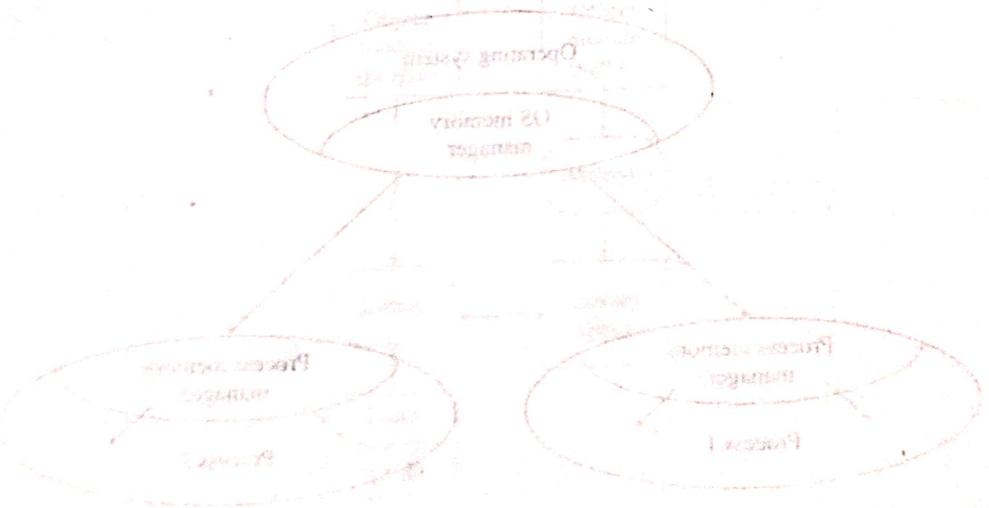
chapter

10

Memory Management

So far, we have talked about processes, the active part of a computation. But a computation also requires memory. In this chapter, we will look at the basic ideas in memory management. In Chapter 11, we will look at virtual memory. In Chapter 12, we will look at page replacement algorithms and some other issues related to memory management.

Main memory is generally the most critical resource in a computer system in terms of the speed at which programs run, and it is important to manage it as efficiently as possible. The operating system is responsible for allocating memory to processes. In this chapter, we will look at the problems of memory management and a range of possible solutions.



10.1 LEVELS OF MEMORY MANAGEMENT

Figure 10.1 shows the two levels of memory management that go on in a running system. The memory manager is the subsystem of an operating system in charge of allocating large blocks of memory to processes. Each process gets, from the operating system, a block of memory to use, but the process itself handles the internal management of that memory.

Each process has a memory allocator for that process. In C++ (and Ada and Pascal), this is the `new` memory allocator, and in C it is the `malloc` memory allocator. These memory allocators do not call the operating system each time they get a memory request. Instead, they allocate space from a large block of free memory that is allocated, all at once, to the process by the operating system. When the per-process memory allocator runs out of memory to allocate, it will ask for another large chunk of memory from the operating system.

In this chapter, we will look at both levels of memory management. We will look at the per-process memory management first. Once we understand memory management inside a single process, we will consider the additional issues that come up in operating system memory management.

10.2 LINKING AND LOADING A PROCESS

In order to understand how memory management inside a process works, we first have to understand how the memory in a process is set up initially, when the process is started. This initialization sets up the memory pool that is allocated from by the per-process memory manager.

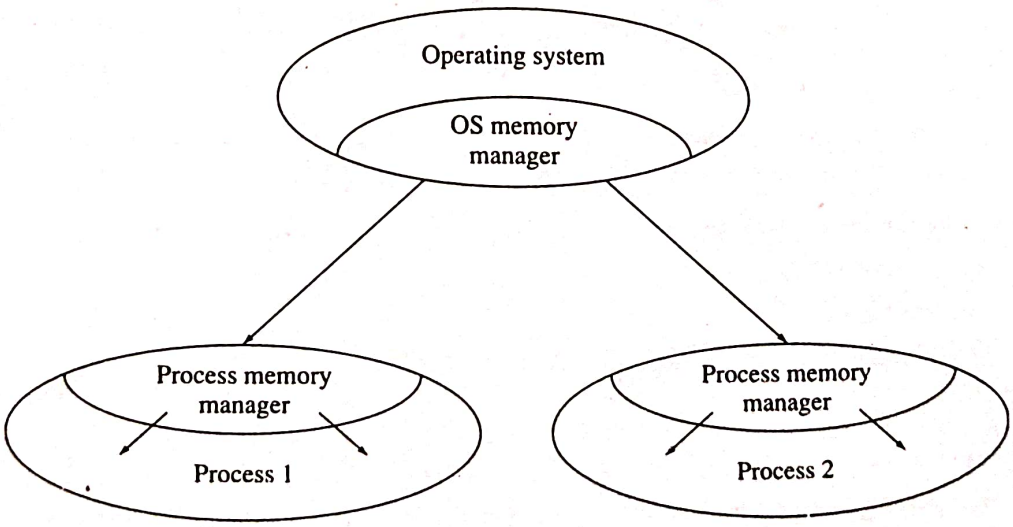


Figure 10.1 Two levels of memory management

A program must go through two major steps before it can be loaded into memory for execution. First, the source code is converted into a load module (which is stored on disk). Second, when a process is started, the load module is loaded from disk into memory.

10.2.1 Creating a Load Module

The process of creating a load module from a source program is shown in Figure 10.2. First, the source code file is compiled by the compiler. The compiler produces an *object module*. Let's look at the contents of an object module, and then we will go back to discussing Figure 10.2 and see how object modules are linked together into a load module.

object module

Object modules Figure 10.3 shows one possible structure of an object module. There are many object module formats. They differ in details, but all contain the same basic information. An object module contains a header which records the size of each of the sections that follow (as well as other information about the object module), a machine code section which contains the executable instructions compiled by the compiler, an initialized data section which contains all data used by

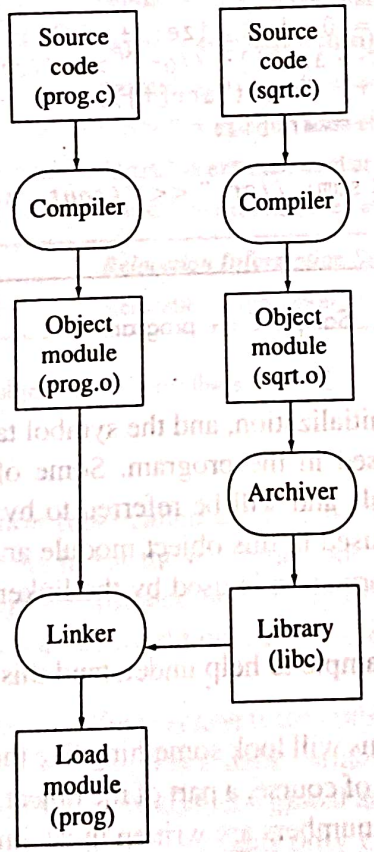


Figure 10.2 Creating a load module

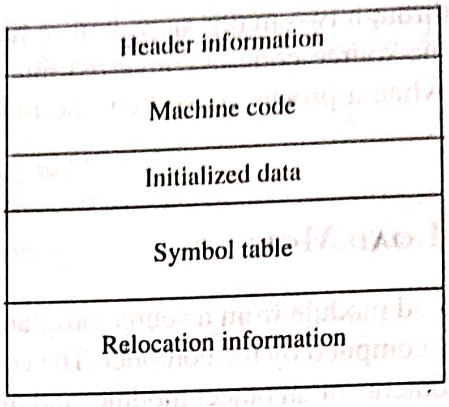


Figure 10.3 Object module format

```
#include <iostream.h>
#include <math.h>
float arr[100];
int size = 100;
void main( int argc, char * argv[ ] ) {
   int i;
   float sum = 0;
   for( i = 0; i < size; ++i ) {
      cin >> arr[i]; //or " >> "(cin, arr[i]);
      arr[i] = sqrt(arr[i]);
      sum += arr[i];
   }
   cout << sum; //or " << "(cout, sum);
}
```

Figure 10.4 Sample C++ program

the program that requires initialization, and the symbol table section which contains all the external symbols used in the program. Some of the external symbols are defined in this object module and will be referred to by other object modules, and some of these symbols are used in this object module and defined in another object module. The relocation information is used by the linker to combine several object modules into a load module.

Let us use a specific example to help understand this process. Consider the program in Figure 10.4.

The object module for this will look something like the one shown in Figure 10.5. The comments fields are not, of course, a part of the object module, but are there in the diagram for explanation. All numbers are written in decimal.

header section

The *header section* contains the sizes necessary to parse the rest of the object module and create the program in memory when it eventually gets loaded.

Offset	Contents	Comment
		Header Section
0	94	Number of bytes of machine code
4	4	Number of bytes of initialized data
8	400	Number of bytes of uninitialized data
12	72	Number of bytes of symbol table
16	?	Number of bytes of relocation information
		Machine Code Section
20	XXXX	Code for top of `for` loop
50	XXXX	Code `arr[i] << cin` statement
66	XXXX	Code for `arr[i] = sqrt(arr[i])` statement
86	XXXX	Code for `sum += arr[i]` statement
98	XXXX	Code for bottom of `for` loop
102	XXXX	Code for `cout << sum` statement
		Initialized Data Section
114	100	Location of `size`
		Symbol Table Section
118	?	"`size`" = 0 (in data section)
130	?	"`arr`" = 4 (in data section)
142	?	"`main`" = 0 (in code section)
154	?	" >> " = external, used at 42
166	?	"`sqrt`" = external, used at 62
178	?	" << " = external, used at 90
		Relocation Information Section
190	?	Relocation information

Figure 10.5 Object module for the sample C++ program

machine code section

The *machine code section* (also called the *text section*) contains the generated machine code. We have put Xs in for the machine code, and put the source code in the comments. We have roughly estimated how many bytes of code will be required for each statement. Different values would only affect the offsets of later items in the object module.

initialized data section

The only item in the *initialized data section* is the constant 100 (named `size`). Note that the variable `sum` is initialized, but it is a stack variable and does not appear in this section. The initialized data section only contains global and static, initialized data.

uninitialized data section

The *uninitialized data section* also contains only global and static data. In this case, that includes only the array `arr`, which is 400 bytes long. Note that the uninitialized data section is not represented explicitly in the object module; only its size is recorded. The initial value of uninitialized data is undefined, so there is

no need to record it. The space will be allocated when the program is loaded into memory.

symbol table
external symbol

The *symbol table* contains two classes of external symbols: undefined symbols and defined symbols. *External symbols* are the symbols in the symbol table. They are distinguished from local symbols that are only used in a single object module.

undefined external symbol

Undefined external symbols are used in this program but defined in another object module. For each of these symbols, we record the symbol name and the places it is used. When the program is linked, these locations will be filled in with the correct value.

defined external symbol

Defined external symbols are defined in this object module, and may be used as undefined symbols in other object modules.[1] For each of these symbols, we record the symbol name and the value of the symbol. We have assumed that each symbol table entry takes 12 bytes. Note that the symbol offsets used are not the offsets in the object module (shown in the left-hand column), but offsets into the section (machine code or data) that the symbol is in. So, `size`, which is at offset 114 in the object module, is at offset 0 in the data section. The same is true for the offsets given for the undefined external symbols. For example, `sqrt` is used in code at offset 82 in the object module, but at offset 62 in the load module (since the code section starts at offset 20 in the object module).

linker
load module

Creating a Load Module Now, getting back to Figure 10.2, the job of the *linker* is to combine one or more object modules and zero or more libraries into a *load module*, which is a program ready to be executed.

library

A *library* is an archive of object modules that are collected into a single file in a special format. There is a library manager (called an "archiver" in Figure 10.2) which manages libraries. The linker knows the library format, and can fetch object modules from the library.

The linker has several jobs:

- Combine the object modules into a load module.
- Relocate the object modules as they are being combined.
- Link the object modules together as they are being combined.
- Search libraries for external references not defined in the object modules.

Here is the process the linker goes through to create a load module.

1. Initialize by creating an empty load module and empty global symbol table.
2. Read the next object module or library name from the command line.
3. If it is an object module, then:
   a. Insert the object module (code and data) in the next available space in the load module. Remember the address (in the load module) where it was loaded.
   b. Relocate the object module to its new load address. Also relocate all the symbols in the object module's symbol table.

[1] With certain compiler options, the symbol table will contain local (internal) symbols also. These are not necessary for linking, but are used by symbolic debuggers.

c. Merge the object module's symbol table into the global symbol table. Steps *d* and *e* describe this process in more detail.
d. For each undefined external reference in the object module's symbol table:
   (1) If the reference is already defined in the global symbol table, then write its value into the object module you just loaded.
   (2) If the reference is not yet defined in the global symbol table, then make a note to fix up the links when the symbol is defined (during the loading of a later object module).
e. For each symbol definition in the object module's symbol table, fix up all previous references to this symbol noted in the global symbol table (these are references in object modules loaded earlier).

4. If it is a library, then:
   a. Find each undefined external reference in the global symbol table.
   b. See if the symbol is defined in any of the object modules in the library.
   c. If it is, then load the object module from the library as described in Step 3 for an object module listed on the command line.
5. Go back to Step 2.

The linker takes one or more object modules and combines them into a load module. In doing this, it has to relocate each object module (except the first one) and link all the object modules together. We will discuss the relocation and linking steps in the next few sections.

Load Module Sections The linking algorithm we described above is inaccurate in one detail. We implied that the object modules are placed sequentially in the load module as they are loaded. Actually, the three sections of each object module (the code, initialized data, and uninitialized data) are separated, and the sections of each type are loaded together in the load module. That is, all the code sections are loaded sequentially together in the first part of the load module, all the initialized data sections are loaded next, and finally all the uninitialized data sections are placed together. The result is that the load module has these same three sections: code, initialized data, and uninitialized data.

This makes the relocation a little more complicated, since each section has its own load address. It also makes management of the memory in the load module more complicated, since you have three sections, each of which is growing.

Relocation of Object Modules The relocation section records the places where symbols need to be relocated by the linker. When an object module is being compiled, the compiler must assume some specific address where it will be loaded because certain places in the program require absolute addresses (that is, addresses that are not relative to a machine register but are a constant value). The compiler usually assumes that the object module will be loaded at address 0, but it records (in the relocation information) all the places that assume a load address of 0.

When the linker combines several object modules, only one of them can actually be loaded at address 0. The rest have to be relocated to higher addresses. When the linker does this, it goes into the object module and modifies all the absolute addresses to account for the new starting address. If the assumed load address is 0, this comes down to adding the real load address to these locations. Otherwise, we add in the difference between the actual load address and the assumed load address.

The linker creates a load module that it assumes will be loaded at address 0. We can think of each object module as an address space starting from 0 and going up to the size of the object module. The linker combines these address spaces into a single address space. In doing so, it must change some of the addresses in the object modules—the ones that assumed that the object module started at address 0.

