exceeds the amount of physical memory available in a given system. To solve this problem, programs must somehow be divided into smaller blocks...
and loaded into memory only as needed. The second problem arises in multiprogramming systems, where several active processes must share main memory at the same time. In this section, we introduce simple schemes for partitioning main memory to permit such sharing and coping with its limited size.

### 7.2.1 Fixed Partitions

The simplest memory management appears in single-programmed systems, i.e., systems where only one user program resides in memory at any one time. Here, the memory only must be divided into two parts—one for the OS and the other for the user program and its data. By convention, the OS can occupy either the lower or the upper portion of the available memory. Most PCs and other smaller computers have used this form of memory management.

In multiprogrammed OSs, memory management becomes significantly more complex. The memory manager must arbitrate among the needs of many processes that are simultaneously competing for this resource. A natural extension of the above two-part scheme is to further subdivide the portion available for user programs into multiple partitions. If the sizes of these partitions are determined at the time the OS is initialized and cannot be changed at runtime, the scheme is called fixed-memory partitioning. Typically, partitions have different sizes to accommodate the needs of different programs.

Choosing appropriate partition sizes has a strong impact on the system performance. Unfortunately, it is very difficult to estimate the demands that will be placed on a system at future times. Bad choices may lead to a severe underutilization of main memory or other resources.

A problem closely related to partition sizes is process scheduling for each of the partitions. Two possible schemes, depicted in Figure 7-5a and b, can be employed. In the first, a separate queue of processes exists for each partition. Typically, a process would be scheduled for the smallest partition that satisfies its memory requirements—a scheme referred to as “best-fit.” This scheme is very simple to implement, provided the memory requirements are specified a priori by each process. Unfortunately, this is not always the case because most programs grow and shrink dynamically. There are two reasons for this. The first is the use of a function (procedure) calling stack, whose size increases with the depth of nested function calls. The second is the use of dynamically acquired memory, for example using the malloc command in UNIX, which permits a program to acquire additional pieces of memory in a structure called the heap. If a process size grows beyond the size of its current partition, it must be moved to a larger partition; to make this efficient requires dynamic relocation capabilities.

The main disadvantage of using separate queues for each partition is that some partitions may remain unused if no processes of the appropriate sizes are available. To eliminate this problem, more complex management of the queues would be necessary such that processes scheduled for a smaller partition can be moved down into a larger partition if the corresponding queue becomes empty.

The second scheme, depicted in Figure 7-5b, employs a single queue for all arriving processes. The assignment to partitions is made by the OS dynamically. Each time a process leaves a partition, the OS must search the queue for the next most suitable candidate. Although more complicated to implement, this scheme offers a greater flexibility for the system to adapt to the current workload. For example, when no process with
FIGURE 7-5. Scheduling of processes into fixed partitions: (a) using a separate queue for each partition; and (b) using a common queue.

the appropriate size is available for a given partition, the OS may decide to assign to it another process with smaller memory requirements, rather than leaving it completely idle.

The main advantage of fixed partitioning, regardless of the scheduling employed, is its simplicity. The partitions are set up at the time the system is initialized, and only a minimum amount of software is necessary to perform the necessary scheduling at runtime. But fixed partitions suffer from two serious drawbacks. First, the maximum size of a program that can run in such a system is limited by the largest partition. Overlays, as discussed in Section 7.4, can alleviate this problem, but the solution is burdensome for the programmer. The second problem is that some portion of memory is always wasted. That is because any given program will rarely fit its partition perfectly. The difference between the partition size and the program size results in memory that cannot be used for anything else. This problem is called internal fragmentation; it is “internal” because it occurs within each partition. We will revisit this problem in the context of paging in Chapter 8. Fixed partitions were used heavily in earlier mainframe computers, such as the IBM OS/MFT (Multiprogramming with Fixed Number of Tasks), but, because of the above limitations, have largely been replaced by more advanced memory management schemes.

7.2.2 Variable Partitions

Instead of deciding on the partition sizes at system initialization, the memory manager can postpone the decisions until runtime, when the actual sizes of programs that must be resident are known. Such a scheme is called variable memory partitioning. As new processes arrive, the OS assigns to each the exact amount of space requested.
However, processes do not normally terminate in the order of their arrival. Whenever a process either terminates or leaves the memory because it is waiting for the completion of an event, such as the completion of an I/O operation, it will create an area of free memory space. Thus over time, memory will consist of a sequence of variable size blocks, alternating between blocks occupied by active processes and free blocks, generally termed holes. Hence, the main task of the OS is to keep track of the holes and to assign these to newly arriving processes.

A major part of the memory management is to coalesce adjacent holes released by terminating processes into larger blocks. Without coalescing, the holes would get progressively smaller. Eventually, all free memory would consist of many adjacent holes, but each too small to accommodate any request.

Figure 7-6 illustrates the four different cases that can arise when a block $A$ is released. In the first case, the two neighboring blocks $B$ and $C$ are both occupied, so block $A$ is simply turned into a new hole $A'$; no coalescing is necessary. The cases (b) and (c) show the two symmetrical situations where one of block $A$'s neighbors is a hole. The hole created by releasing $A$ must be coalesced with the neighboring hole, resulting in a single large hole, $C'$ or $B'$. In the fourth scenario, block $A$ is surrounded by holes on both sides. Coalescing combines all three areas into a single hole $B'$.

