APPENDIX 7A LOADING AND LINKING

The first step in the creation of an active process is to load a program into main memory and create a process image (Figure 7.13). Figure 7.14 depicts a scenario typical for most systems. The application consists of a number of compiled or assembled modules in object-code form. These are linked to resolve any references between modules. At the same time, references to library routines are resolved. The library routines themselves may be incorporated into the program or referenced as shared code that must be supplied by the operating system at run time. In this appendix, we summarize the key features of linkers and loaders. For clarity in the presentation, we begin with a description of the loading task when a single program module is involved; no linking is required.

Loading

In Figure 7.14, the loader places the load module in main memory starting at location $x$. In loading the program, the addressing requirement illustrated in Figure 7.1 must be satisfied. In general, three approaches can be taken:
Absolute Loading

An absolute loader requires that a given load module always be loaded into the same location in main memory. Thus, in the load module presented to the loader, all address references must be to specific, or absolute, main memory addresses. For example, if x in Figure 7.14 is location 1024, then the first word in a load module destined for that region of memory has address 1024.

The assignment of specific address values to memory references within a program can be done either by the programmer or at compile or assembly time (Table 7.2a). There are several disadvantages to the former approach. First, every programmer would have to know the intended assignment strategy for placing modules into main memory. Second, if any modifications are made to the program that involve insertions or deletions in the body of the module, then all of the addresses will have to be altered. Accordingly, it is preferable to allow memory references within programs to be expressed symbolically and then resolve those symbolic references at the time of compilation or assembly. This is illustrated in Figure 7.15. Every reference to an instruction or item of data is initially represented by a symbol. In preparing the module for input to an absolute loader, the assembler or compiler will convert all of these references to specific addresses (in this example, for a module to be loaded starting at location 1024), as shown in Figure 7.15c.

Relocatable Loading

The disadvantage of binding memory references to specific addresses prior to loading is that the resulting load module can only be placed in one region of main memory. However, when many programs share main memory, it may not be desirable to decide ahead of time into which region of memory a particular module should be loaded. It is better to make that decision at load time. Thus we need a load module that can be located anywhere in main memory.

To satisfy this new requirement, the assembler or compiler produces not actual main memory addresses (absolute addresses) but addresses that are relative to some known point, such as the start of the program. This technique is illustrated in Figure 7.15c. The start of the load module is assigned the relative address 0, and all other memory references within the module are expressed relative to the beginning of the module.

With all memory references expressed in relative format, it becomes a simple task for the loader to place the module in the desired location. If the module is to be loaded beginning at location x, then the loader must simply add x to each memory reference as it loads the module into memory. To assist in this task, the load module must include information that tells the loader where the address references are and how they are to be interpreted (usually relative to the program origin, but also possibly relative to some other point in the program, such as the current location). This set of information is prepared by the compiler or assembler and is usually referred to as the relocation dictionary.

Figure 7.14 A Loading Scenario
Dynamic Run-Time Loading

Relocatable loaders are common and provide obvious benefits relative to absolute loaders. However, in a multiprogramming environment, even one that does not depend on virtual memory, the relocatable loading scheme is inadequate. We have referred to the need to swap process images in and out of main memory to maximize the utilization of the processor. To maximize main memory utilization, we would like to be able to swap the process image back into different locations at different times. Thus, a program, once loaded, may be swapped out to disk and then swapped back in at a different location. This would be impossible if memory references had been bound to absolute addresses at the initial load time.

The alternative is to defer the calculation of an absolute address until it is actually needed at run time. For this purpose, the load module is loaded into main memory with all memory references in relative form (Figure 7.15c). It is not until an instruction is actually executed that the absolute address is calculated. To assure that this function does not degrade performance, it must be done by special processor hardware rather than software. This hardware is described in Section 7.2.

Dynamic address calculation provides complete flexibility. A program can be loaded into any region of main memory. Subsequently, the execution of the program can be interrupted and the program can be swapped out of main memory, to be later swapped back in at a different location.