Figure 10.6 shows two object modules being combined into a load module. The first object module is 110 bytes long, and so has an address space of 0 to 109. It is loaded first into the load module, and so it keeps its addresses, 0 to 109. The second object module is 150 bytes long, and so has an address space of 0 to 149. At address 136, there is a data cell named `x`, and at address 80, there is a reference to that address, and so the value 136 is used. When this object module is loaded into the load module, it is placed after the first object module, and so it occupies ad-

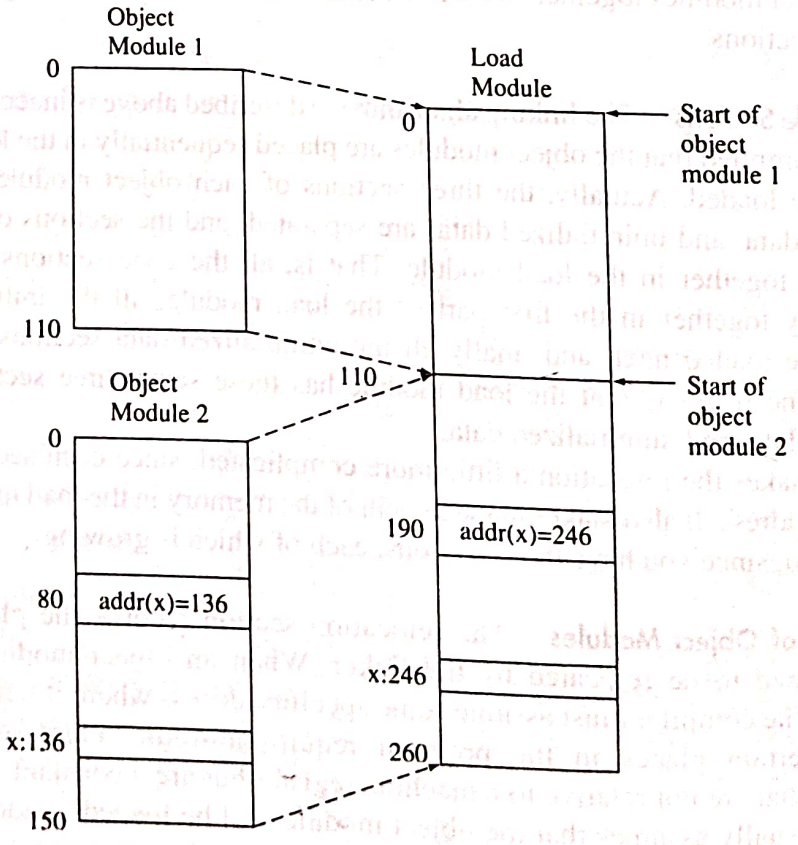


Figure 10.6 Relocating an object module into a load module

dresses 110 to 259 in the address space of the load module. The data cell named x is now at address 246, and so the reference to it (now at address 190) must be changed from 136 to 246.[2]

The compiler (or assembler) that creates an object module makes a record of each place where the address in the object module assumes a loading address of 0. When the object module is loaded into a load module, the linker goes to all these places and changes the value based on the real load address, which is now known. The process is straightforward and depends on the the appropriate tables being created in the object module.

Figure 10.6 only shows one value being relocated, but, in general, there will be a number of locations requiring relocation. The amount of relocation necessary depends on the hardware architecture. Modern architectures do not require much relocation, but still some absolute addresses must be used.

The location we are discussing here is called *static relocation*. Static relocation is done once, by the linker, as the load module is created. When the load module is executed on our hardware, each address will have the contents of the *base* register added to it before it is sent to the memory. This is called *dynamic relocation* because it happens each time the address is used during the execution of the program.

static relocation

dynamic relocation

The load module the linker creates (described in the next section) will be dynamically relocated when it is executed using a base register, but the object modules must be statically relocated because they all share the same base register. If we had a separate base register for each object module, then no static relocation by the linker would be necessary. This is the case in a segmented system (see Section 12.11).

Linking of Object Modules A linker combines several object modules into a single load module. The reason you want to combine the object modules is that they refer to each other. One object module may call procedures in another object module, or use data defined in another object module. Each object module may define some external symbols (procedures or data) that will be referenced by other object modules. When the compiler compiles a program into an object module, it builds a symbol table in the object module that contains the undefined external references that are used in the object module and the external symbols that are defined in the object module. The compiler compiles a zero for the external references to allocate space for the reference. Later, the linker will write in the correct value.

Another function of the linker is to link up these undefined external references with the external symbols to which they refer, and this process is called *linking*. As the linker loads and relocates the object modules, it also creates a global symbol table for the entire load module. All the external symbols are kept in this table, which is just the combination of the symbol tables of all the individual object modules. When an external reference is encountered, the linker looks it up in this global table. If it is found, then the linker writes the correct value into the space for

linking

[2]Again, this diagram is misleading because, in reality, the code sections of the object modules will be loaded together, as will the two data sections.

the reference in the load module. If the symbol is not found, the linker makes a note to link the reference later, when the external symbol is defined in a later object module. Figure 10.7 shows this process.

When the linker is instructed to load a library, it does not load all the modules in the library. Instead, it looks through its global symbol table and finds all the undefined external references. It then goes through the library and loads any object modules that define one or more of these external references.

Differences between Linking and Relocation It is important to remember that the linker is doing two distinct and separate tasks when it creates a load module out of object modules. We will review them here to be sure that you understand the differences. The first task is relocation. Relocation is the combining of the address spaces of all the object modules into the single address space of the load module. Relocation involves modifying each code and data location that assumes the object module starts at address 0. Relocation consists of adding the actual starting address of the object module in the load module to each of these locations.

Linking is the modification of addresses where one object module refers to a location in another object module (such as when code in one object module calls a procedure in another object module). When such an external address is required, the compiler has no idea what the correct value is, and so just writes in a zero and records the location of the address in the object module symbol table. When the linker puts

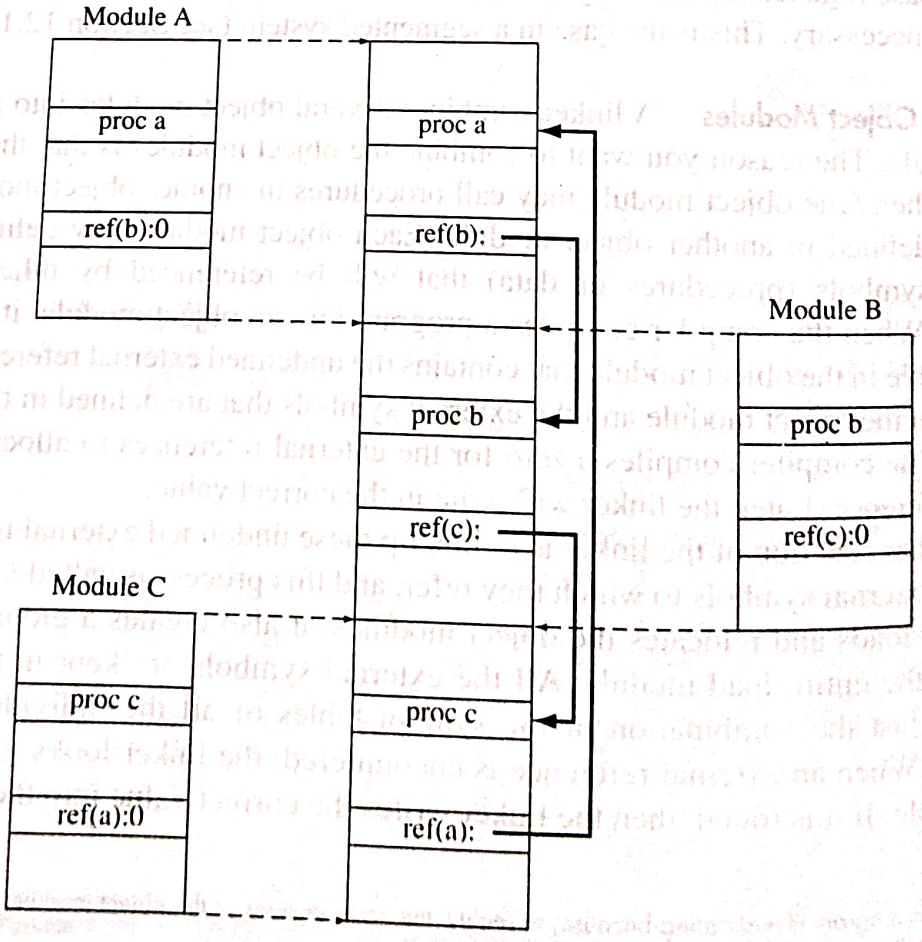


Figure 10.7 Linking object modules in a load module

all the object modules together, it knows all the external addresses, and it can correctly set these external addresses.

Load Modules The format of a load module is similar to that of an object module. A load module, like an object module, will have a code section, an initialized data section, an uninitialized data section (not actually stored in object or load modules but its size is recorded), a symbol table section, and a relocation section. The relocation information is not necessary since most systems use dynamic relocation, and so programs are loaded at (logical) address 0. Since that is what the linker assumes, no further relocation is necessary. The symbol information is no longer necessary, but is kept with the load modules for use by symbolic debuggers.[3]

10.2.2 Loading a Load Module

When a program is executed, the operating system allocates memory to the process (we'll see later in this chapter how it does this) and then loads the load module into the memory allocated to the process. Figure 10.8 shows how this loading is done. The executable code and initialized data are copied into the process' memory from the load module. In addition, two more areas of memory are allocated. One is for the uninitialized data. This area is allocated but does not need to be initialized. Some operating systems will initialize this area to all zeros (UNIX does this).[4] The second area is for the stack. Programming languages usually use a run time stack for keeping information about each procedure call (called activation records or procedure contexts). The stack starts out empty, and so there is no need to store it in the load module.

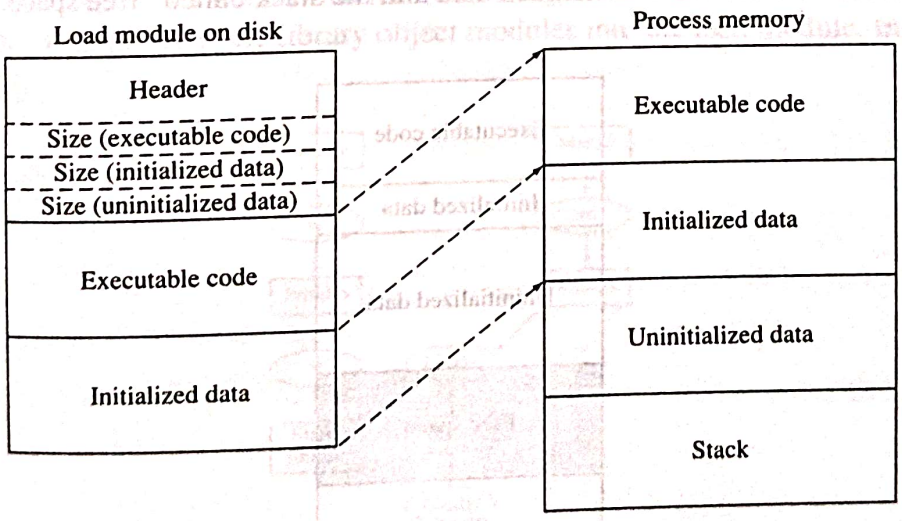


Figure 10.8 Loading a program into a process

[3]UNIX has a command called `strip` which removes the symbol table information from load modules. You can use this to save disk space if you are sure you will not want to symbolically debug the program.

[4]This belies the name "uninitialized data" so we can think of this as data that is not initialized to a specific value by the compiler. In a computer system, no memory is really "uninitialized" since the bits there must have some value at all times. By "uninitialized" we generally mean that it does not have an easily predictable value.

Before the loader does any of this, it computes how much memory the process will require and requests that much memory from the operating system. The memory required is the sum of the sizes of the executable code, the initialized data, the uninitialized data, and the stack. The first three of these sizes are obtained from the load module. The loader reads the load module header first, then allocates the memory, and then completes the loading.

How big is the stack? The loader has a default initial size for the stack (and most loaders have a command line option to change this default size). It starts the stack out at this size. When the stack fills up, more space will be allocated to it. This process will continue until the stack reaches some predefined maximum size. We will defer a discussion of exactly how this happens until we talk about memory mapping.

Object modules have been compiled, but not linked, and cannot be executed. Load modules have been compiled, linked, and are ready to be executed.

10.2.3 Allocating Memory in a Running Process

Most programming languages allow you to allocate memory while the program is running. We saw in Section 10.1 that this is done with a call to the memory allocator, which is called `malloc` in C and `new` in C++, Pascal, and Ada. Where does this memory come from? To see this, we have to revise our diagram of a loaded process from that given in Figure 10.8. Figure 10.9 shows the loaded module, with a new memory area between the uninitialized data and the stack called "free space."

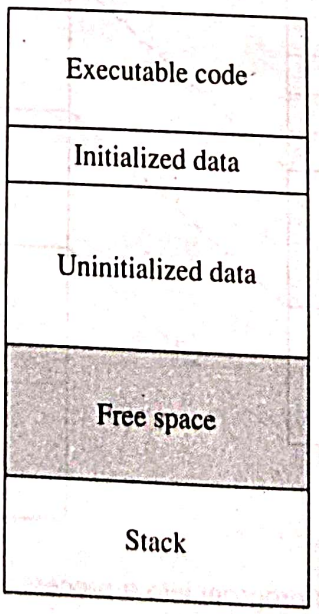


Figure 10.9 Memory areas of a running process

This free space is the memory that the per-process memory allocator allocates. In Sections 10.4 to 10.10 we will talk about how it does this allocation, and that will complete our discussion of the first level of memory management. After that, we will talk about memory management by the operating system. But before we go on to these topics, we will discuss some other methods used in program loading.

10.3 VARIATIONS IN PROGRAM LOADING

We discussed the creation and loading of load modules in Section 10.2. Figure 10.10 shows the process of linking and loading again.

These days, some programs are very large. There are two problems with large load modules. The first is that they take a lot of space on the disk, and the second is that they take a long time to load. Operating systems often use techniques to alleviate each of these problems.

10.3.1 LOAD TIME DYNAMIC LINKING

The large executable problem is especially apparent these days with window systems. It takes a lot of code to run the window interface, and most of this code is in the window libraries provided with the window systems. For example, a minimum X windows program is about 500K bytes long. About 450K of this is window libraries. If you have 11 such programs, you have 10 times 450K, or 4.5M bytes, of duplicated information. This can start filling up your disk pretty quickly.

load time dynamic linking

One solution is to use a technique called *load time dynamic linking*. When the linker is linking from a library that has been designated as a load time dynamic link library, it does not insert the library object modules into the load module. Instead, it

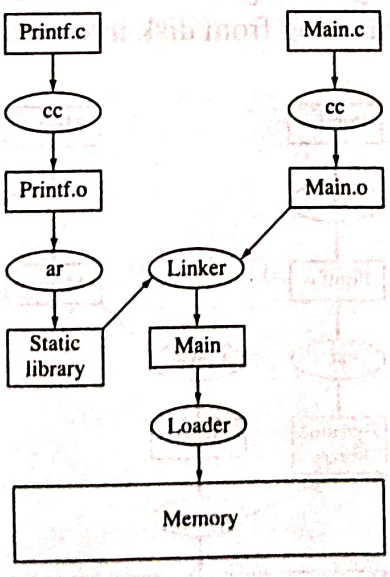


Figure 10.10 Normal linking and loading

inserts information about where to find the library, which of the library modules to load, and where to load them in the program's address space. When the program is actually loaded, the run time loader notices these load time dynamic link library references, goes to these libraries, and completes the library loading process at that time. Figure 10.11 shows this process.