Requests to allocate or free memory areas are a frequent operation. When presented with a request for a given number of words or bytes, the OS must be able to quickly find a suitable hole. Similarly, when a block is returned because it is no longer needed, the OS must be able to efficiently coalesce it with any surrounding holes and include it in its inventory of free spaces. There are two basic approaches to keeping track of the available spaces. One uses a linked list and the other uses a bitmap.

**Linked List Implementation**

In this approach, all holes are tied together using a linked list. However, since the linked list itself requires dynamically managed memory, the question arises: Where do we keep the linked list? The best solution is to take advantage of the unused space within the holes and to distribute the list throughout the holes themselves. That means, each hole contains the pointers to its successor hole on the list.

Unfortunately, a singly linked list does not allow efficient coalescing. When a block is freed, we must be able to find its two neighbors quickly to see if they are occupied or free. One way to achieve that is to use a doubly linked list. Figure 7-7a shows an example of such an implementation. Each block starts with a header consisting of two tags. The first indicates whether the block is free (a hole) or occupied. The second gives
the block size. Each hole also contains two pointers—one to its predecessor and one to its successor on the list of holes. Assume first that the list of holes is maintained sorted by memory addresses.

When a given block, say C in Figure 7-7a, is to be released, we need to check both its neighbors for possible coalescing. Finding the beginning of the right-hand neighbor, D, is easy since the size of the current block C is known. Since D is occupied, no coalescing is necessary. Finding the beginning of the left-hand neighbor of C is more difficult. To avoid having to scan the entire list of holes from the beginning, we can start from the first hole to the right of the current block C. In this example, this is the hole E, which we can find by skipping over any intervening occupied blocks using their size tags. Here, we skip over the block D. From the hole E, we can find the first hole to the left of C using E’s predecessor pointer. This is the hole A. Using A’s size tag, we determine that A is not C’s immediate neighbor, and no coalescing takes place on this side either.

Having to search for the left-hand neighbor makes the above scheme unnecessarily cumbersome. The situation become even more difficult when the holes are not maintained in a sorted order by memory addresses. In this case, finding the left-hand neighbor possibly requires examining the entire list of holes. A simple solution that greatly simplifies checking of both neighbors while keeping the holes unsorted, is to repeat the tag that records the block status (free or occupied) and its size at both the beginning and at the end of each block (Knuth 1968).

Figure 7-7b illustrates this solution for the same block C as before. When this is to be released, we can find its immediate right-hand neighbor, D, by using C’s size tag as before. To check C’s left-hand neighbor is equally simple. We only need to examine the bytes immediately preceding the block C itself; these contain the size and status tags of the neighboring block B. Since both B and D are occupied, no coalescing is necessary in this case. The released block C is just inserted into the list of holes as a new element. And since the list is not maintained in sorted order, C can simply be appended to the end (or head) of that list.
Bitmap Implementation

The second approach to implementing variable partitions is to use a bitmap, where each bit represents some portion of memory and indicates whether it is free or occupied. Maintaining a separate bit for each word of memory would be wasteful, but if memory is allocated in contiguous sequences of fixed-size blocks, a bitmap may be more efficient than a linked list implementation. Assume that memory is broken into 1-KB blocks for allocation purposes and there is 1 MB of main memory. Then the allocation state of memory can be represented by a string of 1024 bits (1 KB × 1024). This string is called the a bitmap, $B = b_0b_1 \ldots b_{1023}$, where $b_i = 0$ or $1$, depending on whether block $i$ is free or used, respectively.

Since few modern programming languages support bit strings explicitly, $B$ can be implemented as an array of characters, integers, or other variables. If we use characters, then an array $B[128]$ could be used, where each $B[i]$ represents 8 bits. To illustrate the use of such a bitmap, consider the blocks of Figure 7-7, and assume that the lengths of blocks $A, B, C, D, E$ are 3 KB, 2 KB, 5 KB, 1 KB, and 5 KB, respectively. Figure 7-8 shows the beginning of the corresponding bitmap. That means, the first three bits are zeros since the corresponding block $A$ is a hole, the next two bits are ones since block $B$ is occupied, and so on.

The main advantage of using a bitmap over a linked list is that all memory-management operations can be implemented efficiently using Boolean bit manipulations. Specifically, a memory-release operation can be performed by applying a bitwise AND to the appropriate portion of the bitmap $B[i]$ and a bit mask, which contains zeros in the positions corresponding to the blocks to be released and ones in the position to remain unchanged. For example, releasing block $D$ in the above example is accomplished by the operation:


Notice that, unlike the list representation, the bitmap does not require any explicit coalescing of holes. Any contiguous sequence of zeros represents a hole of the corresponding size.

