9.5 Memory Allocation Techniques

The primary role of the memory management system is to satisfy requests for memory allocation. Sometimes this is implicit, as when a new process is created. At other times, processes explicitly request memory. Either way, the system must locate enough unallocated memory and assign it to the process.

9.5.1 Free Space Management

Before we can allocate memory, we must locate the free memory. Naturally, we want to represent the free memory blocks in a way that makes the search efficient.

Before getting into the details, however, we should ask whether we are talking about locating free memory in the physical memory space or the virtual memory space. Throughout this chapter, we look at memory management techniques primarily from the perspective of the operating system managing the physical memory resource. Consequently, these techniques can all be viewed as operating in the physical memory space. However, many of these techniques can also be used to manage virtual memory space. Application-level dynamic memory allocation, using familiar operations such as the *malloc*( ) call in C or the **new** operator in C++, often allocate large blocks from the OS and then subdivide them into smaller allocations. They may well use some of these same techniques to manage their own usage of memory. The operating system, however, does not concern itself with that use of these techniques.

9.5.1.1 Free Bitmaps

If we are operating in an environment with fixed-sized pages, then the search becomes easy. We don't care which page, because they're all the same size. It's quite common in this case to simply store one bit per page frame, which is set to one if the page frame is free, and zero if it is allocated. With this representation, we can mark a page as either free or allocated in constant time by just indexing into this **free bitmap**. Finding a free page is simply a matter of locating the first nonzero bit in the map. To make this search easier, we often keep track of the first available page. When we allocate it, we search from that point on to find the next available one.

The memory overhead for a free bitmap representation is quite small. For example, if we have pages that are 4096 bytes each, the bitmap uses 1 bit for each 32,768 bits of memory, a 0.003% overhead.

Generally, when we allocate in an environment that uses paging address translation, we don't care which page frame we give a process, and the process never needs to have control over the physical relationship among pages. However, there are exceptions. One exception is a case where we do allocate memory in fixed sized units, but where there is no address translation. Another is where not all page frames are created equally. In both cases, we might need to request a number of physically contiguous page frames. When we allocate multiple contiguous page frames, we look not for the first available page, but for a run of available pages at least as large as the allocation request.

9.5.1.2 Free Lists

We can also represent the set of free memory blocks by keeping them in a linked list. When dealing with fixed-sized pages, allocation is again quite easy. We just grab the first page off the list. When pages are returned to the free set, we simply add them to the list. Both of these are constant time operations.

If we are allocating memory in variable-sized units, then we need to search the list to find a suitable block. In general, this process can take an amount of time proportional to the number of free memory blocks. Depending on whether we choose to keep the list sorted, adding a new memory block to the free list can also take $O(n)$ time (proportional to the number of free blocks). To speed the search for particular sized blocks, we often use more complex data structures. Standard data structures such as binary search trees and hash tables are among the more commonly used ones.

Using the usual linked list representation, we have a structure that contains the starting address, the size, and a pointer to the next element in the list. In a typical 32-bit system, this structure takes 12 bytes. So if the average size of a block is 4096 bytes, the free list would take about 0.3% of the available free space. However, there's a classic trick we can play to reduce this overhead to nothing except for a pointer to the head of the list. This trick is based on finding some other way to keep track of the starting address, the size, and the pointers that define the list structure. Because each element of the list represents free space, we can store the size and pointer to the next one in the free block itself. (The starting address is implicit.) This technique is illustrated in Example 9.1.

Example 9.1: Free List Structure

Consider a free list with three free blocks, of sizes 3, 8, and 16. If the block of size 16 is first in memory, followed by the blocks of sizes 3 and 8, then the list structure shown in Figure 9-8 stores the list in ascending order of size.