Loading with load time dynamic link libraries takes a little longer, since the loader has more work to do, but it saves a lot of disk space that would have been wasted with duplicate copies of library modules.

Load time dynamic linking will only work if the libraries are in exactly the same place in the file system when the program is loaded as they were when the program was linked. System files like libraries seldom move, and so this is generally the case, but occasionally there is a system reorganization that moves things around and, after this happens, it is possible that some load modules that use load time dynamic link libraries will fail to load correctly.

Another problem is new versions of libraries. Most linkers record the version of the load time dynamic link library they looked at and, when object modules are finally loaded, if the version has changed, a warning message will be generated. Usually, the load will continue, and the resulting program will run correctly. But if it does not, then the version change is one place to look for a reason for the failure.

If you move an executable program from one machine to another, the libraries may be in different locations even though it is the same operating system on the same type of hardware. System managers have flexibility in how they lay out the system files in the file system, and often there are variations between installations.

10.3.2 Run Time Dynamic Linking

What about the second problem? A big program might take a long time to load, since there is so much information to read into memory from disk. Most X windows executables are millions of bytes long. Even with load time dynamic link libraries, there is still the problem of moving from disk into memory.

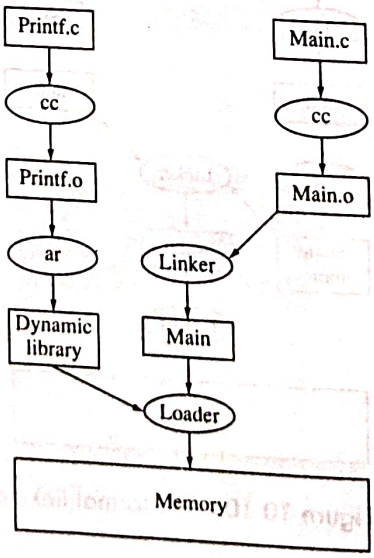


Figure 10.11 Load time dynamic linking

Many newer (and a few older) operating systems provide a service to deal with this problem, and it is called *run time dynamic linking*, or simply, *dynamic linking*. Dynamic linking is the idea of load time dynamic linking taken one step further. With load time dynamic linking, you defer the linking of library modules until just before a program is about to begin execution. With dynamic linking, you defer the loading to the last possible moment, that is, until the module is actually needed for the execution of the program. Figure 10.12 shows this process.

run time dynamic linking

Here is how you can do this. You add code to the program to check if a module has been loaded. This can be done efficiently by accessing the module indirectly through a pointer. The pointer can initially be set to interrupt to the dynamic linker. The first time you call a procedure in the module, the call fails because the indirect pointer is not set. This starts up the dynamic linker, which links the module into the program, loads it into memory from the disk, and fixes the indirect pointer so that, next time, the procedure will be called without an interrupt.

This means that you defer loading until you actually need the module. But how is this an advantage? It seems like it will take the same amount of time to load the module, no matter when you load it. You are not really saving load time, but just moving it around.

Let's take a specific example to explore the issues. Suppose you have a program that has 1M of base code and uses five library modules, each taking 200K. Suppose further that the program always uses the code in all five library modules. With dynamic linking, the program will start up twice as fast. The initial delay will be less, but there will be five more small delays distributed throughout the execution of the program as the five modules are loaded. Often this, in itself, is an advantage. People will notice an extra three-second delay at startup, but may not notice six half-second delays at various places during program execution.

In addition, with static loading the program requires 2M bytes to run during its entire execution, but with dynamic linking the program will use less memory for a while, until all five modules are loaded. If some of these modules are not used for a while, then it will not request that memory until it is actually needed.

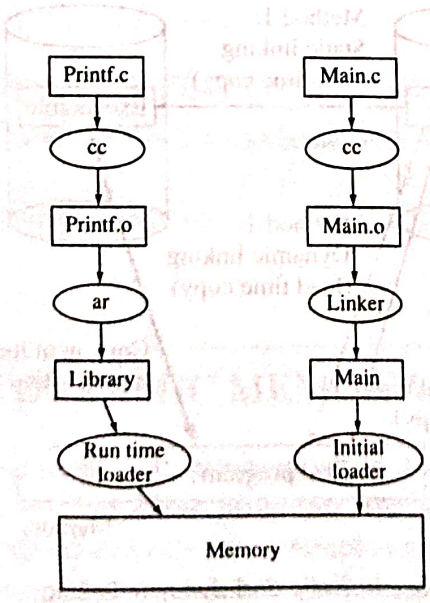


Figure 10.12 Dynamic linking of modules

But dynamic linking might actually save loading a module, as well as saving memory space throughout the execution of the program. Suppose that each run of the program only uses one or two of the library modules—different ones each time, depending on the input data to the program. In this case, dynamic linking is more efficient. The loading takes less time, since only one or two modules are loaded instead of all five.

Finally, dynamic linking always loads the most recent version of the module, since it is not searched for until it is actually needed. This is especially true if we compare it with static loading without load time dynamic linking. A load module might have been linked several months ago, and some of the modules may have newer versions.

Static loading takes a little bit less time, but dynamic loading saves memory during execution, allows faster startups, and always gets the most recent version of modules. All in all, dynamic linking has a lot to recommend it.

Figure 10.13 shows the difference between these techniques by showing the three different times (and two different places) the library routines are copied to. The following table summarizes the these techniques:

Method	Link Time	Load Time	Use Time
Static linking	Copy in libraries.	Load in the executable.	—
Load time dynamic linking	Remember where libraries are.	Load in executable. Copy in libraries.	—
Run time dynamic linking	Remember where libraries are.	Load in executable.	Copy in libraries

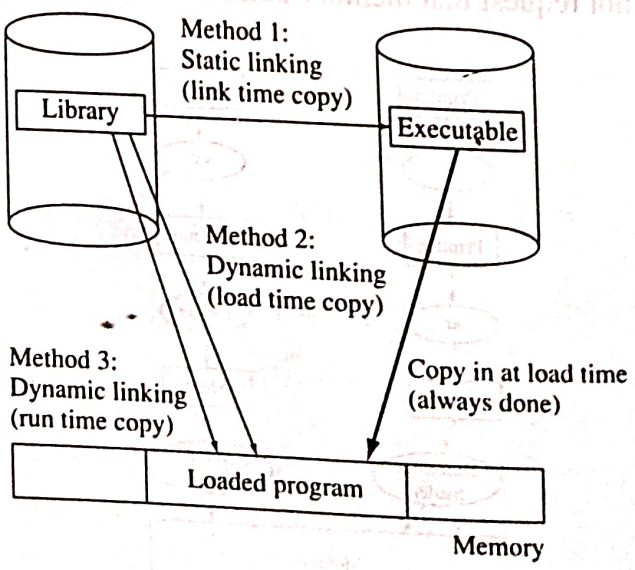


Figure 10.13 Static and dynamic linking

DESIGN TECHNIQUE: STATIC VERSUS DYNAMIC

We saw how run-time dynamic linking has several advantages over static linking. The static versus dynamic tradeoff is a common one in operating systems and in computer science in general. Programs are static and processes are dynamic. Compilation is static and interpretation is dynamic. Memory in a program can be allocated statically or dynamically.

In many cases, static solutions are faster and dynamic solutions are more flexible, but this is not always the case.

Dynamic solutions are a form of late binding.

To read more about static versus dynamic tradeoffs, see Section 13.3.

We can do some simple computations to see the tradeoffs between these methods. Suppose we have the following parameters:

- Program size: 1 Mbyte.
- Library modules: 4 modules of 1/2 Mbytes each.
- Total program size: 3 Mbytes.
- Time to load 1 Mbyte: 1 second.
- Time to invoke the loader: 100 milliseconds.

The following table breaks down the delays in each case.

Method	Disk Space	Load Time	Run Time (4 used)	Run Time (2 used)	Run Time (0 used)
Static linking	3 Mbytes	3.1 sec.	0	0	0
Load time dynamic linking	1 Mbyte	3.1 sec.	0	0	0
Run time dynamic linking	1 Mbyte	1.1 sec.	2.4 sec.	1.2 sec.	0

Dynamic solutions are better when the demand is uncertain. Static solutions are better when the demand is known.

10.4 WHY USE DYNAMIC MEMORY ALLOCATION?

Why does a program need to allocate memory while it is running? In fact, most programs do not, but many do. For some programs, the amount of memory they will need is not known until run time. Let's take a simple example of a program that reads in the description of a large graph, and decides whether the graph is connected or not.

The graph can be represented with a linked network of node structures. Suppose the node structure is 16 bytes long, and there is one for each node in the graph. We will not know how many nodes the graph contains until we read in the description of the graph. There might be 1,000 nodes or there might be 500,000 nodes.

Suppose we tried to avoid dynamic memory allocation for graph nodes. If we assume a maximum graph size of 500,000 nodes, we would have to allocate (16 × 500,000) 8,000,000 bytes for graph nodes each time we run the program. Since most graphs will be smaller than that, some or most of this space will be wasted most times the program is executed. It is more efficient to allocate node space as we need it.

This is a simple case because all the nodes are the same size. If all the memory requests are for the same size of block, then the dynamic memory allocation problem is very simple. In this case, we would take all of the free memory and divide it up into 16-byte blocks. We would link these blocks into a linked list, and allocate nodes from the list as needed. In some graph problems, nodes are deleted, so we might free up graph nodes. In this case, they would be linked back on to the list of free nodes.

This situation (where all the blocks required are the same size) is the simplest dynamic memory situation and, in this case, dynamic memory allocation is easy. But this is not a typical situation. Usually, a program will need to allocate different types of structures of different sizes. In this case, the solution to the problem is more complicated. In the next sections, we will characterize the problem in a general way, and then look at some solutions.

*10.5 THE MEMORY MANAGEMENT DESIGN PROBLEM

The design problem we are presented with is illustrated by Figure 10.14, and can be summarized as follows:

- We have a large block of memory, say N bytes.
- Requests for subblocks of the large block come in at unpredictable times.
- All requests are for from 1 to N bytes of memory.
- All requests must be satisfied eventually.
- When a memory block is available, we allocate it to some request. The memory block is in use for a while, and then returned to the system.
- The goal is to have the average amount of memory allocated over some time period that is as large as possible.
- A constraint is that the memory allocation algorithms should not use too much processor time or memory space to run.

We start with a large chunk of memory. We will receive a series of requests for blocks of the memory. A request can be satisfied with a block of memory that is at least as large as the size requested (it can be larger, but the excess will be wasted until the block is returned to the allocator). Once the request is satisfied, the memory is in use for a while, and then it is returned to the allocator. We assume that there will usually be a queue of

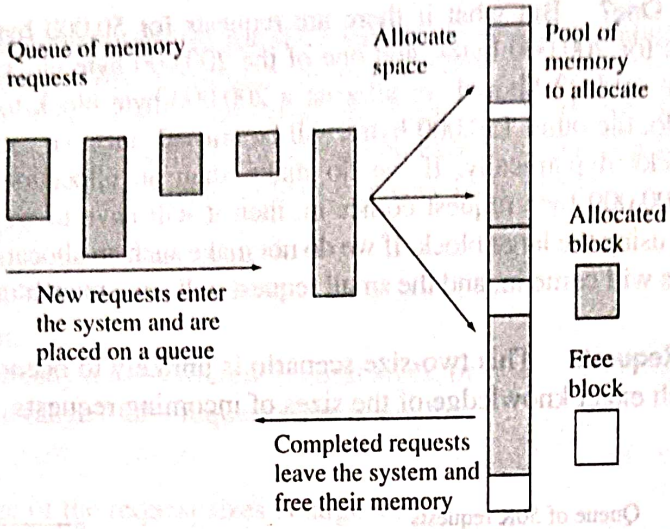


Figure 10.14 The memory allocation problem

memory requests, that is, that there is more demand for the memory than memory available. The goal is to use the memory as efficiently as possible, that is, to have as much of the memory in use at a time as possible. We also want to be sure that no request has to wait forever because a sufficiently large block of memory never becomes available.

*10.6 SOLUTIONS TO THE MEMORY MANAGEMENT DESIGN PROBLEM

As in most design problems, there is not one best solution for all cases, but rather a range of solutions, each with its good and bad points and each of which is best in some situations. In any particular design situation, there will be a best solution for that situation, but often it is hard to determine which is best without implementing several and testing them.

There are two major issues in a memory allocation algorithm: memory organization and memory management. Memory organization is how the block of memory is divided up into subblocks for allocation. You can divide up the memory once and for all, before any bytes are allocated (the static method), or divide it up as you are allocating it (the dynamic method). Memory management (or allocation policy) is the decisions about which to allocate to a request.

10.6.1 Static Division into a Fixed Number of Blocks

Suppose the large block of memory we are allocating is 500,000 bytes long, and we know that half of the requests will be for 50,000 bytes and half will be for 200,000 bytes. We divide the memory into two blocks of 200,000 bytes and two blocks of 50,000 bytes. We will keep two queues, one for each size of block. Incoming requests will be put on the queue corresponding to the number of bytes they need. As you can see, the implementation of this method is very easy. Figure 10.15 shows this solution.

Two Queues or One? But what if there are requests for 50,000 bytes waiting, no waiting requests for 200,000 bytes, and one of the 200,000 byte blocks is unused (as shown in Figure 10.16)? Should we allocate a 200,000-byte block to a 50,000 byte request? If we do, the other 150,000 bytes will be unused since we cannot change the size of the blocks dynamically. If we do make such an allocation, and the next millisecond a 200,000 byte request comes in, then it will have to wait for the small request to finish using the large block. If we do not make such an allocation, then maybe no large requests will come in, and the small request will have to wait unnecessarily.