The allocation of a hole can be done using a bitwise OR operation between the appropriate portion of $B$ and a bit mask containing ones in the positions to be allocated. For example, the following operation would allocate the first 2 KB of the hole $A$, leaving a remaining hole of 1 KB:

$$B[0] = B[0] \| \ '11000000';$$

A search for $k$ contiguous 1-KB blocks requires finding $k$ contiguous zeros within $B$, and can be implemented using logical shifts and bitwise AND operations. Let’s assume

<table>
<thead>
<tr>
<th>$B[0]$</th>
<th>$B[1]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00011111</td>
<td>11100000</td>
</tr>
</tbody>
</table>

FIGURE 7-8. Bitmap representation for holes.
we start the search with $B[0]$. The operation:

$$\text{TEST} = B[0] \& 10000000;$$

can be used to test the leftmost bit of $B[0]$. The variable $\text{TEST}$ contains a zero if that bit was zero, and it contains a one if the bit being tested was one. We then shift the mask by one bit to the right and repeat the AND operation. This will test the next bit of $B[0]$ in the same manner. We repeat the right-shift and AND operation until one of the following occurs:

1. $k$ consecutive zeros are found.
2. The end of $B[0]$ is reached.
3. A one is encountered.

In case 1, we found a hole of the desired size and can stop the search. In case 2, we continue the search with the next byte $B[1]$. If we reach the last byte of $B$ without finding $k$ consecutive bytes, the search has failed. Finally, in case 3, we search for the next zero in the bitmap and restart the search from there.

### 7.2.3 The Buddy System

The buddy system is a compromise between fixed and variable partitions. In its most common binary form, there is a fixed number of possible hole sizes, each of which is a power of two. This implies the basic principle of the buddy system: Any hole of size $2^i$ can be divided into two holes of size $2^{i-1}$. These two holes are called buddies and can be combined into the original hole of size $2^i$.

Assume that memory is allocated in fixed-size units. The unit could be a single word, but, more commonly, it is a fixed sequence of words (e.g., 1 KB or even larger). Assume further that the total memory comprises $2^m$ of the basic allocation units. Then the largest possible hole is the entire memory, or $2^m$ units, the next smaller hole size is $2^{m-1}$, the next smaller hole size is $2^{m-2}$, and so on. The smallest possible hole is 1 unit ($2^0$).

To keep track of the holes, the buddy system maintains an array $H$ of $m + 1$ list headers, one for each hole size. That means, $H[i]$ links together all holes of size $2^i$. A request for $n$ units of memory is then processed as follows. First, the system finds the smallest hole size such that $n \leq 2^i$. If the list of holes of this size is not empty, the system removes a hole from the list and allocates it to the requesting process. If this list is empty, the system considers increasingly larger hole sizes until a nonempty list is found. The first hole on this list, say $H[i]$, is removed and divided into two holes of half the size; let’s call these two halves the $l$ and $r$ holes. The $r$ hole is placed on the next-lower list $H[i-1]$. If the size of $l$ is the smallest possible size for the request, the hole $l$ is allocated to the process. If $l$ is still too large, the process is repeated recursively. That means, $l$ is divided into two holes of half the size, one of the holes is placed on the list of next smaller holes, and the other half is considered for allocation.

When a previously allocated block of size $2^i$ is released, the process is reversed. The system checks if the buddy of the released block is occupied. If so, the new hole is added to the list of holes of size $2^i$. If the buddy is free, it is removed from the list of
holes of size $2^i$ and the two holes are coalesced into a hole of size $2^{i+1}$. This is repeated until the largest possible hole is created.

**EXAMPLE: Memory Requests and Releases Under the Buddy System**

Figure 7-9 illustrates the above principles. We assume a memory of 16 allocation units. The list $H$ has five entries, representing the possible hole sizes of 1, 2, 4, 8, and 16. The initial configuration (Figure 7-9a) shows that the memory currently contains three occupied blocks and three holes. The holes of size 2 are linked together at $H[1]$, and the hole of size 4 is on the list at $H[2]$.

Assume that a request for a block of size 1 is now made. The list at $H[0]$ is empty and the system considers the next larger hole size at $H[1]$. The first hole on this list (at address 4) is removed and split into two buddies. One of these is allocated to the requesting process, and the other is placed on the list $H[1]$. Figure 7-9b shows the memory after the allocation.

Next assume that the block at addresses 12 and 13 is to be released, creating a hole of size 2. The buddy corresponding to this new hole is also free (addresses 14 and 15), and a hole of size 4 is created (starting at address 12). The buddy of this larger hole is free as well (addresses 8 through 11), resulting in a hole of size 8, which is entered into the list $H[3]$. Figure 7-9c shows the memory at the end of the release operation.

The buddy system as presented so far works in principle, but the release operation is quite inefficient. Finding the buddy for a given block, which is necessary for coalescing,
requires sequentially searching the list of holes at the given hole size. To speed up this operation, a set of bitmaps may be provided, one for each hole size. Then, at the time of release, the system can quickly find the bits corresponding to the released block and check if the neighboring blocks are also free.

