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.

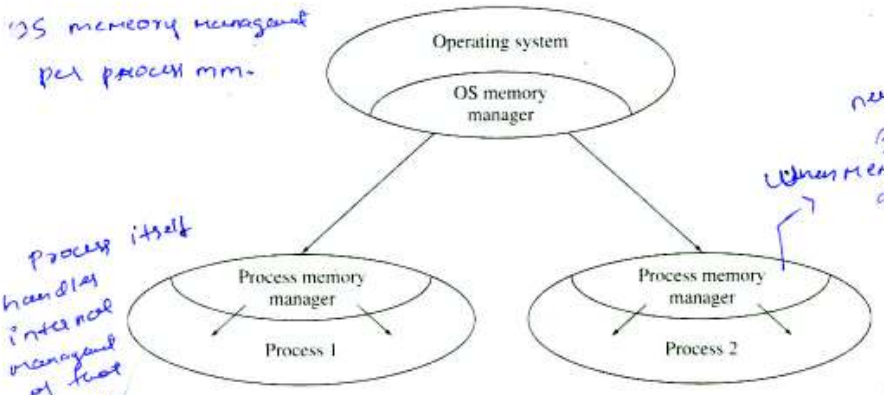

Figure 10.1 Two levels of memory management

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

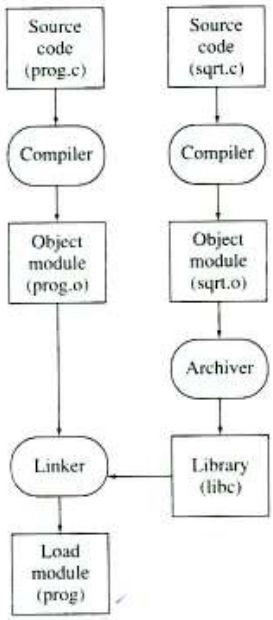

Figure 10.2 Creating a load module

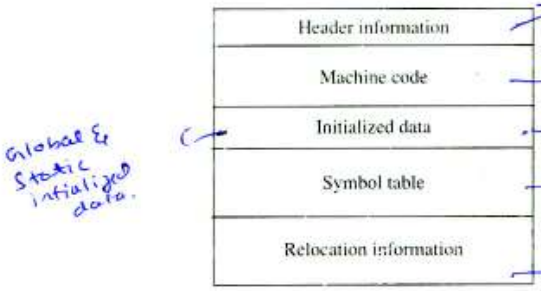

Figure 10.3 Object module format

```
#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

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.

machine code 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.

initialized 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

uninitialized data section

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-

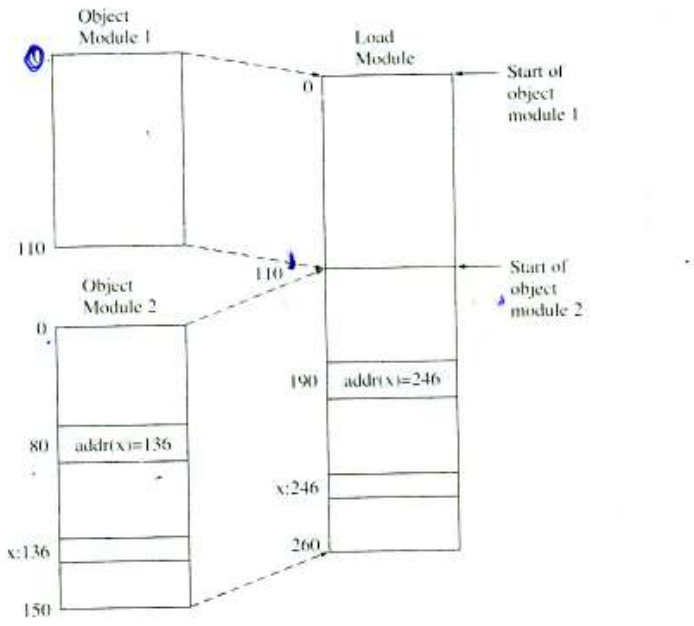

Figure 10.6 Relocating an object module into a load module

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

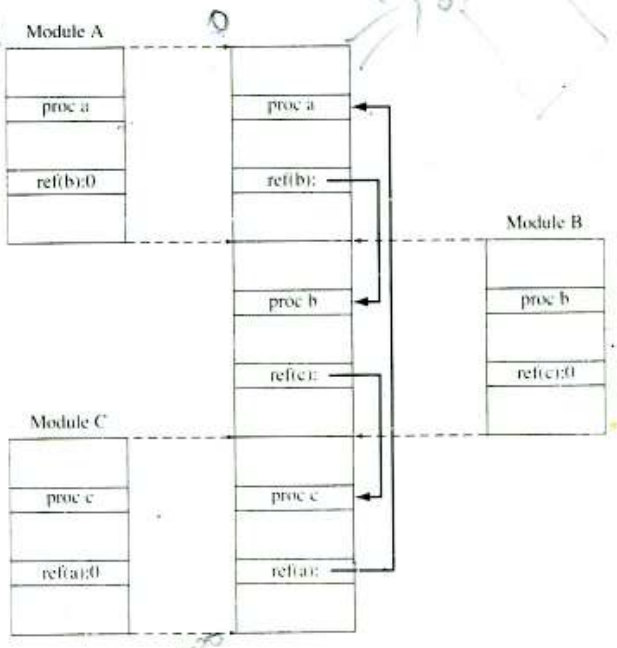

Figure 10.7 Linking object modules in a load module