Variable-Sized Requests This two-size scenario is unlikely to occur. Normally, we do not have such exact knowledge of the sizes of incoming requests. A more typical

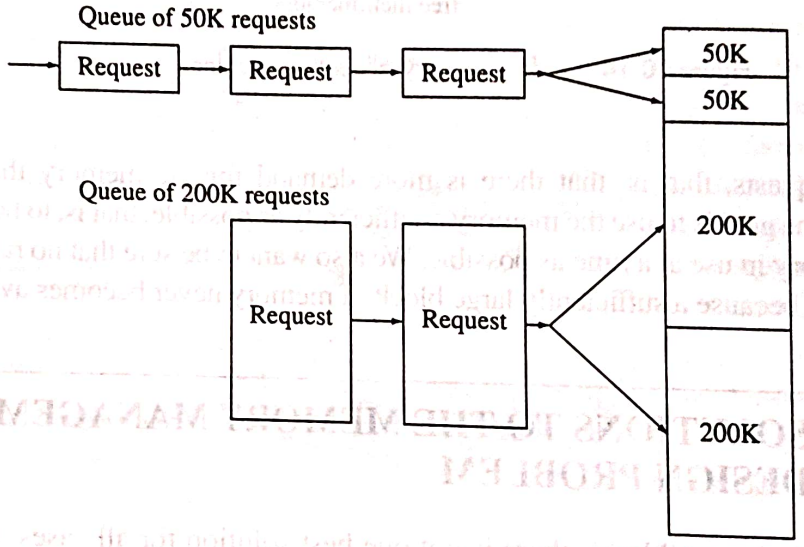


Figure 10.15 Allocation with a queue for each size of block

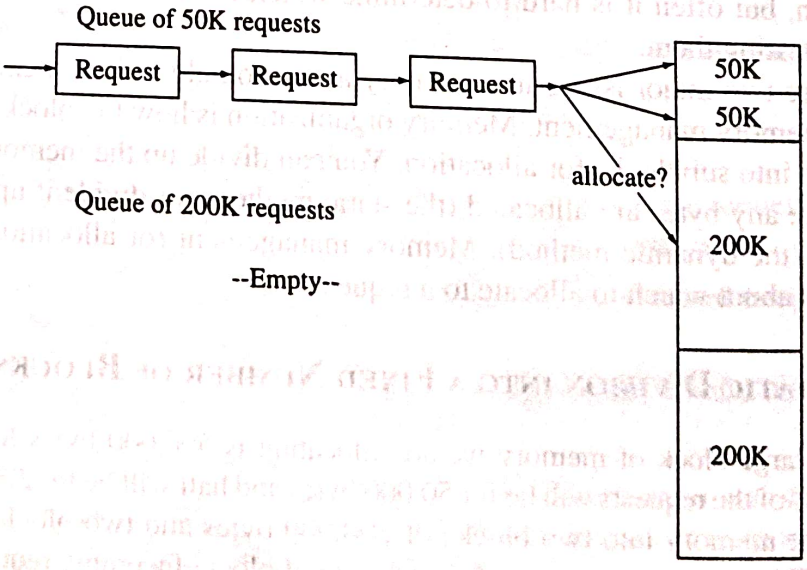


Figure 10.16 Allocate a large block to a small request?

situation is that the requests are for sizes varying over a large range. For example, maybe they are normally distributed around 100,000 bytes (see Figure 10.17.) In these cases, you have to pick a few sizes and hope they work out. You must have a partition big enough for the largest request you expect to get. If a few requests are very large, then you must allocate at least one very large block.

Static division is:

- Easy to implement.
- Good when:
  -There are only a few different request sizes, or
  -When the range of the request sizes is small.
- Bad when:
  -The range of the request sizes is large, or
  -Some requests are very large.

10.6.2 Buddy Systems

We have already noted that, when all requests are for the same size block, then the memory allocation problem is easily solved. You statically divide up the memory into blocks of that size and keep them on a list.

If you have two different sizes of requests and you have some idea of the distribution of the requests for the two sizes, you can statically divide up the memory into blocks of two different sizes and keep a list for each one.

A problem with this technique that we mentioned before is that you might have a large block and a small request, and you have to decide whether to allocate the large block to the small request or not. There is a class of solutions, called buddy systems, to deal with this problem.

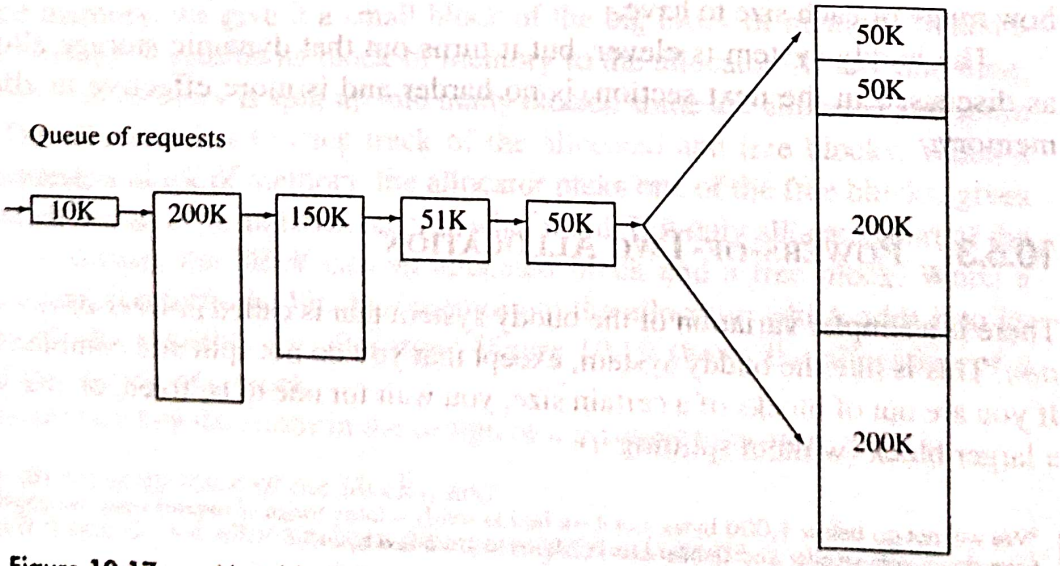


Figure 10.17 Variable-sized requests for memory

Suppose we have 500,000 bytes of memory, and requests of sizes 50,000 bytes and 100,000 bytes. We divide the memory up into five blocks of 100,000 bytes, and put them on a list. We create another list of free blocks of 50,000 bytes, which will start out empty. When a request comes in for a 50,000-byte block and the 50,000 byte list is empty, we look at the 100,000-byte block list. If there is a free block on that list, we divide it up into two blocks of 50,000 bytes, allocate one to the request, and put the other one on the 50,000-byte block list. When a 50,000-byte block is returned, we check to see if the other 50,000-byte block that made up the 100,000-byte block is also free. If so, the two are taken off the 50,000-byte block list, joined again, and the resulting block is put on the 100,000-byte block list. Figure 10.18 shows a series of block allocations and frees.

buddy system

This method allows us to use larger blocks to satisfy smaller requests without wasting half the block. The generalization of this technique is called a *buddy system*. In the buddy system, there is a free block list for each power of two. Say we have 512,000 bytes. Then there will be free block lists for block sizes 512,000 bytes, 256,000 bytes, 128,000 bytes, 64,000 bytes, 32,000 bytes,16,000 bytes, 8,000 bytes, 4,000 bytes, 2,000 bytes, and 1,000 bytes.[5] They all start off empty except for one block on the 512,000 byte block list. When a request is received, you round it up to the next power of two and look on that list. For example, a request for 25,000 bytes will be satisfied with a block of size 32,000 bytes. If that block list is empty, then you try the next-larger power of two, and divide a free block there into two blocks. If you fail to find a block in that list, you keep going up until you do find one, or until you run out of lists, in which case the request must wait.

This method allows you to keep the simplicity of the statically allocated blocks, but handles the difficult cases of large requests and of running out of blocks of a certain size. One problem is that you can waste a lot of space rounding up to the next power of two. For example, a block of 10,000 bytes requires a block of 16,000 bytes, and so 6,000 bytes (37.5 percent) are wasted.

A buddy system is a combination of static and dynamic decisions. We decide statically which sizes of blocks to allow (powers of two), and we decide dynamically how many of each size to have.

The buddy system is clever, but it turns out that dynamic storage allocation, as discussed in the next section, is no harder and is more effective in allocating memory.

10.6.3 Powers-of-Two Allocation

powers-of-two allocation

There is a simpler variation of the buddy system that is called *powers-of-two allocation*. This is like the buddy system, except that you do not split and combine blocks. If you are out of blocks of a certain size, you wait for one to be freed, or else you use a larger block (without splitting it).

[5] We will not go below 1,000 bytes, but if we had to satisfy a large range of request sizes, we could continue down with smaller and smaller blocks, down to just a few bytes.

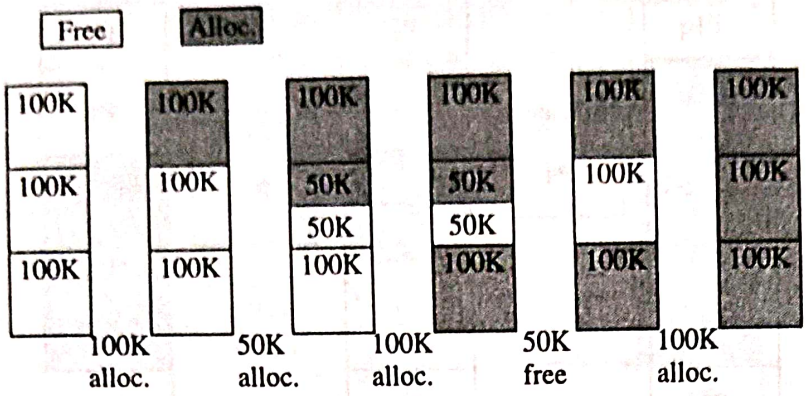


Figure 10.18 The buddy system

The powers-of-two method is not a good system for general dynamic memory allocation, but it becomes useful when combined with a paging system (see Chapter 11). We will look at it again in Section 10.10.

*10.7 DYNAMIC MEMORY ALLOCATION

Since we cannot predict the sizes of processes, the static solution is not a good one for the management of memory within a process, or for the management of process memory in an operating system. So we have to dynamically divide the memory into blocks in response to the requests that come in. This is an interesting problem which we will examine in this and the next two sections.

dynamic memory allocation

Dynamic memory allocation is a classic problem in computer science. It is a task of some complexity, but it is much studied and the techniques are well understood.

The basic idea is simple. We have a large block of memory. When a process needs some memory, we give it a small block of the big block of memory. When a process is finished, it returns its block of memory to the allocator. At any one time, the large block of memory is split up into many blocks; some are allocated and some are free. The allocator has to keep track of the allocated and free blocks. When a process requests a block of memory, the allocator picks one of the free blocks, gives the process all or part of it, and then records what it did. If it only allocated part of the free block, it divides the block into an allocated block and a free block. When a process finishes, it returns the block of memory to the allocator, which adds it to its set of free blocks available for allocation. Figure 10.19 shows the allocation of a block and the freeing of a block.

There are two key decisions in the design of a memory allocator:

- How do we keep track of the blocks, and
- Which block do we allocate from when a request comes in.

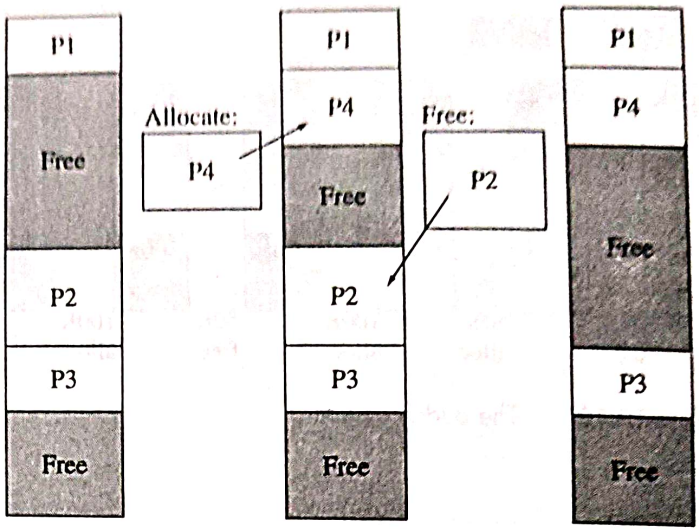


Figure 10.19 Allocating and freeing a memory block

fragmentation

An important issue in dynamic memory allocation is called *fragmentation*, that is, memory becoming divided up into so many small blocks that none of them are useful because they are all too small to satisfy any pending request. One simple precaution is to ensure that there are never two free blocks right next to each other. If this happens, the allocator will combine them into one bigger free block. The method we use to keep track of the blocks is usually responsible for doing this. Even doing this, there might still be lots of small blocks between allocated blocks.[6] We will come back to the fragmentation problem as we talk about allocation methods.

*10.8 KEEPING TRACK OF THE BLOCKS

There are two major methods of keeping track of blocks: a list and a bitmap. The list method is the most popular by far, and is also the most obvious.

10.8.1 THE LIST METHOD

block list

We keep a linked list of all the blocks of memory. We will call this the *block list*. When we need to allocate a block of memory, we go down the block list to find a suitable free block. Once we have found one, we either allocate the entire block, or divide it up into one block of the requested size and a free block containing the rest of the original block. When a block is freed, we change its status on the block list and combine it with the surrounding free blocks, if there are any. Figure 10.20 shows the list method of keeping track of memory blocks. Figure 10.21 shows the block list after block P5 is allocated, and Figure 10.22 shows the block list after block P3 is freed.

[6]Note, however, that there can be at most one more free block than allocated blocks, since two free blocks cannot be next to each other. We will come back to the fragmentation problem as we talk about allocation methods.

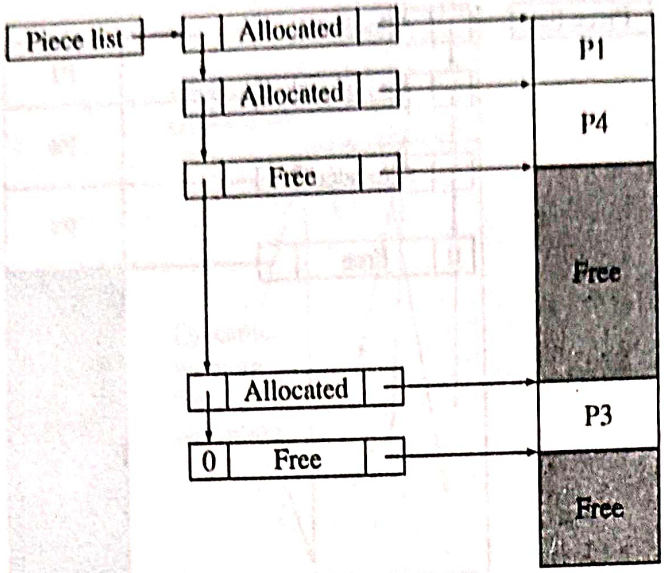


Figure 10.20 The block list method

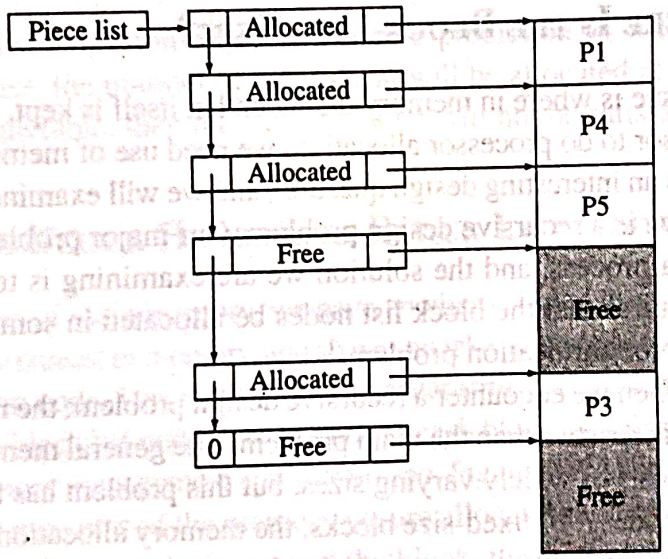


Figure 10.21 The block list after allocating P5

10.8.2 Keeping Allocated Blocks on the Block List

It is not strictly necessary to include the allocated blocks on the block list since we never do anything with them. They are each allocated to a process and will be returned when the process completes (or sooner). However, we still want to keep the allocated blocks on the list for error checking. A program might erroneously try to free memory that has not been allocated to it. If the allocator believes the errant program, the block list will become inconsistent, and the same area of memory might be allocated twice at the same time. Some allocators do not check for this, and it is possible to crash them by improper memory freeing. It is best to avoid this problem and keep track of all blocks, free and allocated.