CASE STUDY: THE BUDDY SYSTEM IN LINUX

Linux implements a paged memory, which, as will be explained in Chapter 8, does not require contiguous allocation of memory units (pages). However, some parts of the Linux OS, notably I/O operations, which move data directly between the disk and main memory, bypass the paging mechanisms. Consequently, memory must be allocated in contiguous sequences. Linux uses the buddy system for this purpose. It maintains lists of 10 different hole sizes, ranging from $2^0 = 1$ to $2^9 = 512$, which are used in the manner illustrated by the previous example (Fig. 7-9). It also maintains a set of bitmaps, one for each hole size. A bit at level $i$ represents the state of two buddies at that level (i.e., size $2^i$). The bit is 0 if both buddies are either free or both are occupied, and it is 1 if one of the buddies is free. Thus, a release operation must check the one bit representing the released block and its buddy; if this bit is 1, the buddies are coalesced and the bit at the next higher level is checked.

7.3 ALLOCATION STRATEGIES FOR VARIABLE PARTITIONS

When the memory manager is presented with a request for a block of memory of a certain size, it must find a sequence of free locations (a hole) that can accommodate that request. In general, there will be more than one hole that is large enough, so the manager must choose which hole to allocate.

Several different allocation strategies are possible, each differing in their performance characteristics. In addition, there are two related issues to consider. The first is memory utilization—the strategy should attempt to minimize the production of holes that are too small to be used. The latter is referred to as external fragmentation. It is called “external” because the wasted space occurs outside of the allocated blocks, i.e., the variable-size partitions, whereas internal fragmentation occurs within the (fixed-size) partitions. One of the main goals of the memory manager is to minimize external fragmentation, which, in turn, will maximize the number of requests that can be satisfied.

At the same time, the manager must minimize search time—the time it takes to find an appropriate hole. Regardless of whether we keep track of the holes using a linked list or bitmap, the manager must scan the holes sequentially and examine their sizes. The following are the most common allocation strategies:

1. **First-Fit.** This strategy always starts the search from the beginning of the list (or bitmap). It stops as soon as it finds the first hole large enough to accommodate the given request.

2. **Next-Fit.** Because first-fit always starts its search from the beginning of the memory, it tends to cluster the smaller holes near the top of the list. With time, it takes progressively longer on average before finding a hole of adequate size. A simple modification of the first-fit strategy is to start each search at the point where the previous search stopped, i.e., with the hole following the previous allocation. This
scheme, called rotating-first-fit or next-fit, keeps the hole sizes more uniformly distributed over the list and decreases the average search time. Unfortunately, simulations show that it results in a slightly worse memory utilization than first-fit.

3. **Best-Fit.** The intuitive idea behind this strategy is to improve memory utilization by always selecting the hole with the closest fit, i.e., the smallest hole that is still large enough to satisfy a given request. Unfortunately, the intuition fails us. Simulations have shown that the best-fit strategy results in a worse memory utilization than first-fit or next-fit. The reason is excessive memory fragmentation. Since the hole with the closest fit is selected, the remaining fragment is usually too small to be of any use. Consequently, best-fit tends to generate large numbers of very small holes, which increases memory fragmentation. Unlike first-fit or next-fit, best-fit also has a long search time because it must examine all holes to find the best one. This search is further exacerbated by the presence of many small holes. Thus best-fit, despite its name, is the worst choice among the three strategies.

4. **Worst-Fit.** Given that best-fit leads to such an excessive memory fragmentation, another idea is to do just the opposite. That means, use the largest currently available hole for any given request. The rationale behind this strategy, called worst-fit, is that the remaining hole will always be large enough to accommodate other requests, rather than just wasting space. Unfortunately, simulations have shown that the performance of worst-fit is not better than that of first-fit or next-fit, which are the easiest to implement.

The general conclusion that first-fit is the best choice among the four strategies of an allocation algorithm is based on simulations performed at different times and assuming different technologies (Knuth 1968, Shore 1975, Bays 1977). Other, more complex strategies, known as optimal-fit algorithms, have been developed to combine the benefits of the basic strategies. The choice for a particular system should be made based on its intended application domain, which determines a number of parameters crucial to performance, such as distributions of request size, request arrival times, and the residency time of blocks in memory.

### 7.3.1 Measures of Memory Utilization

It is possible to obtain general measures of the external fragmentation and use of main memory. To perform this analysis, we assume that the system has reached an equilibrium state where, on average, the number of requests is the same as the number of releases per unit of time. Thus, the average number of holes is constant.

Let \( n \) and \( m \) be the average number of holes and occupied blocks, respectively, at equilibrium. Consider Figure 7-6, which shows the effect of releasing a block \( A \). There are four possible cases, depending on whether \( A \) is surrounded by free or occupied blocks. Let \( a, b, c, \) and \( d \) be the average number of occupied blocks of each type in memory during equilibrium. That means, a block of type \( a \) is surrounded by occupied blocks on both sides. A block of type \( b \) is surrounded by an occupied block on its left side and a hole on its right, and so on. The total number \( m \) of occupied blocks is then the sum of all four types:

\[
m = a + b + c + d
\]  
(7.1)
The number of holes is derived as follows: For a block of type $a$, there are no neighboring holes; for each block of type $b$ or $c$, there is one neighboring hole; and for each block of type $d$, there are two neighboring holes. Since a hole is always between two blocks, it must not be counted twice; thus, the total number is divided by 2:

$$n = \frac{2d + b + c}{2} \quad (7.2)$$

We also know that every consecutive sequence of occupied blocks consists of zero or more blocks of type $a$, surrounded by one block of type $b$ and one block of type $c$ on each side. Thus, $b = c$. Substituting this into Equation 7.2, we obtain:

$$n = d + b \quad (7.3)$$

As illustrated in Figure 7-6, the number of holes increases by 1 in (a), decreases by 1 in (d), and remains constant in (b) and (c) as a result of the release. Thus, the probability that $n$ increases by 1 at any change in memory allocation is:

$$\text{Prob}(\text{release}) \times \frac{a}{m} \quad (7.4)$$

where $\text{Prob}(\text{release})$ is the probability that a memory operation is a release.

The probability that $n$ decreases by 1 consists of two components. When the operation is a release, $n$ decreases if a block of type $d$ is being released. When the operation is a request, $n$ decreases by one but only when the request is an exact match (otherwise the hole is only reduced in size). Let $q$ be the probability of finding an exact hole size for a given request and $p = 1 - q$. Then the probability that $n$ decreases by 1 at any change in memory allocation is:

$$\text{Prob}(\text{release}) \times \frac{d}{m} + \text{Prob}(\text{request}) \times (1 - p) \quad (7.5)$$

where $\text{Prob}(\text{release})$ and $\text{Prob}(\text{request})$ are the relative probabilities of memory requests and releases.

The equilibrium condition implies that $(7.4) = (7.5)$. It also implies that $\text{Prob}(\text{request}) = \text{Prob}(\text{release})$. This yields:

$$\frac{a}{m} = \frac{d}{m} + (1 - p)$$

$$a = d + (1 - p)m \quad (7.6)$$

Substituting (7.3) and (7.6) in (7.1), we have:

$$m = d + (1 - p)m + b + c + d = (1 - p)m + 2b + 2d = (1 - p)m + 2n$$

which gives the final result:

$$n = 0.5 \, pm$$
Section 7.3 Allocation Strategies for Variable Partitions

This measure of memory fragmentation was originally derived by Knuth (1968), who called it the 50% rule, since \( n \) tends to 50% of \( m \) as \( p \) approaches 1. In other words, when the probability \( p \) of not finding an exact match approaches 1, one-third of all the memory partitions are holes and two-thirds are occupied blocks.

The 50% rule only makes a statement about the relative numbers of holes and occupied blocks, but not about their relative sizes. Knowing these would permit us to make a statement about memory utilization, i.e., what fraction of memory is used by blocks and how much is wasted in holes.

Denning (1970) obtained the following result on memory utilization: Let the average hole size be \( h = kb \), where \( b \) is the average occupied block size and \( k > 0 \) is a constant. \( k \) is the ratio between the average hole size and the average occupied block size. If we let \( p \), the probability used in the 50% rule, be 1, then the fraction \( f \) of memory occupied by holes in an equilibrium state is given by the following equation:

\[
f = \frac{k}{k+2}
\]

This can be derived by using the 50% rule with \( p = 1 \) as follows. Let \( M \) be the memory size. Since \( M = nh + mb \), the hole size \( h \) can be expressed as:

\[
h = \frac{M - mb}{n} = \frac{M - mb}{0.5 m}
\]

Since \( h = kb \), we obtain:

\[
k = \frac{M - mb}{0.5 m}
\]

\[
M = 0.5 kmb + mb = mb(0.5k + 1)
\]

The fraction \( f \) of memory \( M \) occupied by holes is then:

\[
f = \frac{nh}{M} = \frac{0.5 mh}{M} = \frac{0.5 mkb}{mb(0.5k + 1)} = \frac{k}{k + 2}
\]

The intuition behind this result is that \( k \), the ratio between the average hole size \( h \) and the average occupied block size \( b \), must be small; otherwise, much of memory will be unused. For example, with \( k = 1 \), i.e., when holes and occupied blocks are of the same size, \( f = 1/3 \). This is consistent with the 50% rule, which states that one-third of memory are holes. To reduce \( f \), \( k \) must be made smaller. But how do we know what \( k \) is in a given situation, and how can we influence it?

Simulations by Knuth (1968) provide an answer. The ratio \( k \) depends on the average block size \( b \) relative to the total memory size \( M \). For example, when \( b \) is below approximately one-tenth of \( M \), i.e., \( b \leq M/10 \), then \( k = 0.22 \). This then results in \( f \approx 0.1 \), i.e., only approximately 10% of total memory remains unoccupied.

In contrast, when \( b \) is large relative to \( M \), say about \( M/3 \), \( k \) becomes 2, and \( f \) rises to 0.5. That means, when requests are large on average, one-half of the total memory space is taken up by holes, where the average hole size is twice that of an occupied
block. The obvious conclusion is that $M$ must be large relative to $b$, otherwise much of main memory will remain unused.