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.

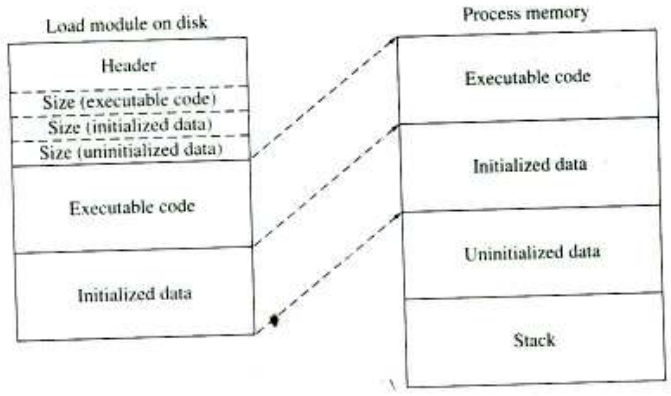

Figure 10.8 Loading a program into a process

[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."

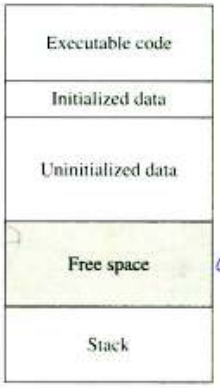

Figure 10.9 Memory areas of a running process

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

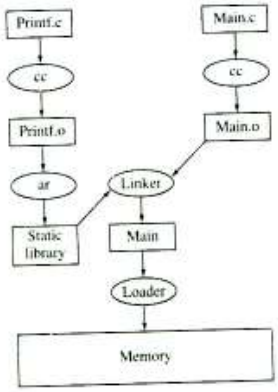

Figure 10.10 Normal linking and loading.

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.

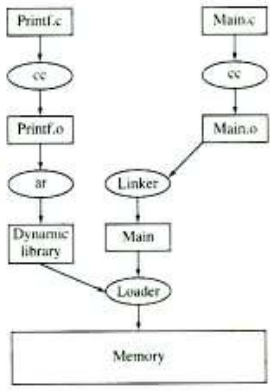

Figure 10.11 Load time dynamic linking

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.

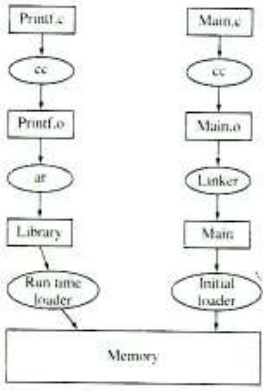

Figure 10.12 Dynamic linking of modules

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

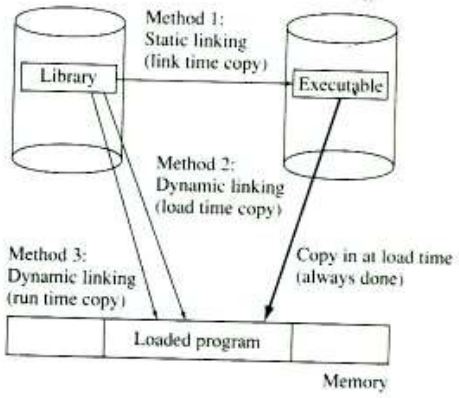

Figure 10.13 Static and dynamic linking

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.