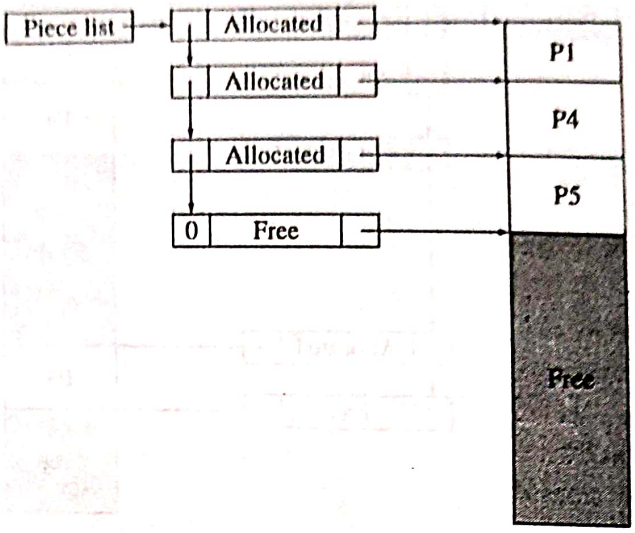


Figure 10.22 The block list after freeing P3

10.8.3 Where Is the Block List Kept?

One important issue is where in memory the block list itself is kept. Just as we needed use of the processor to do processor allocation, we need use of memory to do memory allocation. This is an interesting design question, and we will examine it in some detail.

What we have is a recursive design problem. Our major problem is how to allocate memory in a process, and the solution we are examining is to use a block list. This solution requires that the block list nodes be allocated in some way, and so we have another memory allocation problem.[7]

As is usual when we encounter a recursive design problem, the recursive problem has different characteristics than the main problem. The general memory management problem had requests of widely varying sizes, but this problem has fixed-size blocks, that is, block list nodes. For fixed-size blocks, the memory allocation problem is easy: statically divide the available memory into fixed-size blocks and keep them on a stack.

But then the problem is how many nodes allocate to the block list, that is, how much available memory will there be? There must be a node for each block in the system, but the number of blocks in the system will vary widely depending on how big the memory requests are.

Figure 10.23 shows how this works. Part of the memory is reserved for the block list, and the rest of the memory is available for allocation to satisfy requests for memory.

The space for block list nodes must be taken from the free space for allocation. If we allocate too many block list nodes, then we may run out of memory while a lot of space is wasted on unused block list nodes. If we allocate too few block list nodes, then we may run out of block list nodes and be unable to allocate free memory even though it is available.

[7] We have seen a recursive design problem before, when we used messages or semaphores to implement process-level mutual exclusion, and then found we needed operating-system-level mutual exclusion to implement messages or semaphores.

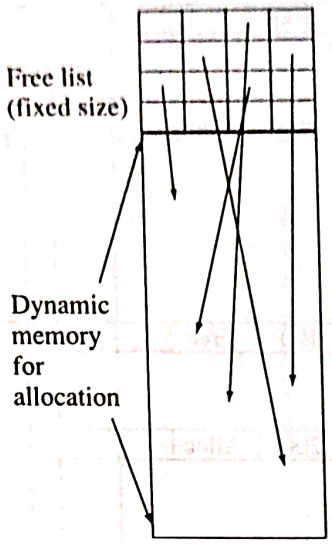


Figure 10.23 Reserving space for the block list

Again, the static solution requires accurate prediction of the behavior of the process (in this case, the number of blocks that will be allocated at one time). If this behavior is unpredictable, then the static solution will not be satisfactory.

10.8.4 Using Block Headers as Block List Nodes

For the special case of main memory, we have another solution that is commonly used. The blocks already consist of memory, and so we can take a little memory from each block to use as a block list node. This method puts a header structure at the front of each block, which is used as a block list node. It is like taxing each block by a fixed number of bytes to pay for the memory management space overhead. Figure 10.24 shows this structure.

This method uses part of the memory we are allocating to user processes for the block list. This header records the size of the block, the status of the block, and the list pointer(s). When the block is allocated, you give the process only the area after the header, since the memory allocator needs to maintain this header information even when the block is allocated. This is a nice method since it uses the same space for the blocks and the list nodes. Since we allocate all memory from the same pool, we will not run out of space until all the memory in the system is exhausted.

A variant of this method can also be used if the allocator does not keep track of allocated blocks. Then the block list only includes free blocks, and so the list headers use the first few words of the free block, space that is not being used anyway. When the block is allocated, it will no longer be free, and so the header is not required. This way, the entire block can be allocated to the requester. As we observed before, this assumes that the memory allocator trusts the requester to release memory correctly. If this is an operating system allocating blocks of memory to processes, the operating system will probably not be willing to trust the processes, and so this method cannot be used. But it is a possible solution with trusted

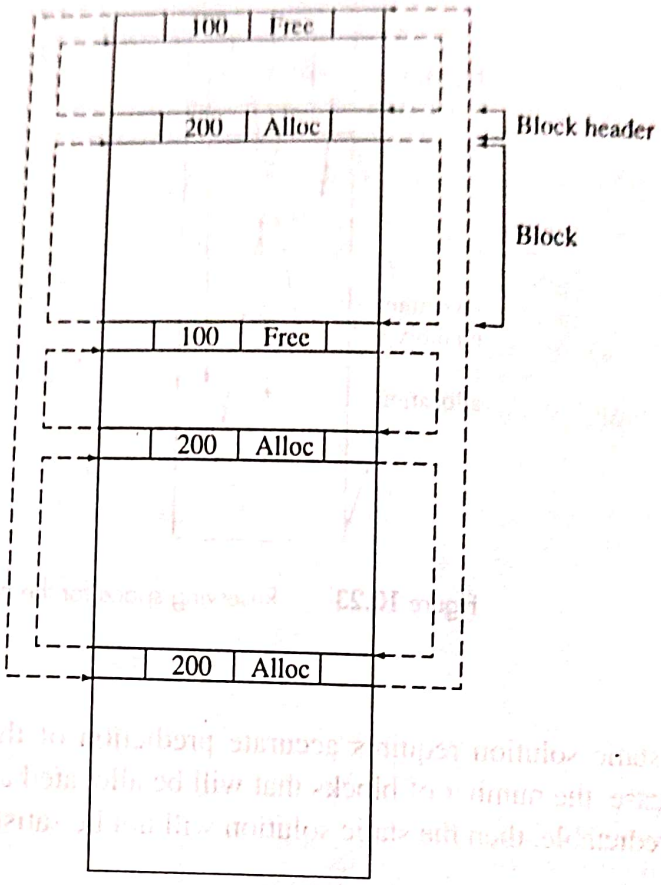


Figure 10.24 Block list with list headers

processes. For example, the other processes might be other parts of the operating system that the memory allocator trusts to work correctly.

10.8.5 The Bitmap Method

In a big restaurant, they need to keep track of which tables are free and which are busy. Often they do this using a diagram of the restaurant showing each table. They mark the diagram to show what tables have people at them. This makes more sense than a list of free tables because it is faster to see if a table is free or busy.

bitmap

The list method is the most obvious way to keep track of free space. The *bitmap* method is a completely different approach to the problem, and is analogous to the restaurant diagram. The bit map is a long string of bits, one for each block of storage. Suppose you divide 4 Mbytes of storage into 128-byte blocks. This makes 32K blocks. A bitmap for this would have 32K bits (4K bytes). If the block is free then the corresponding bit is 1, else it is 0.[8] Figure 10.25 shows an example of a bitmap where each bit tells whether a block of 100 bytes is free or not.

We must allocate space in multiples of the unit of storage we have chosen for the bitmap, 128 bytes in this case. To allocate a block, we round up to the next multiple of 128 and divide by 128 to see how many blocks we need. Then we look through the

[8] Of course, this is just a convention and we could do it the opposite, 1 for allocated and 0 for free.

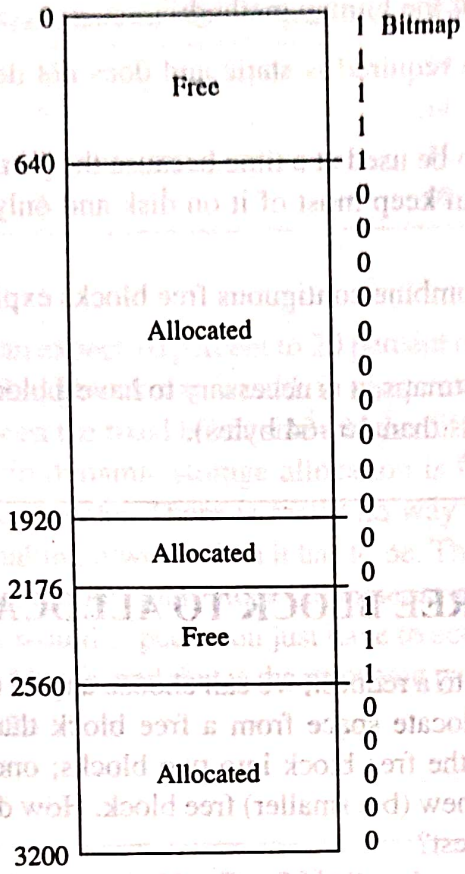


Figure 10.25 Bitmap method for memory allocation

bit map for a string of that many consecutive 1 bits. So, for 10,000 bytes, we need (10,000/128) 79 blocks of 128, or 79 consecutive 1 bits. We record the allocation by setting the bits to 0. Freeing storage is done by setting the bits back to 1.

The bitmap method is not commonly used for main memory allocation, but it is sometimes used to keep track of free space on a disk.

10.8.6 Comparing Methods

Here are the advantages of the list method:

- If there are few blocks, it uses very little space.
- The overhead (one block list node) is the same for all blocks, so this method is more attractive for large blocks.
- The space required for the list is independent of the size of the memory being allocated, and is only related to the number of blocks.
- The block list does not need to be in contiguous storage.
- It is fast to find free blocks.
- Blocks of any size can be allocated, so there is never any space wasted inside a block.

Here are the advantages of the bitmap method:

- The amount of space required is static and does not depend on the number of free blocks.
- Part of the bitmap can be used at a time because the parts do not depend on each other. That is, you can keep most of it on disk and only a portion of it in main memory.
- There is no need to combine contiguous free blocks explicitly. The method does that automatically.
- To avoid very large bitmaps, it is necessary to have a block granularity that is not too small (say, not less than 16–64 bytes).

*10.9 WHICH FREE BLOCK TO ALLOCATE?

When we allocate a block to a request, we can choose any of the free blocks which is big enough. When we allocate space from a free block that is larger than the requested size, we split up the free block into two blocks; one we allocate to the requester, and the other is a new (but smaller) free block. How do we decide which free block to allocate to a request?

The goal of any strategy for allocating free blocks is to minimize fragmentation. As we allocate memory, we accumulate free blocks that are too small to satisfy a request. These are spread between the allocated blocks and called *fragments*, and this process is called *fragmentation* or *checkerboarding*. We would like to allocate blocks to create the least fragmentation. Here are some possible strategies:

fragments
checkerboarding

first fit

next fit

best fit

worst fit

- *first fit*—Choose the first block that is large enough. This avoids going through the entire list and does not favor large or small holes.
- *next fit*—Choose the next block that is large enough. This is the same as first fit, except it starts looking at the point in the free list where the last allocation was done.
- *best fit*—Choose the free block that is closest to the requested size. This creates small holes and preserves large holes.
- *worst fit*—Choose the largest free block. This avoids creating small holes and uses up the large holes.

It is not obvious, a priori, which of these is better, but simulation studies by Shore (1975) and Knuth (1973) show that next fit is a slightly better method than best fit, in that it creates the least fragmentation. Worst fit does considerably worse.

Fragmentation means that memory is wasted, but how much? Knuth derived the "unused memory rule" which states that, if the average allocated block is of size s_0 and the average hole is of size ks_0 (where $0 < k < 1$), then the expected degree of fragmentation is the fraction $\frac{k}{k+2}$. Using this formula, we get the following table relating the ratio of the average free block to the average allocated block and the degree of fragmentation.

Average Free/Allocated Size	Space Lost to Fragmentation
1/2	.20
1/3	.14
1/4	.11
1/5	.09

So, in general, we can expect 10 percent to 20 percent of the storage will be lost to fragmentation. Using a fixed block size would, in most cases, waste even more space due to mismatches between the fixed block size and the different sizes requested.

The main problem in dynamic storage allocation is fragmentation, that is, getting too many small free blocks. There is really no way to avoid this problem, although you can avoid making it worse than it has to be. The problem with the best-fit method is that it produces more small blocks than necessary, and that is why it does not work as well as you would expect. You just have to accept that some storage will be wasted on small free blocks, and that is the price you pay for the advantages of dynamic storage allocation.

10.10 EXAMPLES OF DYNAMIC MEMORY ALLOCATION

The methods we have discussed so far are not used in modern operating systems for two main reasons. The first reason is that they do not solve the exact problem faced by modern operating systems. In Chapter 11, we will see how *paging* works. In a paging system, all of main memory is divided up into pages, which are usually about 4 Kbytes long. Paging is so useful that every modern operating system uses it. In a paging system, memory management is divided into two parts (see Figure 10.26). The first part is the page allocator that allocates physical memory in whole pages. The page allocator has two clients. The first client is the paging system, and the second client is the operating system memory manager. The operating system memory manager only allocates blocks smaller than one page. It allocates space for process descriptors, open file structures, strings, etc.

The operating system memory manager has a simpler job than general dynamic memory allocation since all the blocks it allocates are less than the page size (say 4 Kbytes) and it can always get more memory from the page allocator.

The second reason the operating system does not use one of the algorithms we mentioned so far is that they are too slow. The operating system needs faster memory management that does not require searching lists of unknown length.

A powers-of-two allocator is reasonable in an operating system because there will only be a few sizes to allocate. With 4K pages, we will have lists of blocks of sizes 2K, 1K, 512, 256, 128, 64, and 32 bytes. Each page consists entirely of blocks of one size. If a list is exhausted, then we can allocate a new page for it and divide it

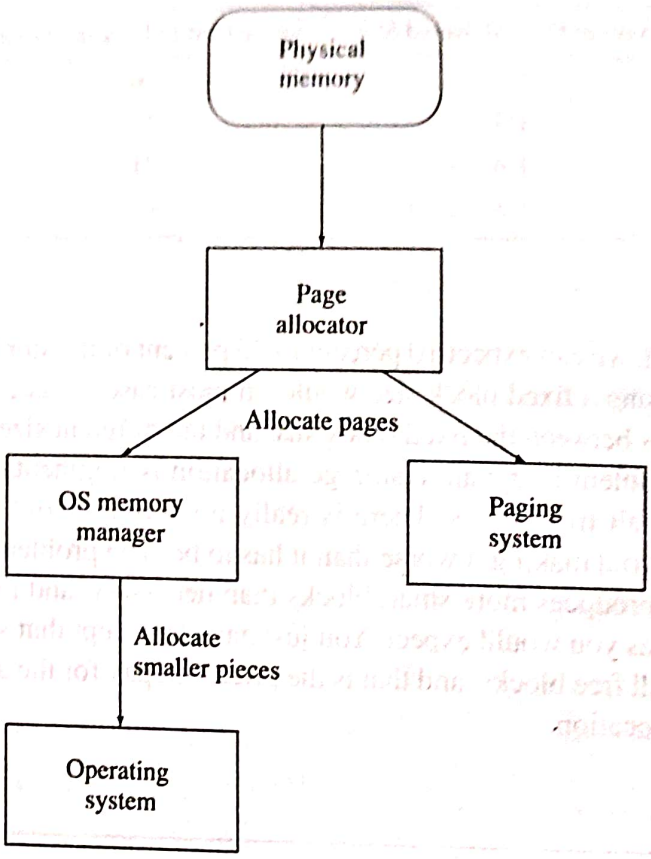


Figure 10.26 Logical and physical address spaces

up into blocks of the required size. If there are lots of free blocks, we can try to find whole pages that are free and remove them from the free list they are on. This method does not require any searching of free lists. McKusick and Karels (1988) describe a powers-of-two memory allocator with these and other improvements.