### 7.4 MANAGING INSUFFICIENT MEMORY

The amount of main memory available in a given computer system has been increasing steadily over past decades. Yet, regardless of how much memory a system has, there are always situations when it is not sufficient. In time-sharing systems, a part of the programs of all active users must be resident in memory. When a new process arrives while memory is full, space must somehow be created; otherwise, the process would have to wait. In batch systems, jobs can be delayed, but this may waste CPU time unnecessarily. Since I/O devices are much slower than the CPU, it is frequently the case that all currently resident processes are blocked, waiting for I/O completion or another event. During this time, the CPU would be idle, unless additional ready processes can be loaded into memory. Finally, the size of even a single program can exceed the size of the largest partition, which in the case of variable-size partitions is the entire memory (minus the portion occupied by the OS). There are several different ways to address these problems.

#### 7.4.1 Memory Compaction

When a request cannot be satisfied because there is no block of free space large enough to accommodate the request, memory compaction may be considered. This involves consolidating smaller holes dispersed throughout the memory into a single larger hole.

There are several approaches to the problem. To illustrate the differences, assume that memory is occupied as shown in Figure 7-10a and that a request for a block of size 10 is to be satisfied. The most straightforward way to accomplish this is to move all currently occupied blocks to one end of the memory, thus creating a single free space of size 20 (Fig. 7-10b).

Note, however, that there is no need to reorganize the entire memory to create a free space of size 10. Instead of moving all the blocks at once, the compaction procedure could start moving occupied blocks to one end of the memory as in the previous case, but stop when a free space of sufficient size is created. Figure 7-10c shows the memory at that point. This approach saves time since, in general, only a small fraction of the blocks may be moved.

Finally, more sophisticated versions of the above strategies have been devised in an attempt to minimize the total amount of work to be performed. These include various heuristics to move only those blocks that are likely to generate the most space. For example, in Figure 7-10d, only the block $p2$ was relocated to free the necessary space. Such algorithms, however, are quite complex and not widely used.

The main limitation of all memory compaction schemes is that they are very costly. Even with dynamic relocation capabilities, which make it possible to move programs into different locations without any modifications to code or data, compaction requires each word of memory to be read from and written into memory. Depending on the size and type of memory, this may take as much as several seconds to accomplish. Due to this high overhead, few contemporary systems implement memory compaction.
Swapping

When a new program must be loaded into memory—either because a new user is joining a time-sharing system or because all currently resident processes are blocked—we can create new space by selecting one of the resident processes and temporarily evicting it to secondary storage. This exchange is generally called swapping.

To make swapping efficient, it is important to decide where to keep the swapped-out process on disk. First, some portions of the process address space are already on the disk and there is no need to create new copies of these areas. This is true of all code areas, which have been loaded from executable files when the process was first started. As long as the process cannot modify its code, these areas do not need to be saved to disk—they are simply reloaded from the original code files when the process is swapped back in.

Data areas (both stack and heap) are created and modified at runtime. To save these on disk, the OS can use the file system and save the areas as special files. To avoid the overhead associated with file management, some systems designate a special area (partition) of the disk as the swap space. This consists of consecutive cylinders (tracks), which the OS can access efficiently by using low-level disk read/write operations.

In a batch system, swapping is typically triggered at the time when all current processes are blocked. In a time-sharing system, swapping can be conveniently integrated with scheduling, and performed at the end of each time quantum. Before restarting the next process, the scheduler can initiate the loading of one of the swapped-out processes so that it is ready to run when it becomes its turn. This process would replace one of the currently resident processes, possibly the one that just exhausted its time quantum, or
one with a low priority. The swapping, an I/O-bound sequence of operations, proceeds concurrently with the execution of other processes and does not cause any significant disruption in the service.

Swapping is more general and more efficient than memory compaction. First, memory compaction cannot guarantee to create a hole large enough to accommodate a request. With swapping, we can always evict as many resident processes as is necessary. Swapping also affects only one or a small number of processes each time, thus requiring fewer memory accesses. Unlike compaction, swapping requires accesses to secondary memory, but these can be overlapped with the executing of other processes. Finally, swapping can be used with both fixed and variable size partitions, whereas memory compaction is applicable to only variable partitions.

### CASE STUDIES: SWAPPING

1. **UNIX.** Most versions of UNIX prior to the Berkeley 3BSD used swapping to manage an appropriate load of processes in main memory. Swapping is implemented by a special process called the **swapper**. This is invoked in the following situations:
   - when a new process is created (using `fork`);
   - when the memory demands of an existing process increase; this is the case when the process explicitly increases its data area (using `brk` or `sbrk`) or when the process stack grows beyond its originally assigned stack area;
   - periodically, every 4 seconds, to guarantee that no process remains swapped out for too long.

   In all cases, it may be necessary to swap out one or more of the currently resident processes. If that is the case, the swapper first considers blocked processes. If no such processes exist, ready processes are considered. Within each type, the swapper considers various attributes of the processes, such as the amount of CPU time used and the time a process has been resident in memory, when making the selection of the victim. In addition, every process is guaranteed a minimum residency time (e.g., 2 seconds) to prevent a situation called **thrashing**, where much of the time is spent on moving processes repeatedly between the disk and main memory, and no real work gets accomplished by any process.

   If no process must be removed, the swapper attempts to swap additional processes back into memory. Again, it considers various characteristics of the processes, notably the time they have been swapped out, when making the selection.