Linking

The function of a linker is to take as input a collection of object modules and produce a load module, consisting of an integrated set of program and data modules, to be passed to the loader. In each object module, there may be address references to locations in other modules. Each such reference can only be expressed symbolically in an unlinked object module. The linker creates a single load module that is

<table>
<thead>
<tr>
<th>Table 7.2 Address Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
</tr>
<tr>
<td>Programming time</td>
</tr>
<tr>
<td>Compile or assembly time</td>
</tr>
<tr>
<td>Load time</td>
</tr>
<tr>
<td>Run time</td>
</tr>
</tbody>
</table>

**Legend**

(a) Loader

(b) Linker

Figure 7.15 Absolute and Relocatable Load Modules

APPENDIX 7A / LOADING AND LINKING 329
the contiguous joining of all of the object modules. Each intramodule reference must be changed from a symbolic address to a reference to a location within the overall load module. For example, module A in Figure 7.16a contains a procedure invocation of module B. When these modules are combined in the load module, this symbolic reference to module B is changed to a specific reference to the location of the entry point of B within the load module.

**Linkage Editor**

The nature of this address linkage will depend on the type of load module to be created and when the linkage occurs (Table 7.2b). If, as is usually the case, a relocatable load module is desired, then linkage is usually done in the following fashion. Each compiled or assembled object module is created with references relative to the beginning of the object module. All of these modules are put together into a single relocatable load module with all references relative to the origin of the load module. This module can be used as input for relocatable loading or dynamic runtime loading.

![Diagram of Linkage](image)

**Figure 7.16** The Linking Function

A linker that produces a relocatable load module is often referred to as a linkage editor. Figure 7.16 illustrates the linkage editor function.

**Dynamic Linker**

As with loading, it is possible to defer some linkage functions. The term *dynamic linking* is used to refer to the practice of deferring the linkage of some external modules until after the load module has been created. Thus, the load module contains unresolved references to other programs. These references can be resolved either at load time or run time.

For load-time dynamic linking, the following steps occur. The load module (application module) to be loaded is read into memory. Any reference to an external module (target module) causes the loader to find the target module, load it, and alter the reference to a relative address in memory from the beginning of the application module. There are several advantages to this approach over what might be called static loading:

- It becomes easier to incorporate changed or upgraded versions of the target module, which may be an operating system utility or some other general-purpose routine. With static linking, a change to such a supporting module would require the relinking of the entire application module. Not only is this inefficient, but it may be impossible in some circumstances. For example, in the personal computer field, most commercial software is released in load module form; source and object versions are not released.

- Having target code in a dynamic link file paves the way for automatic code sharing. The operating system can recognize that more than one application is using the same target code because it loaded and linked that code. It can use that information to load a single copy of the target code and link it to both applications, rather than having to load one copy for each application.

- It becomes easier for independent software developers to extend the functionality of a widely used operating system such as Linux. A developer can come up with a new function that may be useful to a variety of applications and package it as a dynamic link module.

With run-time dynamic linking, some of the linking is postponed until execution time. External references to target modules remain in the loaded program. When a call is made to the absent module, the operating system locates the module, loads it, and links it to the calling module.

We have seen that dynamic loading allows an entire load module to be moved around; however, the structure of the module is static, being unchanged throughout the execution of the process and from one execution to the next. However, in some cases, it is not possible to determine prior to execution which object modules will be required. This situation is typified by transaction-processing applications, such as an airline reservation system or a banking application. The nature of the transaction dictates which program modules are required, and they are loaded as appropriate and linked with the main program. The advantage of the use of such a
dynamic linker is that it is not necessary to allocate memory for program units unless those units are referenced. This capability is used in support of segmentation systems.

One additional refinement is possible: An application need not know the names of all the modules or entry points that may be called. For example, a charting program may be written to work with a variety of plotters, each of which is driven by a different driver package. The application can learn the name of the plotter that is currently installed on the system from another process or by looking it up in a configuration file. This allows the user of the application to install a new plotter that did not exist at the time the application was written.