The buddy system is used in UNIX System V release 4 (Barkley and Lee, 1989). They use a lazy version of the buddy system with bounded delays. Other examples are the zone allocator in Mach and OSF/1 (Sciver and Rashid, 1990) and the slab allocator used in Solaris 2.4 (Bonwick, 1994).

10.11 LOGICAL AND PHYSICAL MEMORY

We discussed the memory-mapping mechanism in Chapter 2, but let's review the terminology again so it is fresh in your mind. A computer system has *physical memory*, which is a hardware device. The physical memory is divided up into small units called *memory cells*, which we will normally assume to be eight-bit bytes. The physical memory cells are named with *physical addresses*, and these physical addresses together form the *physical address space*. The physical addresses are *contiguous*, meaning that each consecutive address refers to a cell. If

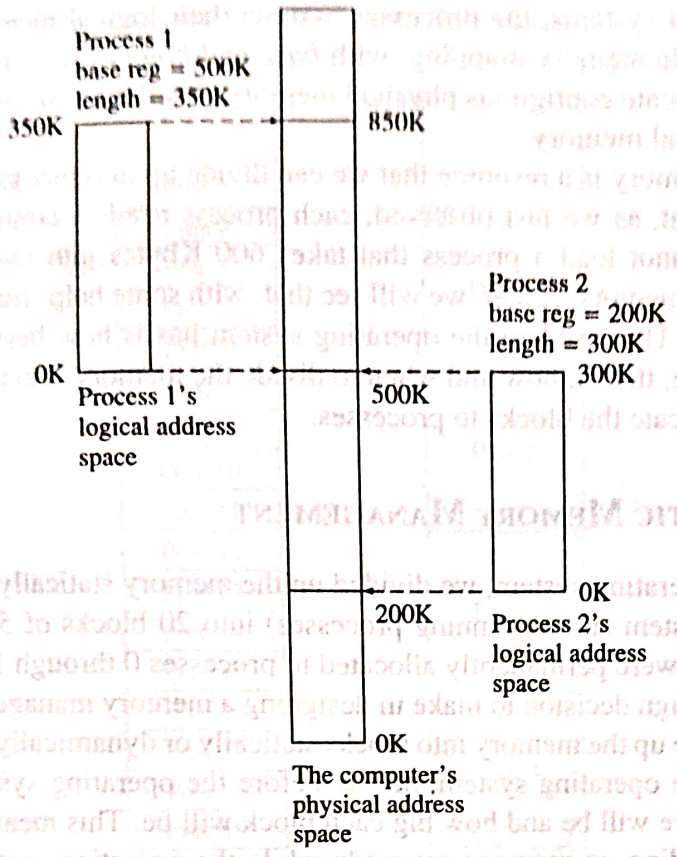


Figure 10.27 Logical and physical address spaces.

there are 4 megabytes (2^{22} bytes) of memory, then the physical addresses will run from 0 to $2^{22} - 1$.

Physical addresses are only used in system mode. In user mode, *logical addresses* are used. When a user mode process accesses a byte using a logical address, the logical address is mapped into a physical address by the hardware. It does this by adding together the logical address, and the contents of the *base* register. So a user-mode program uses logical addresses, and they form the *logical address space*. The logical address space is also contiguous, using consecutive addresses from 0 to some upper limit.

A computer system has only one physical address space, but can have several logical address spaces (see Figure 10.27). The logical address 124 can refer to a number of different memory cells, depending on the current value of the *base* register, but the physical address 124 always refers to the same memory cell.

10.12 ALLOCATING MEMORY TO PROCESSES

Each process requires a certain amount of memory to run. The amount of memory required depends on how big the program and data it uses are. Processes vary widely in how much memory they require, from a few thousand bytes to millions

of bytes. In most systems, the processes assume their logical memory is contiguous. Using simple memory mapping, with *base* and *bound* registers, the operating system must allocate contiguous physical memory in order to provide the user with contiguous logical memory.

Physical memory is a resource that we can divide up into blocks and allocate to the processes, but, as we just observed, each process needs a contiguous block of memory. We cannot load a process that takes 600 Kbytes into two separate 300-Kbyte blocks of memory. (Later, we will see that, with some help from the hardware, we *can* do this.) The problem the operating system has is how best to manage the memory resource, that is, how and when to divide the memory into blocks, and how and when to allocate the blocks to processes.

10.12.1 Static Memory Management

In our simple operating system, we divided up the memory statically (that is, before the operating system started running processes) into 20 blocks of 50 Kbytes each. These 20 blocks were permanently allocated to processes 0 through 19.

The first design decision to make in designing a memory management system is whether to divide up the memory into blocks statically or dynamically. We can, as we did in the simple operating system, decide before the operating system starts how many blocks there will be and how big each block will be. This means that no decisions about dividing up memory are made while the operating system is running. This strategy was sufficient for our simple operating system, where we were concentrating on process implementation and did not want to worry about memory allocation, but it is not a practical solution for a real operating system.

In addition to dividing up the memory statically into fixed-size blocks, we can also, as we can in our simple operating system, allocate each block to a specific process. Actually, we allocated it to a process slot. If the slot is unused, then the memory is unused and wasted. It might be better to only allocate memory to actual processes rather than process table slots.

What we could have done is to divide the physical memory into fewer but larger blocks, and allocate the blocks to processes as they are created. For example, we could have divided our 1 Mbyte of memory into five blocks of 200K each, instead of 20 blocks of 50K. The operating system gets one block, so user processes can use the other four blocks. When a process is created, it is allocated a block of memory. Figure 10.28 shows the original and revised versions.

But what happens after four processes have already been created and we want to create the fifth process? All the blocks of memory have been allocated to processes (remember that one goes to the operating system). What happens is that the fifth process must wait until one of the four existing processes exits and its memory can be reclaimed. Process create requests will be put on waiting lists until memory is freed to satisfy them.

But it is useless to allow for 20 processes when there are only four blocks of memory for processes to run in. A system that statically allocates memory blocks need only have one process slot for each block of memory.

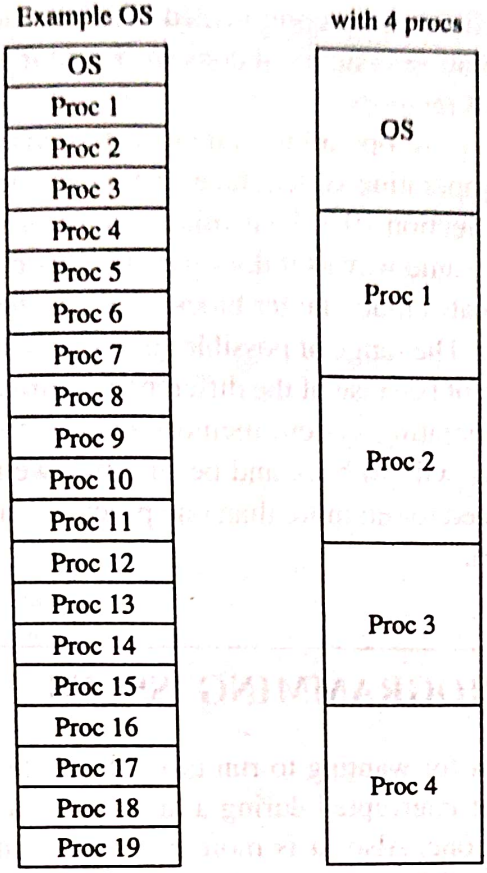


Figure 10.28 Static allocation of larger blocks

In other words, if you statically divide up the memory, there is no difference between static and dynamic allocation of memory to processes. There is a one-to-one correspondence between process table slots and memory blocks. In order to use dynamic allocation, we need to do dynamic division of memory.

10.12.2 Handling Variable-Sized Processes

Now it is time to think about dynamically dividing the memory into blocks. There are several reasons why we want to do that.

In a typical operating system, we do not know how big the processes will be. Some will be small and some will be large. If we statically divide the memory into blocks, we would have to make the memory blocks big enough for the largest process we want to run. But then the memory would be largely wasted when we were running small processes.

An alternative solution would be to statically divide the memory up into different-sized blocks. A small process could run in any block (although we would prefer to give it a small block and save the larger blocks for larger processes), and a large program might fit into only one of the blocks. This makes the queuing of processes

waiting to be created a little more complicated (but not much). This solution was tried in several early operating systems. It does work, but it is inflexible and does not lead to the efficient use of memory.

The solution used in most operating systems is to dynamically divide the memory into blocks. Then the operating system faces the same dynamic memory allocation problem we looked at in Section 10.7. Dynamic memory allocation works in an operating system in about the same way as it does within a process. One difference is that an operating system allocates much larger blocks and gets fewer requests than a per-process memory manager. The range of possible solutions is the same, but the best one to choose might be different because of the different conditions in the design problem.

So the solution to operating system memory management is dynamic memory management, but now we will go back and be sure that we really want to solve the problem. Do we really need to run more than one process at a time? This topic is discussed in the next section.

10.13 MULTIPROGRAMMING ISSUES

There are several reasons for wanting to run more than one program at a time. We have seen that people get interrupted during a task, and often want to suspend one process and run another one. Also, it is more efficient to run several processes at once. For example, we would want to be able to print a document while working on another document. Finally, it is often more convenient to break up a task into several distinct parts, and let one program handle each part. This increases the reusability of programs. The UNIX shell pipeline is an example of this. By packaging various functions into small programs that run in parallel as separate processes, we can use a small set of programs to do a large set of useful functions.

So we need multiprogramming, the ability to run several programs at the same time, and to do this we need to have several programs in memory at the same time. Thus the task we are presented with is how to use the memory we have to the best possible advantage, that is, we want to have as many programs in memory as possible, and always the most useful ones.

But having more than one program in memory at a time presents some ancillary problems that we need to deal with. The two main problems are

- Memory allocation, and
- Memory protection.

With monoprogramming, the operating system could just allocate all the memory to the process that is running, sit back and wait for it to finish, and then reclaim all the memory to give to the next job. With multiprogramming, we have to do dynamic memory allocation, that is, we need to respond to a series of memory requests and be constantly allocating and freeing memory.

Since there is more than one process in memory, we have to be sure that one process does not read or write the memory of another process. Actually, this problem was present in monoprogramming as well, since we had to protect the operating system from the user process.

Multiprogramming improves hardware efficiency and user efficiency. It is worth the trouble.

10.14 MEMORY PROTECTION

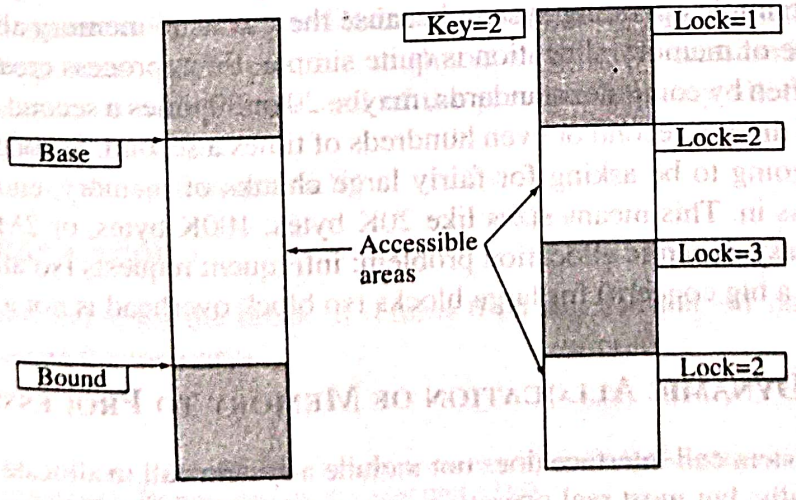
memory protection

The *base* and *bound* registers described in Chapter 2 provide a satisfactory solution to the protection problem. A process cannot access any memory with lower addresses, since the hardware adds the relocation value to every address (using unsigned arithmetic), and it cannot access any memory above its allocation, since the limit register will prevent this.

We should note, however, that *base* and limit registers (or some generalization of them) are not required to implement memory protection. Another approach is to have some idea of the identity of a process and of ownership of parts of memory. For example, suppose each process had an eight-bit process number, and each 4K block of memory had an eight-bit owner register. A process can only access a block of memory if its process number is the same as the owner register of the block. In this case, all processes can address the memory of all other processes, but they cannot actually access the memory since the owner register and the process number will not match.

We might reserve some special values. A process number of 0 can be used by the operating system and represent a "skeleton key" that always matches. An owner number of 0 can represent public memory that any process can access, no matter what its owner number is. The IBM360 used this kind of memory protection scheme. Figure 10.29 shows the two forms of memory protection.

Note that *base* and *bound* registers solve the memory relocation problem as well as the memory protection problem, but lock and key systems only solve the memory



(a) Base and bound registers (b) Lock and key registers

Figure 10.29 Two forms of memory protection

protection problem. On the other hand, lock and key systems allow for noncontiguous memory allocation.

10.15 MEMORY MANAGEMENT SYSTEM CALLS

memory manager

The part of an operating system that manages memory is called the *memory manager.* An important question to ask is, who are the clients of the operating system memory manager, that is, who makes memory requests? We have seen two examples of this in this chapter.

- When a process is created, memory for it is allocated, and the program it will run is loaded into that memory. So the first client of the operating system memory manager is the process creation module of the operating system.
- When a process is running and it runs out of free memory in its allocated memory, it asks for more from the operating system.

10.15.1 STATIC ALLOCATION OF MEMORY TO PROCESSES

In our simple operating system, there were no system calls to change the memory allocation of a process. In that system, the only time memory is requested is when a process is created. A process was created with an initial size, and the size never changes during execution. In a system with no system calls to change the memory allocation of a process dynamically, the create process procedure in the process manager is the only client of the memory manager.

Figure 10.30 shows the situation. The process manager requests a block of memory for the process whenever a new process is created. The request is queued until it can be satisfied. We are assuming that there is more demand for memory than there is memory available, so that the queue is not usually empty and a request usually has to wait for a while before it is satisfied. If this is not the case, then memory management is easy; just give out whatever memory the processes ask for. There was no queue in our simple operating system because there was no memory allocator at all.

This type of memory allocation is quite simple. First, process creation does not occur very often by computer standards, maybe 20 or 30 times a second, certainly not thousands of times a second or even hundreds of times a second. Second, the process manager is going to be asking for fairly large chunks of memory, enough to put a whole process in. This means sizes like 20K bytes, 100K bytes, or 2M bytes. Thus we have an easy dynamic allocation problem: infrequent requests (so allocation efficiency is not a big concern) for large blocks (so block overhead is not a concern).

