CHAPTER 8

Memory Management

Learning Objectives

After reading this chapter you should be able to:

- Describe how multiprogramming is achieved by sharing the main memory.
- Explain address binding requirements at various stages such as program compilation, linking, loading, and runtime.
- Discuss three runtime address translation schemes, namely segmentation, paging, and paged segmentation.
- Explain memory allocation and deallocation mechanisms.
- Describe memory sharing and protection.

8.1 Introduction

The main memory is a vital hardware resource in a computer. It stores programs (and their data) during their execution. In essence, each executable is a sequence of machine instructions and the CPU executes these instructions one by one to carry out the intended task. The CPU can only execute instructions from the memory (and not from the I/O devices). That is, the instruction that the CPU executes and its associated data (aka, operands) must be in the main memory; operands can come from the operating registers of the I/O controllers.

Recall that in each CPU execution cycle, the CPU first fetches the instruction from the main memory, next it brings the data from the main memory if required, then executes the instruction, and finally stores the result, when produced, back into the main memory. So, program instructions and data are accessed from the main memory during their executions. That means, ideally, in each CPU execution cycle, the program counter (also called instruction pointer) must point to an absolute memory address

» The analytical engine proposed by Charles Babbage, the forerunner of modern computers, had four components: mill, store, input unit, and output unit. There, the store played the role of the main memory of today. Although in modern usage the term store refers to secondary storage, some people still refer to the main memory as store.

» For reasons of efficiency, some intermediate data may be kept in the CPU registers, for some time, during the execution of a program. However, those data first brought into the CPU registers from the main memory are eventually written back into the main memory. Consequently, almost all data accessed during program execution are in terms of absolute memory addresses.

to access the instruction. Moreover, instruction executions must access the data from the memory in terms of their absolute memory addresses. Hence, for a program to be executable, it must be written in terms of the absolute memory addresses.

Creating an executable as a machine-level program (either directly from scratch or translating from a higher-level program), using absolute memory addresses, would be relatively straightforward if the following conditions hold:

1. