

7 Basic Memory Management

Memory management in ST systems focuses on how the application program, including the system functions that have been linked into the program, are prepared for execution, how the memory load map is constructed, and then the program is loaded into the computer’s memory. With the exception of some new ideas that accompany several object-oriented systems like Java and C#, the traditional model for preparing a program for execution is very old technology. We will take a look at the conventional as well as this modern alternative in this chapter.

The coroutine model depends on the executable memory being divided into regions, some of which will be used to store information that the system functions need to support the coroutine execution model, and others of which are used to store the program and its data during execution – see [Section 4.2.1](#) to refresh your memory on the executable memory organization. However the memory allocation mechanisms are elementary compared to the memory management strategies used in MT and MP systems. If the OS designer decides to allocate portions of memory for different purposes, there will be a need for a means to prevent unauthorized access to these allocated parts of memory: this is where the ST hardware is not really capable of doing a really good job.

7.1 Creating an Executable AUC Image

Throughout this book, we have used the term “executable memory” to refer to the main memory in the computer – usually implemented using read-only memory (ROM) and randomly accessible memory (RAM). It is called executable memory because the processor is able to fetch instructions for execution from it; memory on storage devices can be read a byte at a time, or a block at a time, but memory on devices cannot be read a variable number of bytes at a time to fetch an instruction. Also, the executable memory is orders of magnitude faster than reading (or writing) any storage device.

Before the control unit can fetch instructions from the memory, the system software must have prepared a program with the instructions represented in the computer’s machine language format, and then have stored the sequence of machine language instructions in the executable memory, for example using the general organization shown in Figure 4.3.

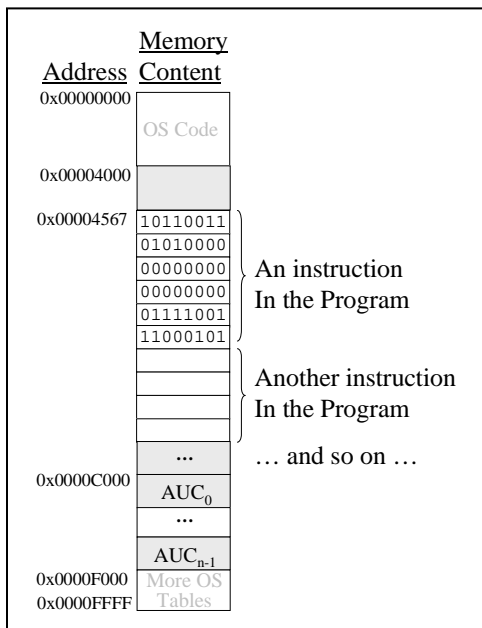


Figure 7-1: The Executable Image

The executable memory image – called the **executable** (or the **binary**, **absolute**, or **object**) form of the program – refers to program in the form that is used when the program is stored in the computer's executable memory, ready for the control unit to fetch, decode and execute it. For modern microprocessors, instructions are usually stored in variable length blocks of bytes (such as 2-6 bytes), depending on the nature of that instruction's format, the machine design, and other factors. In order to store the entire program, many such groups of bytes are used to store the program's many instructions. As a result of this arrangement, when the program is stored in the executable memory in preparation for execution, it is laid out in several contiguous bytes of memory so that the first machine language instruction occupies the first few bytes in memory, the second machine instruction occupies the second few bytes, and so on as suggested by Figure 7-1 (this figure is a refinement of Figure 4.3, but it uses hexadecimal, rather than decimal, numbers for the memory addresses).

For example the instruction written as 48 bits (6 bytes) in Figure 7-1 could represent the symbolic assembly language statement

```
load    R5, 0x000079C3
```

In this hypothetical example, the machine instruction format for this instruction might require 6 bytes (48 bits) to represent the instruction: for example, it might use 8 bits for the operation code (`load`), 4 bits for the register designation (`R5`), another 4 bits for instruction options that are not used in this example (such as indirect or immediate addressing), and 32 bits for the address field (`0x000079C3`). The executable memory image for the 6 bytes that represent the `load` instruction might be something like `0xB350000579C3`, or the 48-bit representation of

```
1011 0011 0101 0000 0000 0000 0000 0111 1001 1100 0101
```

So conceivably, the 6-byte instruction shown at executable memory location `0x00004567` in Figure 7-1 could be the binary representation of the `load` instruction shown above. When the control unit fetches this bit pattern from the executable memory, it will load register `R5` with the 32-bit image that is stored in executable memory location `0x000079C3`.

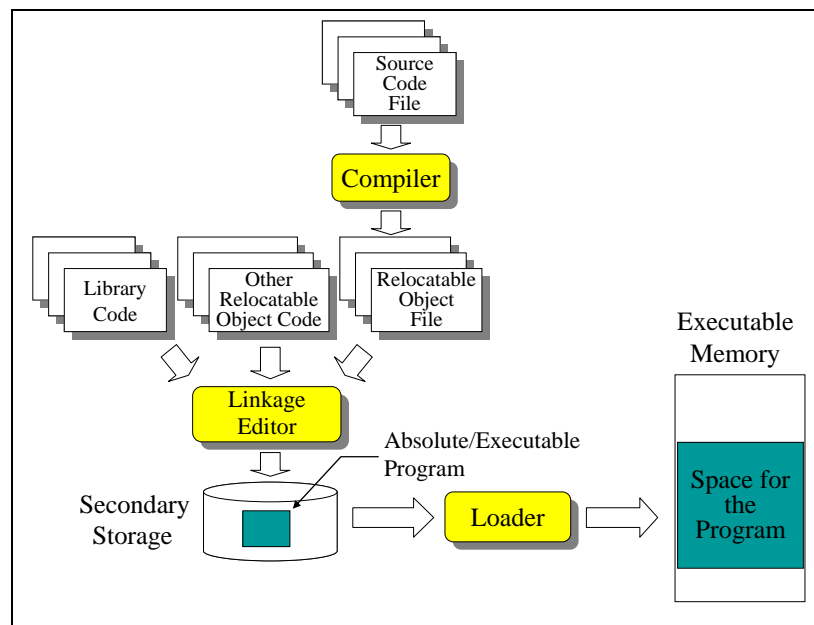


Figure 7-2 Creating an Executable Program

How does the system get from programs written in a **source programming language**, like C, C++, or Java that are stored in a file as (ASCII¹) characters, to a bunch of bits in a particular place in the executable memory that will cause the computer to execute the sequential computation specified by that source program? Let's consider the procedure for creating the in-memory executable image in some detail, starting with the overview in Figure 7-2.

There are 3 phases in the job of preparing the source program for execution: the compile time phase, link time, and load time. Once the program has been placed in the executable memory, it enters a fourth phase called the run time phase.

At **compile time**, the source language translator converts the source program instructions into machine language instructions that perform the desired high level instruction on a target hardware platform (such as an Intel Pentium microprocessor).² The **compiler** creates a relocatable object module (file) into which it places all of the compiler output – the code, data, and information that will be used in subsequent phases of the translation process. As noted in [Section 4.2.1](#), the relocatable object module can reference functions that are not defined within the relocatable object module currently being created. For example, an application program may call functions that are defined in different files (“Other Relocatable Object Code” in the figure), or functions that are kept in libraries such as a standard I/O library, a graphics library, a math library, and so on – the “Library code” in the figure. Once all of the source code has been translated into relocatable object modules or library code, the compilation phase is complete and the next phase of the translation process can begin.

At link time, the **linkage editor** combines all of the relocatable object modules and library functions to create an **absolute** (also called the **load** or **executable module**) representation of the program. When the compiler created each relocatable object module (or a library module), it identified all of the labels in the program that code in some other file might *reference*; for example, the entry points for all functions that are defined in a file are such labels. The compiler gathers all of these external labels that were declared in the file being compiled and saves them in a **definition table** that is bundled with the relocatable object. It does this by determining the names of all of the functions that are referenced in the relocatable object modules, and the name of all of the functions that those functions call, and so on. The compiler also creates a **reference table** of all of the references to entry points and global addresses that are defined in some other relocatable object module or library function. The linkage editor uses these tables determine which library functions to include into the absolute module, along with all of the designated relocatable object modules. Finally, the linkage editor modifies the addresses in the code that was saved in the relocatable object module so that the instructions then use the addresses of the functions that were combined in the absolute module. The absolute module can then be stored in a file that can be retrieved, loaded into the executable memory during the third phase of translation, and then executed on demand (at run time).

The third phase of translation is the **load time** phase during which the OS-dependent loader retrieves the file containing the absolute module and creates the actual run time image as it places the code, data, and stack in the executable memory. Since the exact location at which the absolute program will be loaded into the memory is not generally known until load time, it is necessary to adjust addresses in instructions one last time before the code can be executed. This culminates in the creation of the **run time memory image** in particular locations in the executable memory, thereby completing the load phase.

When the program is to be executed, the fourth phase – the run time phase – begins. The loader provides the main program entry point that it passes to the OS process manager in preparation for initiating execution. The OS transfers control to the program's entry point when it schedules the execution of the program (that is, when it allocates the processor to the associated AUC). The remainder of this section provides additional discussion of compile, link, and load time activity.

¹ “ASCII” is short for the American Standard Code for Information Interchange, a widely used format for encoding ordinary upper and lower case characters in a byte.

² Certain classes of translation systems, such as Java, convert the source program instructions into intermediate language instructions, which can be interpreted by a virtual machine emulator or later translated into target machine instructions. We will presume conventional compiled environments in this overview discussion.

7.1.1 Compile Time Translation

A **compiler** or **translator** is system software that converts each source language statement into one or more machine-language instructions in a relocatable object format (we say that a relocatable object module is created by compiling all of the code in file containing source language statements). The system software must have a compiler for the target source programming language that translates it into machine language for target computer hardware. For example a C compiler for an Intel Pentium is different than a C compiler for a Sun SPARC processor. Likewise, a C compiler for a Pentium is different from a Fortran compiler for a Pentium.

Of course each statement in a high level source program, such as

```
force = mass * acceleration;
```

is compiled into a series of machine language instructions (that we can express as a small block of symbolic assembly language):

```
load   R3, 0x00007904      ; Load acceleration in to register R3
load   R6, 0x00007A12      ; Load mass into register R6
mult   R3, R6              ; Place the product of R3 and R5 in R3
store  R3, 0x00008C44      ; Save the result in the force variable
```

A programmer views a system in terms of the programming language and the runtime³ system interface: the programmer expresses components of the sequential computation using the language, but also calls subordinate functions from the OS and the runtime system. From the programmer's perspective, a function that is implemented in the OS or runtime system is essentially the same as any other individual instruction in the source programming language – there is no need to distinguish between language instructions and functions.

In a compiled-program environment (as opposed to an interpreted-program environment), at *compile time* each file containing a source program (such as a C program) is translated into a **relocatable object module** file (“Relocatable Object File” in Figure 7-2). Programs in the relocatable object format have had all high level language statements converted into machine language instructions, but the *addresses* in the machine language instructions contain extra **meta information** that provides additional information about addresses for the program's variables and references to other functions. For example, the meta information might include a pointer to a temporary table in the relocatable object module that includes the string name, the relative location of the variable in a block of variable storage, and so on. We will not go into the details of such meta information here, as there are many details that should be more fully addressed in a book about compilers.

Each compiler uses its chosen format for relocatable object modules, but a pedagogical example describing the components of the relocatable object module created by the compiler (similar to a conventional C/Unix formation) is illustrated in Figure 7-3. The **Text Segment** in the figure is a block in the module that contains all of the translated machine instructions. The Data Segment contains space for static variables – variables that are not allocated on the runtime stack. The Stack Segment is not actually created at compile time, although the compiler implicitly lays out its organization when it generates machine instruction addresses; the Stack Segment component of the relocatable object module is an abstract (compiler-specific) description of the stack that the runtime system will interpret to instantiate the stack when the program is ultimately executed. The addresses used in instructions are the offset target item from the beginning of the corresponding segment; for example an instruction address is an offset from the beginning of the text segment, a static variable address is an offset from the beginning of the data segment, and an automatic variable is the offset from the beginning of the stack segment. The compiler attaches meta information to each address that will be used by the linkage editor. The Def and Ref tables are used by the linkage editor when it combines multiple relocatable object modules and code from libraries when it builds the absolute (or executable) program image.

³ The system that supports program execution during run time has come to be called the “runtime” system. Hence, we will use “run time” to refer to the phase of program processing during which it is executed, and (in order to conform with standard usage) the term “runtime” to refer to the software that supports execution during run time. It's just one of those flakey little inconsistencies in the computer science jargon.

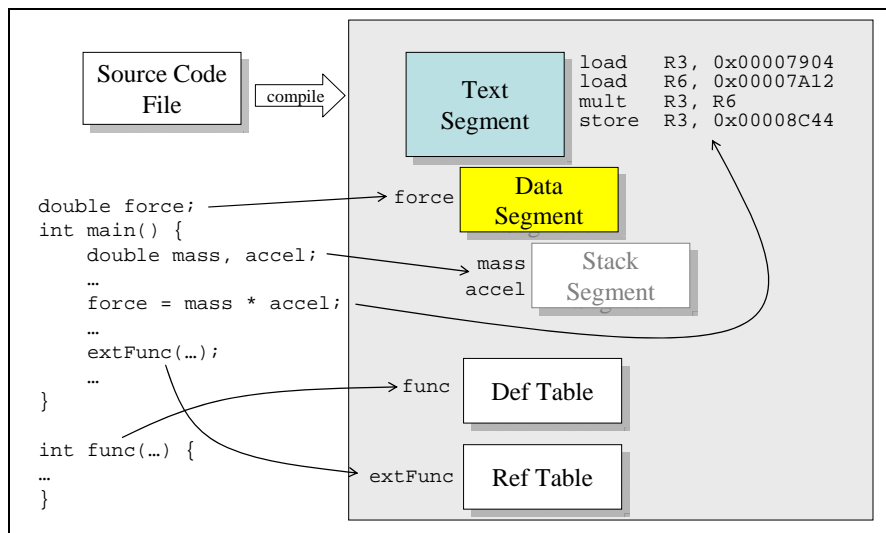


Figure 7-3: Creating a Relocatable Object Module

The strategy for choosing where the variable will have space allocated in the executable memory depends in part on the nature of the programming language. In C, ordinary variables are called automatic variables; such variables have space allocated from the runtime stack, which has its storage allocated in the Stack Segment. Other variables, such as ones that are either globally accessible or are static (ones that retains the last value stored in it even when it goes out of scope), have their storage allocated from the portion of the executable memory associated with the Data Segment.

Whenever the source code references a variable that is allocated in the Data Segment, the compiler uses the *relative address to the beginning of the Data Segment*. If the variable is an automatic variable, the compiler generates code to allocate the variable on the run time stack, meaning that the automatic variable is dynamically created on, and released from, the Stack Segment as the program executes. As a result the compiler generates references for these variables that are *relative to the beginning of the Stack Segment*.

Consider a reference to a procedure entry point for a procedure/function in a relocatable object module. If the target procedure is in the file currently being compiled, then the entry point symbol can be determined by the compiler. However, in many cases, the target procedure is in a *different* relocatable module or library. For example, if the target procedure is a library routine, such as `printf()`, the target function will have been compiled when the system software was built. Since this target address is unknown at compile time, the external symbol cannot be translated until the link editor combines the module that calls the function with the module that defines the function. The compiler will annotate each such *external reference* so that the link editor can place the correct address in the code as soon as it combines the application program with the targeted program. Most compilers gather the external references that appear in a relocatable object module and store them in an external reference table (the **Ref Table** in Figure 7-3).

Finally, consider entry points that are defined in a source program. (An entry point is a function name that can be called from other functions, whether or not they are compiled at the same time as the code that defines the function.) An entry point is defined whenever a function is defined, that is, the programmer writes source code with a function header, and a body that defines its behavior. When the compiler translates a function, it saves the newly defined symbol in a relocatable object module external symbol definition table (the **Def Table** in Figure 7-3).

7.1.2 Linkage Editing

The purpose of the linkage editor is to combine a collection of relocatable object modules and library functions to create an absolute file that contains all of the data and code for a program (organized into a single Text Segment, Data Segment, and implied Stack Segment) – see Figure 7-4. To do this, the linkage editor extracts the text segments from all relocatable object modules and libraries, then combines them into a single text segment for the absolute module. Notice that this means that the addresses that are in the instructions in the relocatable object module text segments need to be adjusted when they are placed in the absolute module. As for relocatable object module addresses, the absolute module addresses are offsets from the beginning of the appropriate segment (text, data, or stack). This procedure is called **relocating** the addresses in the instructions so that they reference the updated addresses in the aggregated data segment.

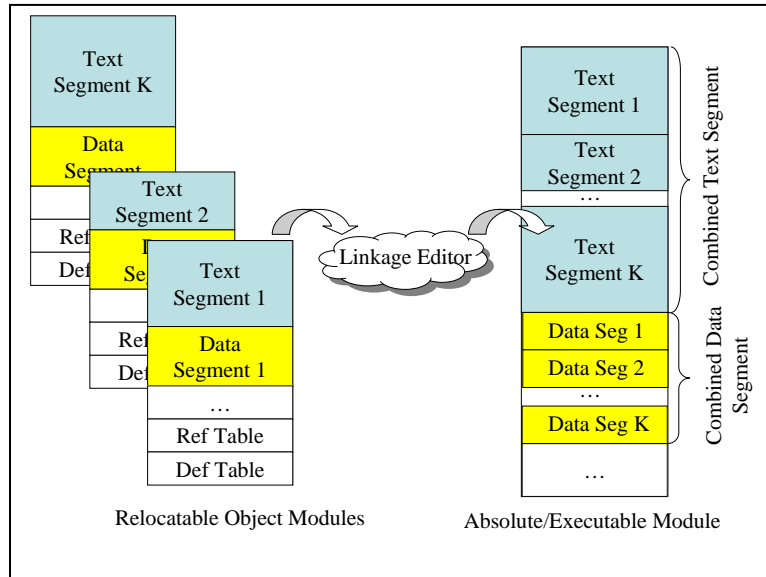


Figure 7-4: Linkage Editor

The linkage editor then matches each entry point *reference* (in each relocatable object module's Ref Table) to *defined* entry point addresses (in the collective Def Tables). All the undefined address references are eventually discovered by the linkage editor as it combines relocatable modules – the resulting composition includes all the program text and data, so that every reference to data or a program entry point is resolved. The absolute module can then be stored in a file (on a storage device) until it is to be loaded into the executable memory and executed.

The principle for linkage editing is easy to understand, but the algorithms to perform the actual repackaging of all of the relocatable object modules and library routines is usually time intensive. The link phase can be the most time consuming part of translation when many relocatable object files and library routines are used in the absolute module.

7.1.3 The Loader

Absolute program modules are created in a format that is understood by the operating system's loader, for example, Unix files are generally in a.out (named after the default file name for an absolute program), ELF (Executable and Loadable Format), or COFF (Common Object File Format) format. You can find the exact specification of these file formats at various web locations. The OS loader chooses a file format that it will support, then it is constructed to read files in the given format, and to place them into a region of the executable memory as directed by the OS.

Since all absolute program addresses are all relative to the beginning of one of the three segments, the loader once again adjusts the addresses in the text segment whenever it loads the text, data, and stack segment into physical executable memory (again see Figure 7-1). For example, if it loads the text segment at memory location 0x00004000, then it must add 0x00004000 to each relative address in each instruction in the text segment. Similarly, static data addresses are adjusted using the address of the first location in the data segment, and stack variable addresses are adjusted by adding the address of the first location in the stack segment. This procedure of adjusting the addresses is called **binding** the program's addresses to the corresponding physical addresses in the memory – in Section 7.1.4 we will see how it can be done at run time instead of load time. The loader then copies the modified absolute program into the block of primary memory.

After the program addresses have been adjusted this last time, and the program has been loaded into the memory, the OS begins executing the program by setting the IC (instruction counter) register in the control unit to the physical memory address of the first executable instruction—the **main entry point**—for the program, and the processor begins to execute the program.

7.1.4 The Memory Load Map

In ST class computer, memory allocation is a simple process (compared to MT and MP systems) – the single application has access to all of the computer's memory. The relevant issue in a ST OS is *how* the memory is partitioned into regions, and how each region is used. The OS does not incorporate any mechanism to allocate memory to different executing programs at different times. Instead, it simply establishes the general characteristics of the memory load map, without setting the boundaries among the text, data, and stack segments. Figure 7-5 is a refinement of the memory load map introduced in Figure 4.3. In the figure, we show an example partition of the Application Space – the part of the memory load map that would be determined by the linkage editor when it combines the relocatable object module and library text, data, and stack spaces. The OS memory manager need not be concerned with setting the boundaries among the internal segments, and there are no boundaries between AUCs in an ST system.

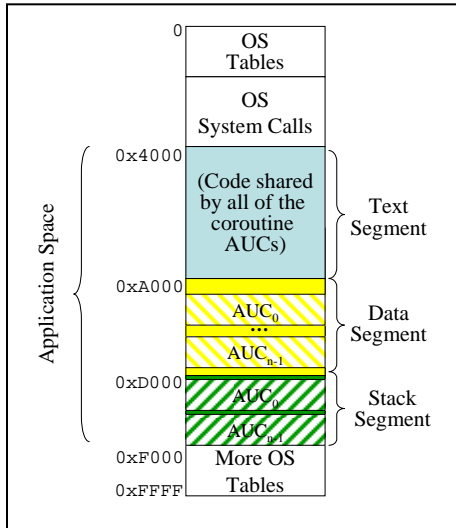


Figure 7-5: A Map of Executable Memory for ST Systems

The translation system and the application itself determine the boundaries between the coroutines. Let's assume that a coroutine can be implemented using a number of functions (it would be difficult to implement interesting coroutines without making this assumption, since they all will naturally need to use library routines such as `Read()` and `Write()`). We would also expect that each coroutine would have its own private static memory, for example to share variables among its internal functions – this is shown as AUC-specific space within the data segment. Because a coroutine will have scopes and functions, it will

need its own stack. Therefore, the stack segment needs to be partitioned for this purpose.⁴ Ultimately, we will want one other part of the memory reserved for saving overall system state information – the “OS Tables” in the figure.

7.2 Dynamic Address Relocation

The difficult part of the loader’s work is to bind the executable image to physical executable memory addresses once it determines exactly where the absolute module is to be loaded. This same problem is a barrier for moving code or data around in the memory once the loader has bound the image to the physical memory. For example, suppose that an application program is written, but when its absolute image is created, it is larger than the amount of physical memory available in the memory load map (see Figure 7-6(a)). There are a few choices: the programmer could attempt to reduce the size of the application program; other parts of the software could be eliminated, thereby making more of the memory available to the large application code; or the program could be designed so that only part of it is loaded at a time. This third technique is a classic programming approach that was widely used in mainstream computers prior to the invention of virtual memory – it is called programming using overlays.

7.2.1 Overlay Loaders

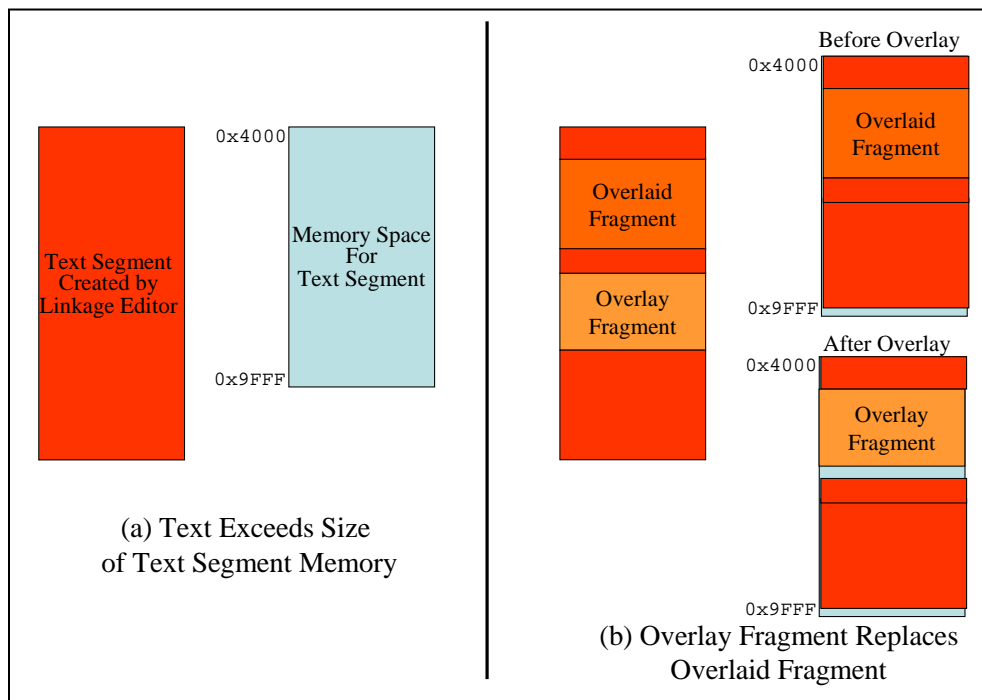


Figure 7-6: Overlay Loading

⁴ It can be tricky to implement multiple stacks as part of a single, shared stack. The Rotor distributed virtual machine environment explicitly provides this type of implementation. For example see Chapter 8, [Nutt, 2005] and Chapter 6, [Stutz, et al., 2003].

An **overlay** is a fragment of the text space that is not initially loaded into the executable memory at load time (see Figure 7-6(b)). The programmer identifies the overlay fragment in the source code representation; the compiler and linkage editor effectively creates the code for the overlay in a special overlay text segment, then the linkage editor creates a text segment image that includes a section of code (the Overlaid Fragment in the figure) that can be overwritten with the Overlay Fragment whenever it is needed. During run time, when the control is about to flow into the (missing) code that is in the Overlay Fragment, the program execution is suspended and the **overlay loader** is called to unload the Overlaid Fragment and to load the Overlay Fragment in the portion of the executable memory that was being used by the Overlaid Fragment. If control flows back into the code in the Overlaid Fragment, then it can be reloaded into the executable memory (by removing the original Overlay Fragment or some other overlayable fragment).

Notice that the same technique could be used in the data segment if the application program happened to create more coroutines than there was space to describe each of them. The system could simply suspend one AUC that was not very active, copy its data segment record to a storage device, and then use the vacated data segment space to accommodate the new coroutine. Historically, this form of overlaying is called **AUC swapping**.

In certain situations, a similar problem can happen with dynamically allocated data structures. Figure 7-6(b) suggests that using overlays and swapping can create losses of memory by creating many small “holes” of unused space. These holes remain unusable because they are too small to accommodate an overlay or a swapped data region. The longer the system runs, the worse this **external fragmentation** problem becomes. A conventional way to handle such fragmentation is by **compacting** the information in the given memory space. This means that code and/or data are moved slightly in memory to remove a small hole; for example, in Figure 7-6(b), compaction would remove the small hole below the Overlay Fragment by moving all of the code below the hole “up” so that the hole would effectively be placed near the end of the text segment (and could be combined with the other hole left there when the text segment was loaded).

Overlays and swapping were heavily used in commercial software in mainframe computers prior to 1980, at which time virtual memory provided functionality to “automatically overlay” fragments of the loaded code and data. ST computers do not have sufficient hardware to support virtual memory, so they can revert to the old overlay technology.

7.2.2 Hardware to Simplify Run Time Loading

There is one big problem with the overlay (and swapping and compaction) approaches: it requires that the loader be dynamically invoked at run time, in order to bind code to physical addresses. This is because each time code or data are moved, then the addresses that referenced that code or data must be adjusted – the job that was done by the (static) loader when it originally loaded the absolute module into executable memory. This approach is called **static address relocation**, since it is all done prior to run time.

Suppose the memory manager had access to an alternative mechanism that was able to bind absolute program addresses at run time, rather than at link and load times – **dynamic address relocation**. Then the memory manager would be able to move a program around in the memory without having to adjust the addresses in the instructions in the program.

Think about the technique that the loader uses to adjust addresses in the absolute module so that they match the physical addresses. Since the absolute module is built as if it were to be loaded at memory location 0, all addresses in the absolute module are relative to the beginning of the module. (Be careful: programs can also have “addresses” other than relative addresses compiled into operand fields. For example, an *immediate* operand must not change when a module is relocated. Some instruction sets include an offset address, meaning the operand is an offset from the current IC contents. This offset enables a program to branch forward or backward the number of addresses specified by the operand.) When the loader determines the actual address of the first location in the module, it adjusts all of the *relative* address by adding the value of the first location to it. In Figure 7-5, each relative address from the absolute module had 0x4000 added to it because the module was loaded at location 0x4000.

By 1980, hardware designers had come up with a relatively simple solution to this problem. They made a simple enhancement to the hardware so that it performed a simple relocation to every memory reference on-the-fly, that is, *each time the CPU makes a reference to memory*. Recall that the linkage editor creates the text, data, and stack segment so that any references to their contents is relative to the beginning of the corresponding segment. For example, if there is an entry point to a function that is offset 0x600 bytes from the beginning of the text segment, the compiler and linkage editor system simply use the address of 0x600 to reference that given entry point. The loader then leaves the address at 0x600 when it loads the code into the in-memory text segment region. Now, when the instruction that references the entry point (for example a machine language `branch` instruction), the hardware adds the physical memory address of the first location in the text segment to target address – in this case, it adds 0x4000 to 0x600 – and then uses the result to access the memory. In the case of the example `branch` instruction, the machine instruction

```
branch    0x600
```

will actually branch to $0x4000 + 0x600 = 0x4600$, which is exactly the right physical memory address in this case.

The simple hardware enhancement to accomplish this **dynamic address relocation** is shown in Figure 7-7. Without dynamic address relocation, each address that is emitted from the ALU or the control unit is sent directly to the Memory Address register in the executable memory. With the enhancement, the CPU incorporates two registers and an adder. The Relative Address register contains a copy of the “virtual” address (0x0600 in our example), and the Relocation Register contains the address of the (first byte in the) text segment. The adder computes the sum of the integers in the two registers, and then send the result to the executable Memory Address register. The address from the absolute image is dynamically relocated to reference the proper physical executable memory address. This simple hardware modification greatly simplifies the problem of dynamically loading/moving code and data at run time, and is one of the great early inventions in systems. Some variant of it is used in nearly every contemporary microprocessor.

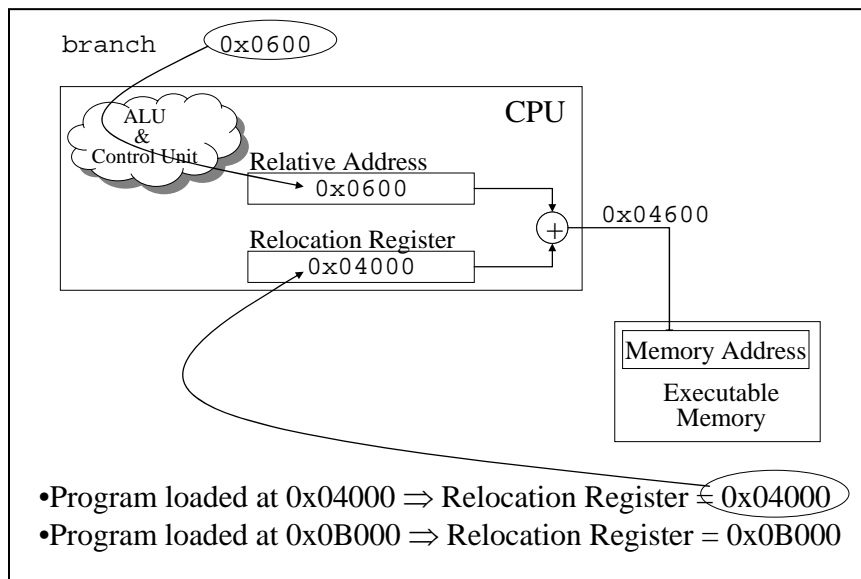


Figure 7-7: Conceptual Hardware for Dynamic Address Binding

The conceptual discussion above gives you the general idea for how dynamic address relocation hardware is used to solve the dynamic loading problem. However, in its commercial realizations, it is slightly more complex. Note that contemporary compilers and linkage editors emit code and data into three different memory segments: text, data, and stack. The Intel 8086 architecture incorporates 4 relocation registers rather than just one: code segment (CS), data segment (DS), stack segment (SS), and an extra segment (ES) relocation registers (see Figure 7-8).⁵ This was intended to support the C/Unix compilation model (with the text segment being renamed the code segment). Any one of the four relocation registers could be used to relocate a relative address: the Intel 8086 hardware is designed so that the ALU and control unit select the correct register. If the control unit is currently issuing an address to the memory so that it can fetch the next instruction to be executed, then it also selects the Code Segment Register to be added to the contents of the Relative Address register. The ALU (arithmetical logical unit) can select any one of the other three registers, depending on which kind of address it is currently transmitting to the Relative Address register. If the relative address is a stack reference, then it selects the Stack Segment Register; but if the address is a data segment reference, then it selects the Data Segment Register to relocate the Relative Address. The Extra Segment Register is used to relocate the Relative Address if the programmer explicitly uses it in an assembly language instruction – C programs do not generally use the ES register at all.

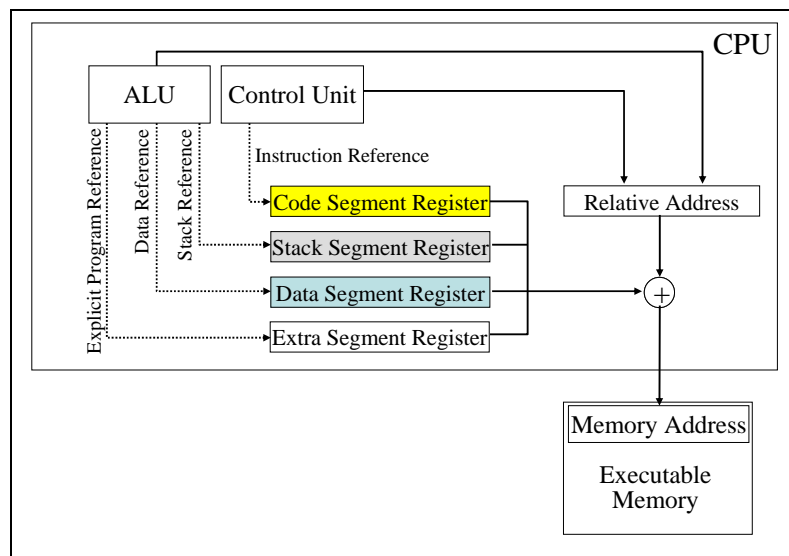


Figure 7-8: Intel 8086 Multiple Segment Relocation Registers

The principle that is in use here is that the AUC can have N different blocks in the executable memory. The CPU hardware would then need to incorporate N different relocation registers, one to address the base of each of the N different blocks. Next, the CPU needs to have a means to select a particular relocation register for the block for which the relative address is referring.

This is the perfect scenario for supporting coroutine AUCs, since we want the coroutines to all use the same code segment and data segment registers, but to each use their own stack segment register value. Then, all coroutines share the program code and global data, but each coroutine can have its own stack. To do this, the stack segment register needs to be changed each time the system begins to execute a new coroutine – that is, it is a line or two of code in the `Resume ()` system call.

⁵ There are many good references to explain this. For example see Chapter 3 of [Messmer, 1995].

7.3 OS Run Time Memory Support

There are various memory management functions that must be handled dynamically, meaning as the application runs, rather than being handled by the static memory layout and dynamic hardware relocation. This section describes the runtime heap storage to support dynamic data structure allocation, and then the sandbox model that can be used to provide surprisingly strong barriers between AUCs (provided that they are only implemented in strongly typed languages like Java).

7.3.1 Heap Storage

Modern programming languages take advantage of dynamically allocated storage for data structures. For example, C++ allows programmers to dynamically create and destroy objects using the language primitives `new` and `delete` operations. The C runtime library includes two library calls, `malloc()` and `free()`, for manipulating dynamic storage allocation (`new` and `delete` are typically implemented using these functions). The `malloc()` function is used for requesting memory space for dynamic data structures. The programmer requests more space by writing a code sequence such as

```
struct ListNode *node;
...
node = (struct ListNode *) malloc(sizeof(struct ListNode));
...
```

When the `malloc()` call returns, `node` points to a memory block large enough to hold an instance of the `struct ListNode` data structure.

After the AUC has finished with the dynamic storage, it releases it with a call of the form

```
free(node);
```

which releases allocated block of memory for the node data structure to be released back “to the system.”

Traditionally these functions operate on a data structure that manages a large block of memory that has been reserved for use by `malloc()` and `free()` called a **heap** storage.

In general purpose Unix systems a heap storage is reserved for each process, then `malloc()` allocates space for dynamic data structures from within the process’s private heap. It is worthwhile to look more closely at how this is done in a Unix process. The linkage editor creates a Unix process memory layout similar to the one shown in **Error! Reference source not found.** (the details differ for different Unix variants). Like the ST memory load map, the C runtime system and Unix linkage editor allocate space for the Code (or text) Segment, the Data Segment and Stack Segment. Since the size of the Stack Segment depends on the the execution path through the program, C/Unix reserves a large block of the memory load map for the combination of the stack and the heap. Thus, C/Unix does not require that the maximum size of the stack nor heap storage be known prior to run time. Instead, it determines the maximum size of the memory load map address space, allocates a region for the text/code and data segment, and for other miscellaneous items (including main program arguments and environment variables), and then leaves a large hole in the memory load map for the dynamically allocated storage – the heap and stack. The stack begins at the high address end of the region and grows downward as the stack increases. Meanwhile, the heap starts at the low address end of the region and grows upward as the process allocates heap space (using `malloc()`); the stack and heap grow toward one another as the process executes.⁶

⁶ Since Unix is a MP OS, it is supporting multiple processes, each with their own stack. Therefore it has a *memory allocation* policy and mechanism by which it allocates space to processes. ST operating systems do not have processes – just a single threaded application program. In Unix, a process can grow its address space if it uses all of the stack and heap, but in an ST OS, when the region is full, the memory is full.

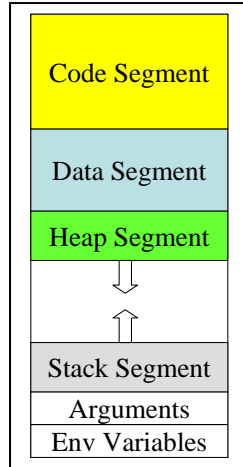


Figure 7-9: UNIX-style Memory Layout

This tangential discussion of the C/Unix heap storage suggests a refinement from our original patterns for ST memory layouts: while the entire address space is shared among all the AUCs, by exploiting dynamic memory relocation hardware, it is possible to design the system so that each AUC has its own, private mini address space. That space need not contain a code segment, nor segments for the arguments and environment variables. However, it can include a data and stack segments for the AUC. However, the heap would normally be shared among all AUCs, suggesting a memory layout similar to that shown in Figure 7-10. The heap storage that AUCs use for objects and dynamic data structures are allocated from the heap (using a pointer into the heap to dereference the object). These pointers, themselves, can be allocated in the AUC’s local private storage area, for example in its data segment.

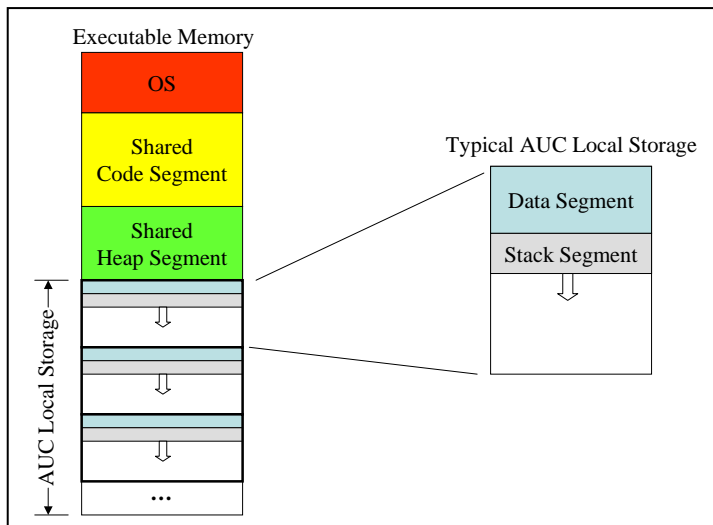


Figure 7-10: ST Memory Layout

7.3.2 The Address Space Abstraction

Memory managers have evolved with operating systems and hardware technology. However, the API to the executable memory is about the same as the one that was used in 1950. Generally, an executing program is provided with a collection of linear addresses that are used to read and write the array of bytes in the executable memory hardware. In the earliest days of operating systems – and in many ST class computers – the linear address space that the programmer uses is the physical executable memory address space. Our prior discussion of how compile/link/load time works is based on this simple model. The dynamic hardware relocation introduces a twist, since the executing program contains addresses that are not the same as the actual physical memory addresses, that is, the dynamic hardware relocation changes the addresses just before they are issued to the executable memory unit.