10.15.2 DYNAMIC ALLOCATION OF MEMORY TO PROCESSES

So far, our system call interface does not include a system call to allocate more memory dynamically, but most real operating systems do allow this. This means that any process can request more memory, and so is a potential client of the memory manager.

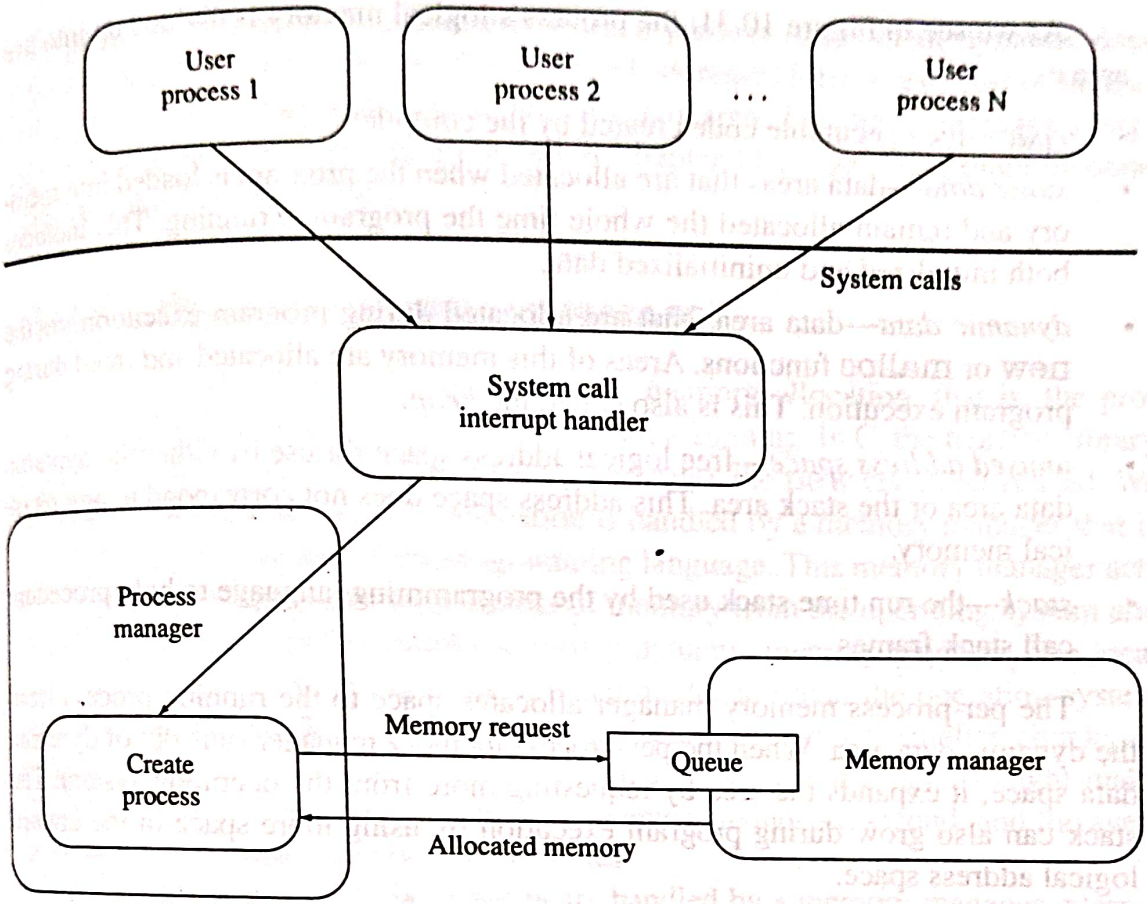


Figure 10.30 Memory requests in the simple operating system

The UNIX memory management system call is a good example to look at, since it has a very simple interface. When a process starts, it gets an initial allocation of memory. The highest memory address is called the "break" (for historical reasons). There is only one memory management system call. It requests that the value of the break be changed, that is, that the amount of memory allocated to the process be changed. This system call can be used to increase or decrease the amount of memory allocated to the process, but, in practice, in almost all cases an increase is requested.[9] Additional memory is added at the end of the previous memory allocation, so the user sees a larger address space.

The format of the system call is:[10]

```
int brk( char * addr );
```

This sets a new value for the break. It returns 0 on success and −1 on failure (if it could not allocate the memory).

[9] In fact, in UNIX, a process is not allowed to reduce its memory allocation below what it was initially allocated. It can, however, ask for more memory and later give it back.

[10] They used brk in UNIX instead of break because break is a reserved word in the C programming language.

As we see in Figure 10.31, the process's logical memory is divided up into five areas:

- *code*—the executable code created by the compiler.
- *static data*—data areas that are allocated when the program is loaded into memory and remain allocated the whole time the program is running. This includes both initialized and uninitialized data.

static data

- *dynamic data*—data areas that are allocated during program execution via the new or malloc functions. Areas of this memory are allocated and freed during program execution. This is also called the *heap*.

dynamic data

heap

- *unused address space*—free logical address space for use by either the dynamic data area or the stack area. This address space does not correspond to any physical memory.
- *stack*—the run time stack used by the programming language to hold procedure call stack frames.

The per-process memory manager allocates space to the running process from the dynamic data area. When the per-process memory manager runs out of dynamic data space, it expands the area by requesting more from the operating system. The stack can also grow during program execution by using more space in the unused logical address space.

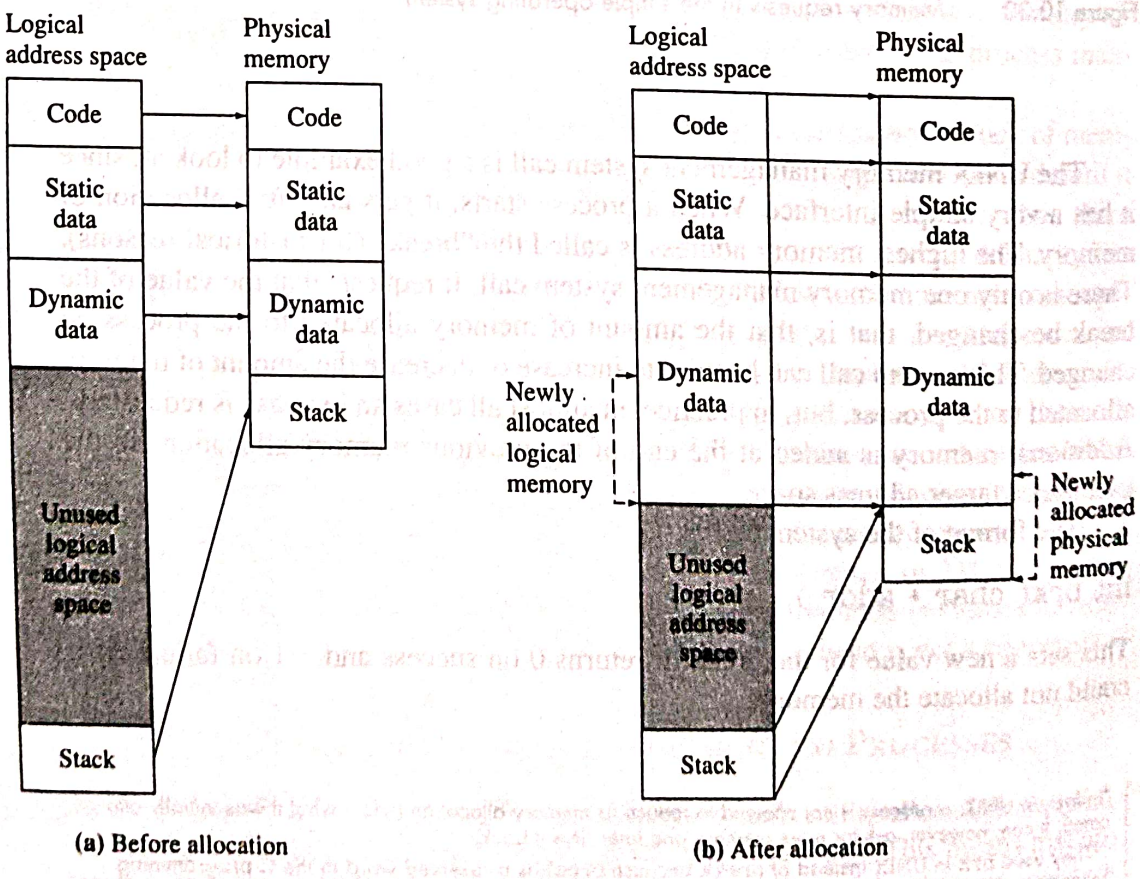


Figure 10.31 Physical memory allocated to a running process

Figure 10.31 shows what happens when a process runs out of dynamic data space and needs to allocate more. These unused addresses form a reservoir of logical addresses to use for the newly added dynamic data area. This figure assumes a paging system (paging systems are discussed in Chapter 11), since this cannot be done with *base* and *bound* registers.

10.15.3 What about NEW and MALLOC?

Most programming languages allow dynamic memory allocation, that is, the programmer can request memory while the program is running. In C, the malloc library procedure is used, and in C++(and Pascal and Ada), the new construct is used. We have seen that dynamic memory allocation is handled by a memory manager that is part of the runtime control of the programming language. This memory manager acts as a middleman since it gets large blocks of memory from the operating system and then doles it out in smaller chunks to satisfy dynamic memory requests. The local memory manager in the process has a much harder job than the operating system memory manager since it gets many more requests for much smaller chunks of memory so the management problems are greater. An active program might make thousands or tens of thousands of dynamic memory requests a second, and the average request size might be only 10 or 20 bytes.

It is appropriate that these requests are handled by a memory manager internal to the process, since you would not want to pay the overhead cost of a system call for so many small memory requests.

10.15.4 Freeing Memory at Each Level

As we have seen, the operating system manages its memory, allocates it to processes, and, later, processes free the memory. When a process exits, all the memory it has been allocated is freed, but a process can also free memory while it is still running if it no longer needs the memory.

Within each process, there is another memory manager, the new/malloc memory manager, that manages the dynamic data area for the process. Figure 10.32 shows the two levels of memory managers.

The per-process memory manager only manages the dynamic data area. The programming language runtime procedures manage the stack. The code and static data areas do not change size as the program executes. For the rest of this section, we will refer to the per-process memory manager as the *malloc* memory manager, or just *malloc*.

The two levels of memory management do interact. When *malloc* runs out of memory, it asks the operating system for more. But what happens when you free memory from a program? This frees the memory to *malloc*, but not to the operating system. If an area of memory at the end of *malloc's* area all becomes free, malloc *could* decide to free that memory and return it to the operating system. Figure 10.33 shows a situation where there is a large free area at the end of the dynamic memory area.

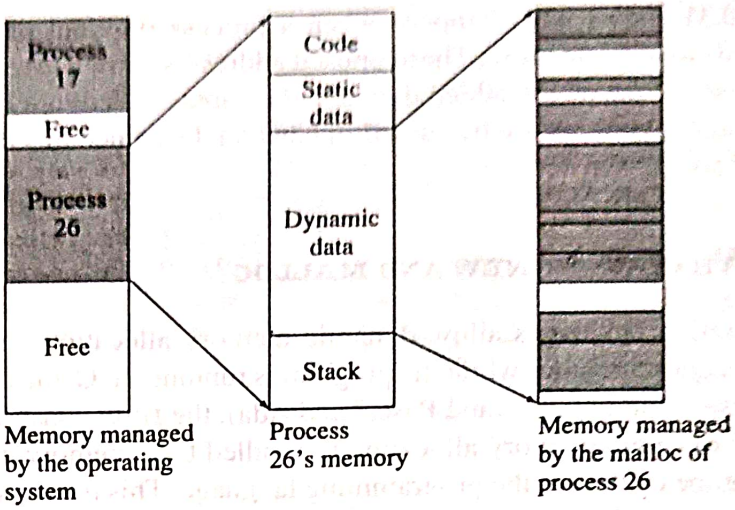


Figure 10.32 Two levels of memory management

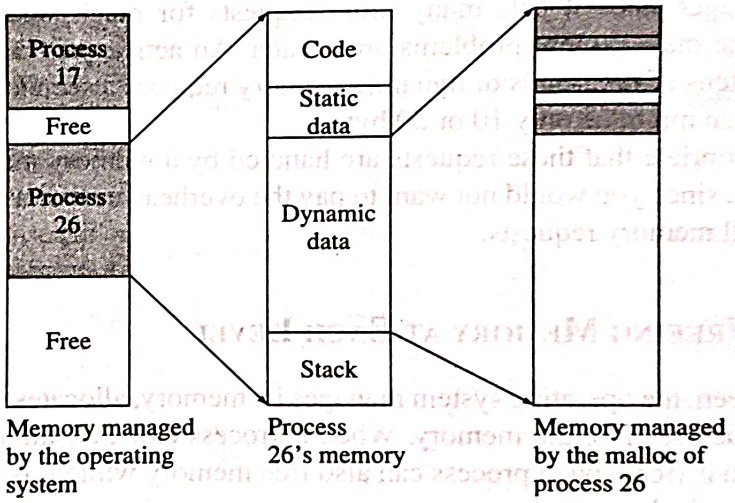


Figure 10.33 Free memory at malloc level, but not at OS level

In practice, however, very few, if any, per-process memory managers do this. The main problem is that the situation happens so seldom that it is not worth it to check for it. The free memory has to be at the end of the dynamic data area, and it has to be completely free. A single, small block near the end of the dynamic data area that is not free will prevent the entire area from being freed. So programs generally grow larger, but never grow smaller.

This effect is a consequence of our two levels of memory management. All design choices have advantages and disadvantages.

Remember, it is not that *malloc* cannot free memory, it is just that it probably will not. If you have a very long-running program that tends to allocate a lot of memory and then free it, you might want to have a custom memory manager for it that will free the memory back to the operating system when it is no longer being used. An example of a program like this would be an X server. If you load some fonts and

bitmaps, they will take up memory. If you later unload them, the memory will not be returned to the operating system. The memory will be reused, of course, if you load some other fonts or bitmaps.

10.15.5 A Different Memory Management System Call

The UNIX *brk* system call was developed in a time when memory was always allocated in large contiguous blocks. In Chapter 11, we will see that modern memory management can easily deal with logical address spaces that are not contiguous. This allows us to use a more obvious system call for memory allocation. We can present the system call here, but you will have to wait until Chapter 11 to see how it could be implemented.

The format of our new memory allocation system call is:

```
char * AllocateMemory( int length );
```

This call allocates *length* bytes and returns the starting address of the the new memory. If the memory cannot be allocated, it returns an address of 0. This memory might not be contiguous with any other memory you have been allocated, so each block of memory you get is independent of all the other blocks.

This is certainly a more straightforward way to allocate memory. There is no need to move the stack or readjust any existing memory. Modern operating systems are more likely to have a memory allocation call like this.

*10.16 EXAMPLE CODE FOR MEMORY ALLOCATION

In this section, we will show some example code to explain in more detail how memory allocation in an operating system would be handled.