2. **Windows 2000.** Windows 2000 uses swapping to remove idle threads and processes from main memory until they must run again. A system thread called the **swapper** is woken up every 4 seconds. It looks for threads that have been idle for more than a certain length of time (3 seconds on small memory systems and 7 seconds on large memory systems). All such threads, i.e., their kernel stacks, are swapped out. When all threads of a given process have been swapped out, the remainder of the process, which includes code and data shared by the threads, is also removed from memory. The entire process is then considered as swapped out.
Section 7.4    Managing Insufficient Memory    227

Overlays

Memory compaction and swapping address the problem of insufficient memory caused by the presence of other programs. This shortage of memory is only temporary. But neither technique can do anything about the inability to run programs that exceed the total size of physical memory or, in the case of fixed partitions, the size of the largest partition. This shortage of memory is permanent and can only be addressed by dividing the program into smaller parts that can be loaded into memory and executed separately.

The simplest solution is overlaying, where different portions of the program replace (overlay) each other in memory as execution proceeds. The programmer is required to specify those parts of the program that must reside in memory simultaneously. Such a specification may be obtained from the calling structure (tree) of the program, which describes the dependencies among functions and the data they access.

**EXAMPLE: Overlays**

Consider a program consisting of five functions A, B, C, D, and E. The calling structure depicted in Figure 7-11a shows the function A as the parent (caller) of the two functions B and C, and C as the caller of D and E. Since B and C do not call each other, they need not reside in memory at the same time. The same is true of the functions D and E; furthermore, their presence is needed only when C is resident. Based on this information, the OS can allocate memory as shown graphically in Figure 7-11b. The two functions B and C have the same starting address k1, whereas k2 is the common starting address of D and E. The compiler inserts a special piece of code at the common addresses k1 and k2, which causes the overlay manager to be invoked. Thus, each time A calls either B or C, the manager is invoked, it determines which of the two functions is to be executed, and, if the needed function is not present, it loads and transfers control to it. The same principle applies when C calls D or E.

![Diagram](image)

**FIGURE 7-11.** Program overlays: (a) function call hierarchy; and (b) address assignments.

The main drawback of overlays is that it places the burden of memory management on the programmer, who must plan and design all programs according to the available physical memory space. As a result, manual overlays are rarely used in modern systems. The main reason for discussing them in this chapter is that they represent a natural precursor for automated techniques known as virtual memory. In the next chapter, we
Chapter 7 Physical Memory

present the concept of virtual memory, which liberates the programmer from the limits of physical memory by shifting the burden of memory management entirely to the OS.

CONCEPTS, TERMS, AND ABBREVIATIONS

The following concepts have been introduced in this chapter. Test yourself by defining and discussing each keyword or phrase.

<table>
<thead>
<tr>
<th>Address map</th>
<th>Linking loader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Load module</td>
</tr>
<tr>
<td>Best-fit memory allocation</td>
<td>Next-fit memory allocation</td>
</tr>
<tr>
<td>Binding of addresses</td>
<td>Object module</td>
</tr>
<tr>
<td>Bit map</td>
<td>Overlaying</td>
</tr>
<tr>
<td>Buddy system</td>
<td>Partitioning memory</td>
</tr>
<tr>
<td>Coalescing holes</td>
<td>Relocatable program</td>
</tr>
<tr>
<td>Compacting memory</td>
<td>Relocation constant</td>
</tr>
<tr>
<td>Fifty-percent rule</td>
<td>Relocation register (RR)</td>
</tr>
<tr>
<td>First-fit memory allocation</td>
<td>Source module</td>
</tr>
<tr>
<td>Fragmentation</td>
<td>Swapping</td>
</tr>
<tr>
<td>Holes in memory</td>
<td>Variable partitions</td>
</tr>
<tr>
<td>Linking</td>
<td>Worst-fit memory allocation</td>
</tr>
</tbody>
</table>

EXERCISES

1. Consider the examples in Figure 7-2 and Figure 7-4. Assume that another function, occupying 50 memory words, is linked in front of function \( f \). Assume further that the final load module will be loaded into memory starting at location 1500. Show how the various addresses will be changed in the two cases.

2. Consider a request for \( n \) bytes of memory using variable partitions. A hole of size \( m \) (where \( m > n \)) is chosen to satisfy the request. Is it better to allocate the first \( n \) bytes or the last \( n \) bytes of the hole? Justify your choice.

3. Consider a variable partition scheme where tags are replicated at the beginning and at the each of each block and each hole (Fig. 7-7b). Draw a diagram showing the layout of main memory after each of the following operations:
   (a) memory is initialized to empty; the total memory size is 64 MB
   (b) three blocks (A, B, C), each of size 1 MB, are requested and allocated in the given sequence
   (c) block B is released
   (d) block A is released
   (e) block C is released

4. An implementation of a variable partitions scheme using linked lists can make the following choices:
   • the list of holes is singly linked or doubly linked
   • the list of holes is kept sorted by memory addresses or unsorted
   • the tags (size and type of each block) is kept only at the beginning of each block or it is replicated at both ends.