Because of the decoupling of the addresses used in the program from the physical memory addressed, OS designers are careful to distinguish an AUC's logical addresses from the physical addresses it uses. We say that the AUC has an *address space* that contains every address that the AUC can use. Coroutines share the code and data components of an address space, but each has its own address space for the stack. The dynamic relocation hardware translates addresses from the AUC's address space into physical memory addresses.

The idea of an address space allows translation software to create and save absolute program images without being concerned about the details of **binding** each abstract address to a physical address in the executable memory (see Figure 7-11). With the hardware dynamic address relocation facility, this binding is done each time the CPU issues a memory read/write request. With multiple relocation registers (such as the code segment, stack segment, and data segment registers of the 8086 architecture) the abstract address space can be divided into segments and stored in 3 distinct areas of the executable memory.

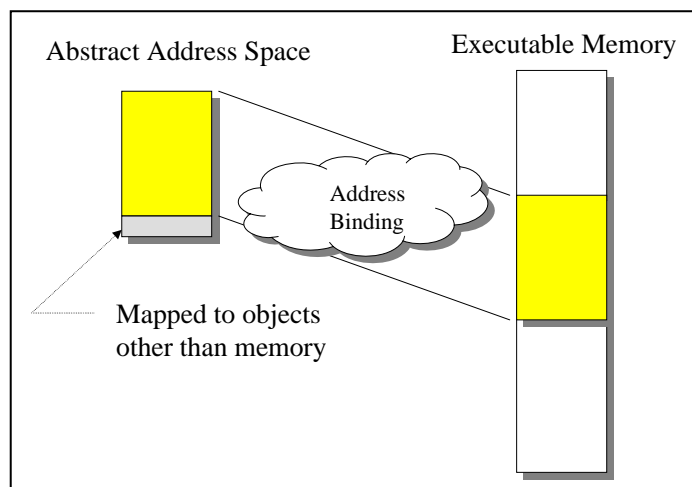


Figure 7-11: The Relationship Between the Address Space and Primary Memory

The memory manager need not actually participate in dynamic address relocation, except to control the settings of the relocation registers. Even in a system with cooperative AUCs (such as coroutines), the memory manager must keep track of the relocation register contents for each of an AUC's 3 segments; the `resume()` OS call will be responsible for saving the old AUC's relocation register settings, and placing the new AUC's relocation register settings into the CPU relocation registers.

7.3.3 Mini Address Spaces: The Sandbox Model

Contemporary computing environments often encourage the use of “mini” address spaces within a classic address space. OO systems encourage the idea of a private mini address space for each object. The extension to mobile code systems (such as Java and .NET) rely on the mini address spaces to define an address space for a *software component* – an address space that is larger than an object’s address space, but smaller than an AUC’s address space.

Suppose that every program that is to run in a restricted mini address space were written in a programming language that enforced the use of consistent types – typically called a strong typed language. Such languages are designed so that every language element has a type, and so that elements of one type, say an expression, cannot be assigned to an element, say a variable, of a different type. For example, an integer value cannot be stored in a variable that is a pointer to an integer. Furthermore, operations that might violate the type rules can generally be detected at compile time.

Programs written in a strong typed language cannot use pointers to arbitrary memory locations – since each pointer must be to a typed element, and the location of the element is somewhere within the scope of the program. This effectively prevents a program from reading or writing variables that are not part of the program. This has come to be called the sandbox model, since such environments allow a programmer to write an arbitrarily complex program, provided that they do not read or write variables “outside the sandbox.”

7.4 Summary

Memory management refers to all aspects of managing a computer’s executable memory so that it best serves the needs of the application programmer. In this chapter you have learned how software for a ST is translated from a collection of source programs into an absolute object module – one that is ready for the OS loader to copy into the computer’s executable memory. You also learned that this absolute program must have its addresses adjusted, depending on which part of the executable memory in which the program is to be stored. Static loaders perform this adjustment after the OS allocates the memory, but before it copies the code into the memory. Dynamic loaders that use hardware dynamic address relocation can defer the last relocation step until run time by using one or more segment relocation registers.

Dynamic address relocation enables the OS to move executing code around in the memory, and to change the size of memory allocated to an entity. This, in turn, enables the ST memory manager to easily support AUC local storage for the AUC’s stack and other local information. This provides a more robust AUC execution environment, and definitely gives the memory management more responsibility for correct behavior of the computer system.

Finally, you learned more about the importance of address spaces in modern operating systems. Using the C/UNIX heap model, this chapter suggested how the memory manager can provide AUC local storage to each AUC in the community while preserving the shared memory containing code and global data.

7.5 References

1. Messmer, Hans-Peter, *The Indispensable PC Hardware Book: Your Hardware Questions Answered*, Second Edition, Addison Wesley, Reading, MA, 1995.
2. Nutt, Gary, *Distributed Virtual Machines: Inside the Rotor CLI*, Addison Wesley, Boston, MA, 2005.
3. Stutz, David, Geoff Shilling, and Ted Neward, *Shared Source CLI Essentials*, O’Reilly, Santa Clara, CA, 2003.