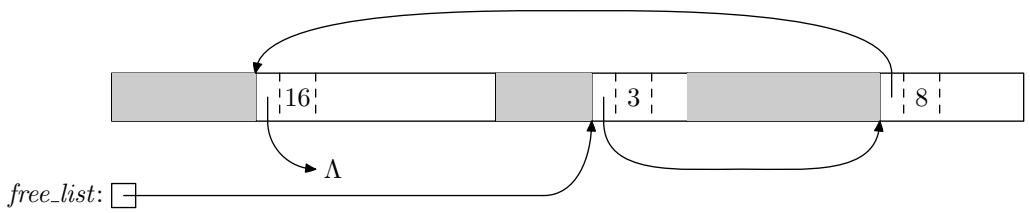

Figure 9-8: Free List Example

There are several things in this example we should note. First, except for the global pointer *free_list*, we store everything in the free blocks themselves. The only overhead is this pointer. Second, if a block is of size 16 KB and the pointers and sizes are stored in 4-byte integers, then the unused space while the block is in the free list is $16384 - 8 = 16376$. However, the full 16,384 bytes are available to be allocated to a requesting process. Third,

we do not store the starting address of a block in its descriptive information. When we follow a pointer to a block, that pointer gives us the starting address.

9.5.2 Fragmentation

When allocating memory, we can end up with some wasted space. This happens in two ways. First, if we allocate memory in such a way that we actually allocate more than is requested, some of the allocated block will go unused. This type of waste is called **internal fragmentation**. The other type of waste is unused memory outside of any allocated unit. This can happen if there are available free blocks that are too small to satisfy any request. Wasted memory that lies outside allocation units is called **external fragmentation**.

9.5.3 Partitioning

The simplest methods of allocating memory are based on dividing memory into areas with fixed **partitions**. Typically, we administratively define fixed partitions between blocks of varying size. These partitions are in effect from the time the system starts to the time it is shut down. Memory requests are all satisfied from the fixed set of defined partitions. This approach is illustrated in Example 9.2.

Example 9.2: Fixed Partitioning

Although fixed partitioning is not commonly found in modern, general-purpose systems, it is seeing a sort of revival. Some of the virtualization systems use simple, fixed partitions between the various virtual systems. One good example of this is Xen. In Xen, the memory used by the OS identified as the Domain 0 OS is specified with the option `dom0_mem` in whatever boot loader is used to load the Xen hypervisor into memory. For example, when using `grub`, the line

```
kernel=/xen.gz dom0_mem=262144 console=vga
```

declares that the Domain 0 OS has 256 MB reserved for it. For OSs run in other domains, the line

```
memory = 128
```

in a configuration file reserves 128 MB for the corresponding domain.

More commonly, however, we want the flexibility of allocating memory in either large or small units as needed. When we allocate memory in these systems, we pick a free block and split it into two parts. The first part is the memory we allocate to the requesting process, and the second part is returned to the set of free memory blocks. When allocating in such a variable partition scheme, we allocate in multiples of some minimum allocation unit. This unit is a design parameter of the OS and is not a function of the hardware MMU design. This helps to reduce external fragmentation. In most cases, these allocation units are relatively small—usually smaller than the page frame sizes we typically see in systems that use paging. Furthermore, if we are using a free list data structure stored in the free blocks, then we must ensure that each free block is large enough to hold the structure.

9.5.4 Selection Policies

If more than one free block can satisfy a request, then which one should we pick? There are several schemes that are frequently studied and are commonly used.

9.5.4.1 First Fit

The first of these is called **first fit**. The basic idea with first fit allocation is that we begin searching the list and take the first block whose size is greater than or equal to the request size, as illustrated in Example 9.3. If we reach the end of the list without finding a suitable block, then the request fails. Because the list is often kept sorted in order of address, a first fit policy tends to cause allocations to be clustered toward the low memory addresses. The net effect is that the low memory area tends to get fragmented, while the upper memory area tends to have larger free blocks.

Example 9.3: First Fit Allocation

To illustrate the behavior of first fit allocation, as well as the other allocation policies later, we trace their behavior on a set of allocation and deallocation requests. We denote this sequence as A20, A15, A10, A25, D20, D10, A8, A30, D15, A15, where An denotes an allocation request for n KB and Dn denotes a deallocation request for the allocated block of size n KB. (For simplicity of notation, we have only one block of a given size allocated at a time. None of the policies depend on this property; it is used here merely for clarity.) In these examples, the memory space from which we serve requests is 128 KB. Each row of Figure 9-9 shows the state of memory after the operation labeling it on the left. Shaded blocks are allocated and unshaded blocks are free. The size of each block is shown in the corresponding box in the figure. In this, and other allocation figures in this chapter, time moves downward in the figure. In other words, each operation happens prior to the one below it.