   For each of the eight possible combinations of design choices, describe advantages and disadvantages with respect to:
   (a) searching for a hole of a given size
Section 7.4 Managing Insufficient Memory

(b) allocating a hole (or a portion of a hole) to satisfy a request
(c) releasing a block

5. Consider a buddy system with 5 different hole sizes ($2^0$ through $2^4$).
   (a) Assume a sequence of requests of the following sizes is made: 1, 2, 4, 2. Show
       the memory layout, including the header array $H$, after all requests have been
       accommodated.
   (b) Assume that the four blocks (each of size 1) at addresses 4, 5, 6, and 7 are
       released one at a time. Show the memory layout, including the header array $H$, 
       after each release.

6. Assume a request for 1 byte of memory. How many holes will be visited by each of
   these schemes:
   (a) first fit
   (b) next fit
   (c) best fit
   (d) worst fit

7. Assume that memory contains three holes of 10 MB each. A sequence of 14 requests
   for 1 MB each is to be processed. For each of the memory allocation methods listed
   below, determine the sizes of the remaining holes after all 14 requests have been
   satisfied:
   (a) first fit
   (b) next fit
   (c) best fit
   (d) worst fit

8. Assume that the list of holes in a variable partitions memory system contains the
   following entries (in the given order): 190 KB, 550 KB, 220 KB, 420 KB, 650 KB,
   110 KB. Consider the following sequence of requests (in the given order): A =
   210 KB, B = 430 KB, C = 100 KB, D = 420 KB. Determine which holes would be
   allocated to which request by each of the following schemes:
   (a) first fit
   (b) next fit
   (c) best fit
   (d) worst fit
   What is the average hole size for each scheme at the end of the allocation sequence?

9. Consider a system with 640 KB of main memory. The following blocks are to be
   requested and released:

<table>
<thead>
<tr>
<th>block</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>size (KB)</td>
<td>200</td>
<td>200</td>
<td>150</td>
<td>80</td>
<td>80</td>
<td>240</td>
</tr>
</tbody>
</table>

   Obviously, not all blocks may be accommodated at the same time. We say an allocation
   failure occurs when there is no hole large enough to satisfy a given request. Devise a
   sequence of requests and releases that results in an allocation failure under:
   (a) the first fit policy, but not the best fit policy
   (b) the best fit policy, but not the first fit policy
   For each of the two situations, draw a diagram that shows what parts of memory are
   allocated at the time of failure. (Hint: There is no systematic approach to this problem;
   you simply must experiment with different sequences.)

10. Consider a 16-MB main memory using variable partitions. Assume that (1) $hole_{\text{size}}$
    = $block_{\text{size}}$ = 1 KB, and (2) the 50% rule holds. Determine the following:
    (a) Total number of holes
11. Consider a memory with variable partition memory. Assume that 1) hole_size = block_size; and 2) the 50% rule holds. If one-half of all occupied blocks are released in a single operation, how will the total number of holes change? How will the average hole size change?

12. Compare the overhead of using a bitmap versus a linked list to keep track of memory allocation. The bitmap approach allocates memory in chunks of $c$ bytes. For the linked list implementation, assume that the average size of both occupied blocks and holes is 16 KB. The size of each of the tags is 2 bytes, and they are replicated at both ends of each block/hole. Determine the value of $c$ for which the two approaches have the same overhead.

13. Consider a memory organized into variable partitions and make the following assumptions. If holes are organized using a linked list, the overhead is 8 bytes per block to store the type and size tags. If a bit map is used, one bit is needed for every 128-byte block. Determine the average size of an occupied block such that the overhead of both methods is the same for the following two cases:

(a) $k = \frac{\text{hole size}}{\text{block size}} = 1$

(b) $k = \frac{\text{hole size}}{\text{block size}} = .5$

(Hint: Use the 50% rule with $p = 1$.)

14. Consider a 256-MB memory organized into variable partitions. Assume that 1) hole_size = block_size = 1 KB; and 2) the 50% rule holds. How long will it take to compact the entire memory if reading or writing of one 32-bit word takes 10 ns?

15. Consider a system with 4.2 MB of main memory using variable partitions. At some point in time, the memory will be occupied by three blocks of code/data as follows:

<table>
<thead>
<tr>
<th>starting address</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000</td>
<td>1 MB</td>
</tr>
<tr>
<td>2,900,000</td>
<td>.5 MB</td>
</tr>
<tr>
<td>3,400,000</td>
<td>.8 MB</td>
</tr>
</tbody>
</table>

The system uses the best-fit algorithm. Whenever a request fails, memory is compacted using one of the following schemes:

- All blocks are shifted toward address zero (see Fig. 7.10b)
- Starting with the lowest-address block, blocks are shifted toward address zero until a hole big enough for the current request is created (see Fig. 7.10c)

Assume that three new blocks with the respective sizes 0.5 MB, 1.2 MB, and 0.2 MB are to be loaded (in the given sequence).

(a) Show the memory contents after all three requests have been satisfied under the two different memory compaction schemes.

(b) How many bytes of memory had to be copied in each case?

16. When a memory request fails, a larger hole may be created by:

- moving one or more blocks within main memory (compaction);
- moving one or more blocks to the disk (swapping).

What are the main advantages and drawbacks of each scheme?