First, consider the data type and data object declarations. We need a queue (RequestList) of memory requests (struct MemoryRequest), which we will keep as a doubly linked list. It is initialized as an empty list. We also need a doubly linked list (BlockList) of blocks (struct Block).

The memory allocator is initialized (Initialize) by giving it a large chunk of memory to allocate. The block list starts out with one free block on it, which is the entire of chunk of memory that is being managed. Later requests will divide it up into smaller blocks.

MEMORY ALLOCATOR DATA

```
// The structure for memory requests
struct MemoryRequest {
   int size;              // in bytes
   Semaphore satisfied;   // signal when memory is allocated
   char ** startAddressSlot; // return block address to the caller here
   MemoryRequest *next, *prev; // doubly linked list
};

// The memory request list
```

```
// keep a front and back pointer for queue
MemoryRequest * RequestListFront, *RequestListBack;

// The structure for block list nodes
struct Block {
   int size;          // in bytes
   int isFree;        // free or allocated block
   char * start;      // where the block starts
   Block *next, *prev; // doubly linked list;
};

// The block list
Block * BlockList;

// The initialization procedure needs to be called before
// any requests are processed.
void Initialize( char * start, int size ) {
    RequestListFront = 0;
    BlockList = new Block(size, True, start, 0, 0);
}
```

The RequestABlock procedure is called to request a block of storage to be allocated. Along with the size of the request, the requester passes a semaphore for the memory allocator to signal when the request has been satisfied, and the address of a place to put the address of the allocated block.

MAKE AN ALLOCATION REQUEST

```
// The request procedure: request a block to be allocated
void RequestABlock( int size, Semaphore * satisfied,
     char ** startAddressSlot) {
     MemoryRequest * n = new MemoryRequest( size, satisfied,
     startAddressSlot, 0 , 0);
    if( RequestListFront == 0 ){ // list was empty
     RequestListFront = RequestListBack = n;
    } else {
     RequestListBack-> next = n;
     RequestListBack = n;
    }
    TryAllocating();
}
```

The TryAllocating procedure is called whenever something has changed that makes us believe a request might be able to be satisfied. It is called each time a new request comes in and each time a block of memory is freed. It looks through all the pending requests and checks if any of them can be allocated. If the request is allocated, then it is removed from the request queue.

TRY TO ALLOCATE TO A REQUEST

```
// The allocation procedure
void TryAllocating( void ) {
    MemoryRequest * request = RequestListFront;
    // look through the list of requests and satisfy any ones you can
    while( request != 0 ) {
        // can we allocate this one?
        if( CanAllocate( request ) { // yes we can
            // remove from the request list
            if( RequestListFront == RequestListBack ) {
                // it was the only request on the list
                // of the request list is now empty
                RequestListFront = 0;
                break; // no more requests
            } else {
                // unlink it from the list
                request->prev->next = request->next;
                request->next->prev = request->prev;
                MemoryRequest * oldreq = request; // save the address
                // get the link before we delete the node
                request = request->next;
                delete oldreq;
            }
        } else {
            request = request-> next;
        }
    }
}
```

The CanAllocate procedure checks to see if a specific request can be satisfied. It does this by looking through the block list for the first free block that is big enough. This is the *first-fit* algorithm.

If it finds a free block, it splits the free block up into an allocated block (to return to the requester) and a free block, which consists of the rest of the block that was not allocated. It passes back the location of the allocated block, and signals the semaphore to inform the requester that the request has been satisfied.

TRY TO ALLOCATE A BLOCK

```
// See if we allocate one request
int CanAllocate( MemoryRequest * request ) {
    int size = request->size;
    Block * p = BlockList;
    // go through the list of blocks
    while( p != 0 ) {
        if( p->size >= size ) {
            // this block is big enough to use. see what is left over
```

```
            int extra = p->size - size;
            if( extra != 0 ) {
                // split the block into two blocks
                Block * np = new Block;
                np->size = extra;
                np->isFree = True;
                np->start = p->start + size;
                np->prev = p;
                np->next = p->next;
                p->next->prev = np;
                p->next = np;
            }
            p->isFree = False;
            *(request->start) = p->start;
            SignalSemaphore( request->satisfied );
            return True;
        }
        p = p->next;
    }
    return False;
}
```

The FreeBlock procedure is called to free a block of memory after the requester has finished using it. It goes through the block list looking for the block (based on its start address). We merge free blocks with adjacent free blocks on either side, if possible.

FREE AN ALLOCATED BLOCK

```
// Free a block of memory
void FreeBlock( char * start ){
    Block * p = BlockList;
    // go through the list of blocks to find this one
    while( p != 0 ) {
        if( p->start == start ) {
            p->isFree = True;
            // merge with the next block( if it is free )
            Block * nextp = p->next;
            if( nextp != 0 && nextp->isFree ) {
                p->size += nextp->size;
                p->next = nextp->next;
                nextp->next->prev = p;
                delete nextp;
            }
            // merge with the previous block( if it is free )
            Block * prevp = p->prev;
            if( prevp != 0 && prevp->isFree ) {
                prevp->size += p->size;
                prevp->next = p->next;
```

```
            p->next->prev = prevp;
            delete p;
        }

        return;
    }
    p = p->next;
  }
  // ERROR: returned block not found
}
```

10.17 SUMMARY

There are various ways to manage the memory resource in an operating system. Most processes do their own low-level memory management, and only rely on the operating system for allocating large chunks of memory. The study of memory management can be divided into two parts: the management of memory within a single process, and the management of memory between processes in an operating system.

A compiler generates an object module as the result of a compilation. Each object module is divided into a header, a machine code section, an initialized data section, an uninitialized data section, a symbol table, and a relocation section. Useful object modules are collected into libraries. A linker combines object modules and libraries into a load module that is ready to be loaded into the memory of a process. When a process is initiated, the run-time loader initializes the process's memory from the load module. The loading of library modules can be delayed until the process is executed (this is called load time dynamic linking). This saves disk space in storing the load module. The loading of library modules can be delayed until they are first used by the running process (this is called dynamic linking). This saves startup time and can sometimes avoid unnecessary loading.

Each process manages its own memory, and so must solve the general memory allocation problem. Static allocation of memory and static division of memory into fixed-size blocks does not work very well in general. The best solution is dynamic memory allocation. The basic algorithm is simple, but there are a number of design decisions to be made. You can keep a free list (the most common method) or use a bit map. You can keep the free list in a separate area or with the rest of the dynamically allocated memory. All dynamic memory allocation leads to fragmentation despite our best efforts to avoid it.

The dynamic allocation of more memory to running processes is important so that processes can efficiently use memory, but it makes memory management a little bit harder. The two levels of memory management (within a process and between processes) do not interact well, and, in particular, it is hard to free memory once it has been allocated at both levels.

10.17.1 TERMINOLOGY

After reading this chapter, you should be familiar with the following terms:

- best fit
- bitmap
- block list
- buddy system
- checkerboarding
- defined external symbol
- dynamic data
- dynamic memory allocation
- dynamic relocation
- external symbol
- first fit
- fragment
- fragmentation
- header section
- heap
- initialized data section
- library
- linker
- linking
- load module
- load time dynamic linking
- machine code section
- memory manager
- memory protection
- next fit
- object module
- powers-of-two allocation
- run time dynamic linking
- static data
- static relocation
- symbol table
- undefined external symbol
- uninitialized data section
- worst fit

10.17.2 Review Questions

The following questions are answered in the text of this chapter:

1. Why are there two levels of memory management?
2. What are the two different kinds of external symbols and how do they differ?
3. Why do we distinguish between initialized and uninitialized data in an object module?
4. What is the purpose of the header section in an object module?
5. Describe the linking algorithm.
6. Describe the differences between linking and relocation.
7. Describe the format of a load module.
8. Describe the process of loading a load module for execution.
9. Give the relative advantages and disadvantages of load time dynamic linking and run time dynamic linking.
10. What is the dynamic memory allocation design problem?
11. What is the difference between powers-of-two allocation and the buddy system?
12. Give the advantages and disadvantages of keeping allocated blocks on the block list.
13. Compare the block list and bitmap methods of keeping track of free blocks.
14. What are the differences between first fit, next fit, best fit, and worst fit?
15. What is the difference between a logical address space and a physical address space?
16. Why is it better to allocate memory to processes dynamically?
17. Why is memory protection important in a multiprogramming system?
18. Compare the `brk` and `AllocateMemory` system calls described in the chapter.

10.17.3 Further Reading

See Korn and Vo (1985) for more information on new and more efficient `mallocs`. See Maccabe (1993, Ch. 10) and Stallings (1992, Appendix 5A) for more information on linking, loading, and object module formats. Knuth (1973) gives a derivation of the 50-percent rule and has a good discussion of dynamic memory allocation techniques. See Peterson and Norman (1977) for a complete discussion of buddy systems. More information about dynamic memory allocation can be found in Shore (1975, 1977), Bays (1977), and Beck (1977). Stephenson (1983) gives an improved version of first fit.

10.18 PROBLEMS

1. Give two reasons why every `new` or `malloc` request is not handled by a system call to the operating system instead of by a per-process memory manager.
2. Linking requires that we remember the places that need linking and do the linking once the symbol is defined. Describe the data structures that would be used to keep track of the information during the linking process and to make the links when the symbol is defined.
3. Consider the algorithm we gave for creating a load module. Modify this algorithm to reflect the fact that the code sections of all the object modules need to be loaded together, as do the initialized and uninitialized data sections.
4. Why do we want to keep the three sections (code, initialized data, and uninitialized data) separate in a load module?
5. Some systems guarantee that "uninitialized" data will be initialized to some value (usually zero). UNIX system do this. By "uninitialized," we then mean not initialized by the programmer. Other systems do not guarantee any initialization of this storage. Give arguments for and against both of these design choices.
6. Describe the object module format on a machine you use. Compile the C++ program in Figure 10.4 and look at the object module it produces. How does it differ from the example given in Figure 10.5?
7. A runtime loader usually reads the header of the load modules first, then it reads the rest of the load modules. Why does it do this? What does it do between the reading of the header and the reading of the rest of the load module?
8. What information must be kept about a load module after it is loaded if some parts of the load module will be dynamically loaded?
9. A linker must keep track of all references it sees to an external variable it sees until it sees the definition of the external variable. Then it must fill in the correct value for all the references. This involves going back to parts of the load module that have already been processed and patching them. The loader could do this in two ways: (1) it could thread a linked list through the places in the load module where the references themselves go; or (2) it could keep a linked list in its own memory of the location of all the references it needs to patch. Compare these two methods. What are the advantages and disadvantages of each one?
10. A linker will read library files and find modules that define external references that the linker has not found yet. Design a data layout for a library file that will facilitate this activity. Remember that a library is basically a collection of object modules. What tables and indexes, if any, will it have? Will these tables be at the beginning or end of the library file? How will they be sorted?
11. What is the purpose of the relocation information in the object module format? How might this information differ from architecture to architecture?

12. Often, different operating systems will run on the same hardware architecture. As an example, both UNIX and MS/DOS run on Intel 80x86 machines. Suppose we wanted to design a common object file format that would work for both operating systems. The idea is that a module compiled on one operating system could be run on the other. What are some of the problems you would encounter in doing this?

13. Suppose you keep a block list with both free and allocated blocks on it. What error checking would you do when a block is freed? Now suppose we only keep a record of free blocks. What error checking is possible when a block is freed?

14. Suppose we use a dynamic memory management scheme that requires 8 bytes per block in space overhead and requires 50 microseconds to respond to a request for memory. Suppose that the average request size is 256 Kbytes, and you get an average of 30 requests a second.

a. What is the average space and time overhead for dynamic memory management?

b. Now compute the average space and time overhead for dynamic memory management in a system where the average request is 40 bytes and you get an average of 2,000 memory requests a second.

c. The first figures are more typical of the memory management demands on an operating system and the second are more typical of the memory management demands of a program doing a lot of dynamic memory allocation. What does this say about how each one should be designed?

15. Describe a dynamic memory management situation where you think the buddy system would perform very well, and one where you think it will perform very poorly.

16. Suppose you have 16 Mbytes of main memory. Using the list method, you have an overhead of eight bytes per memory block. Using the bitmap method, you use an allocation granularity of 128 bytes. How many blocks are there when the space overhead of both methods is the same? What is the average block size for this many blocks?

17. Suppose you were allocating 16 Mbytes (2^{24} bytes) of memory, and will use either bitmap allocation or list allocation. For the bitmap, you will use allocation units of 128 bytes (2^7 bytes, so there will be 2^{17} allocatable units). For the list system, you will thread the list through a block header at the beginning of each block (free or allocated). This header will take up 8 (2^3) bytes.

Consider the situation where the average block allocated is 256 (2^8) bytes (some are larger and some are smaller, but the average is 256 bytes). Which method will use more storage for allocation system overhead, bitmap or list?

Now consider the situation where the average block allocated is 64K bytes (2^{16} bytes). Which method will use more storage for allocation system overhead, bitmap or list?

Show some figures (and explain them) to support each answer.

18. Write a program that allocates a block of memory dynamically every few seconds. Run it, and while it is running use an operating systems utility to look at how much memory it is using. Do this for various sizes of blocks. Report your results.

Hint: In C++, use `new` and `delete`, and in C, use `malloc` to allocate, say, 20,000 bytes at a time.

19. Write a program that allocates a lot of space, then deallocates it. Does the memory it uses go up and then down, or does it stay up? *Hint:* In C++, use `new` and `delete`, and in C, use `malloc` and `free`.

20. Give the advantages and disadvantages in using a doubly linked list for the free list.

21. Write a C++ subroutine that finds a block of *N* units in a memory allocation bitmap. Write a C++ subroutine that finds a block of *N* units of memory in a system that uses a free list and first fit. Compare the two procedures.

22. Explain why it is easy to dynamically allocate blocks of memory if all the blocks are the same size. Why don't we have to use a dynamic memory allocation algorithm in this case?

23. Describe a kind of operating system where it would be reasonable to assume that all processes were about the same size.

24. Why is the space allocated to a stack automatically increased when the stack overflows it? Why is there a limit to how much it can be increased altogether?

25. Describe an application that would be much easier to implement on a computer that used the lock/key form of memory protection than it would on a computer that used the *base/bound* register form of memory protection.

26. Design an extension of the lock/key method of memory protection that allows different protection for reading, writing, and executing in an area of memory. Can you think of a similar extension for the *base/bound* method of memory protection? Can it solve part of the problem, if not all of it?

27. Many modern OSs will dynamically load parts of the operating system. Give one example of a part of the operating system that would be a very good candidate for dynamic linking, and give one example of a part of the operating system that you would not want to dynamically load.

28. Consider the problem of dynamically loading operating system modules. What problems would you encounter? What would be the benefits of doing this?

29. Suppose you wanted to implement a per-process memory manager that returned excess unused space to the operating system. Say that, as soon as 75 percent of the dynamic memory was free, it would halve the size of the memory pool. What techniques would you use to accomplish this goal? How hard would it be? What kind of overhead would you incur?

30. Modify the code for memory allocation in Section 10.16 to use block headers at the beginning of each block rather than a separate list of block structures like it uses now.

31. Modify the code for memory allocation in Section 10.16 to use a next-fit algorithm rather than the first-fit algorithm it uses now.

32. Modify the code for memory allocation in Section 10.16 to use a best-fit algorithm rather than the first-fit algorithm it uses now.