A20	20	108					
A15	20	15	93				
A10	20	15	10	83			
A25	20	15	10	25	58		
D20	20	15	10	25	58		
D10	20	15	10	25	58		
A8	8	12	15	10	25	58	
A30	8	12	15	10	25	30	28
D15	8	37			25	30	28
A15	8	15	22		25	30	28

Figure 9-9: First Fit Allocation

9.5.4.2 Next Fit

If we want to spread the allocations out more evenly across the memory space, we often use a policy called **next fit**. This scheme is very similar to the first fit approach, except for the place where the search starts. In next fit, we begin the search with the free block that was next on the list after the last allocation. During the search, we treat the list as a circular one. If we come back to the place where we started without finding a suitable block, then the search fails. Example 9.4 illustrates this technique.

Example 9.4: Next Fit Allocation

For the next three allocation policies in this section, the results after the first six requests (up through the D10 request) are the same. In Figure 9-10, we show the the results after each of the other requests when following the next fit policy.

A8	20	15	10	25	8	50		
A30	20	15	10	25	8	30	20	
D15	45			25	8	30	20	
A15	45			25	8	30	15	5

Figure 9-10: Next Fit Allocation

9.5.4.3 Best Fit

In many ways, the most natural approach is to allocate the free block that is closest in size to the request. This technique is called **best fit**. In best fit, we search the list for the block that is smallest but greater than or equal to the request size. This is illustrated in Example 9.5. Like first fit, best fit tends to create significant external fragmentation, but keeps large blocks available for potential large allocation requests.

Example 9.5: Best Fit Allocation

As with the next fit example, we show only the final four steps of best fit allocation for our example. The memory layouts for these requests is shown in Figure 9-11.

A8	20	15	8	2	25	58		
A30	20	15	8	2	25	30	28	
D15	35		8	2	25	30	28	
A15	35		8	2	25	30	15	13

Figure 9-11: Best Fit Allocation

9.5.4.4 Worst Fit

If best fit allocates the smallest block that satisfies the request, then **worst fit** allocates the largest block for every request. Although the name would suggest that we would never use the worst fit policy, it does have one advantage: If most of the requests are of similar size, a worst fit policy tends to minimize external fragmentation. We illustrate this technique in Example 9.6.

Example 9.6: Worst Fit Allocation

Finally, Figure 9-12 shows the memory layout after each of the last four requests in our example for worst fit allocation.

<table>
<tr><td>A8</td><td>20</td><td>15</td><td>10</td><td>25</td><td>8</td><td colspan="2">50</td></tr>
<tr><td>A30</td><td>20</td><td>15</td><td>10</td><td>25</td><td>8</td><td>30</td><td>20</td></tr>
<tr><td>D15</td><td colspan="2">45</td><td></td><td>25</td><td>8</td><td>30</td><td>20</td></tr>
<tr><td>A15</td><td>15</td><td colspan="2">30</td><td>25</td><td>8</td><td>30</td><td>20</td></tr>
</table>

Figure 9-12: Worst Fit Allocation

9.5.5 Buddy System Management

There is another memory allocation system, which is very elegant and which tends to have very little external fragmentation. This approach is called the **buddy system** and is based on the idea that all allocated blocks are a power of 2 in size. The buddy system allocation algorithm can be described as follows:

Buddy System Allocation: Let n be the size of the request. Locate a block of at least n bytes and return it to the requesting process.

1. If n is less than the smallest allocation unit, set n to be that smallest size.
2. Round n up to the nearest power of 2. In particular, select the smallest k such that $2^k \geq n$.
3. If there is no free block of size 2^k, then recursively allocate a block of size 2^{k+1} and split it into two free blocks of size 2^k.
4. Return the first free block of size 2^k in response to the request.

We call this technique the buddy system because each time we split a block, we create a pair of buddies that will always either be split or paired together. Neither will ever be paired with another block. From the offset of a block, we know whether it is the left buddy or the right buddy in a pair. In particular, if we have a block of size 2^k, then bit k of the offset (numbered from the least significant bit starting at 0) is 0 if the block is a left buddy and 1 if it is a right buddy. For instance, a block of 32 bytes will start at an offset whose last six bits are either 000000 or 100000. In the first case, it is the left buddy and

in the second, it is the right. To find a block's buddy, we only need to complement bit k of the offset. So when freeing a block, we can easily find which block can be combined to make a larger block. If we do find that the block's buddy is also free and combine the two, then we will recursively attempt to combine the new block with its buddy. This method is illustrated in Example 9.7.

Buddy system allocation is very straightforward and tends to have very low external fragmentation. However, the price we pay for that is increased internal fragmentation. In the worst case, each allocation request is 1 byte greater than a power of 2. In this case, every allocation is nearly twice as large as the size requested. Of course, in practice, the actual internal fragmentation is substantially smaller, but still tends to be larger than what we find with the other variable-sized block techniques.

Example 9.7: Buddy System Allocation

We return again to the allocation requests in the previous examples to see how the buddy system would handle them. Figure 9-13 shows the memory allocations under the buddy system. Within each allocated block, both the size of the allocated block and the size of the request are given to illustrate the internal fragmentation.

	128					
A20	32:20	32			64	
A15	32:20	16:15	16		64	
A10	32:20	16:15	16:10		64	
A25	32:20	16:15	16:10		32:25	32
D20	32	16:15	16:10		32:25	32
D10	32	16:15	16		32:25	32
A8	32	16:15	8:8	8	32:25	32
A30	32:30	16:15	8:8	8	32:25	32
D15	32:30	16	8:8	8	32:25	32
A15	32:30	16:15	8:8	8	32:25	32
D8	32:30	16:15	16		32:25	32
D30	32	16:15	16		32:25	32
D15	64				32:25	32

Figure 9-13: Buddy System Allocation

The set of allocations and deallocations in this example illustrate a number of aspects of this technique. In the first request, we want to allocate a block of size 20, that we round up to the nearest power of 2, which is 32. There are no free blocks of that size, so we take the only free block there is (size 128) and split it into two free blocks of size 64. We still don't have a block of size 32, so we take the recursion one more step and split one of the blocks of size 64 into two of size 32. Now we have a block we can use to satisfy the request. The other allocation requests are comparatively straightforward in that they can all be satisfied either with an available free block or with one split. The last three deallocations

(beyond those in the earlier examples) also deserve note. In each of these three cases, the block that is deallocated is a buddy to free block. In these cases, we combine them into a bigger free block. The last deallocation is especially interesting. We deallocate the block of size 16 (of which the application used 15). Because its buddy (also of size 16) is free, we combine them into a free block of size 32. The buddy of this new free block is also free, so we recursively combine the two blocks of size 32 into one of size 64.

9.6 Overallocation Techniques

Up to this point, we have assumed that we can just deny an allocation request if there is no suitable free block available. In some environments (for example, supercomputers) this is a reasonable policy. However, in most general-purpose systems, we want to be able to overextend ourselves a bit. Our justification is that it is very rare that every block of allocated memory is currently in use, in that it is frequently accessed. If we can identify blocks that are not in current use and temporarily store them to disk, then, in most cases, we'll have enough memory to hold everything that is currently active.

In dealing with overallocation, we often encounter the term **virtual memory**. From our previous discussion, it would be quite reasonable for the term virtual memory to refer to the memory addressable with a virtual address. However, we have been careful to use the term virtual memory space to describe this, because in common usage, the

Historical Note: Overlays

In addition to the swapping and paging techniques discussed here, there is another overallocation technique that has been largely lost to history. There was a time when programmers provided their own overallocation management as part of the applications themselves. The most common mechanism was the **overlay**. Suppose the program's structure can be broken up into a number of distinct phases. A compiler is a good example of this. One design calls for the parsing phase to produce an intermediate representation that is then operated on by the code generation phase, and the output of the code generator is operated on by the optimizer. Upon completion of the parser, we no longer need that code to be resident in memory. If we're short of either virtual memory space or physical memory, we can start the program with just the parser code resident, then overlay that code with code for the code generator after the parser is finished. Similarly, when the code generator completes its task, we read the code for the optimizer into memory on top of the code generator. The application itself will read the new code into memory and transfer control to it. The operating system isn't responsible for managing memory except for allocating space for the process as a whole. Like many techniques, overlays were used for several generations of computers, including mainframes, minicomputers, and microcomputers, where the technique was used with operating systems such as CP/M and MS-DOS. Between large virtual address spaces on modern systems and demand paging, programmers are very rarely burdened with implementing overlays today.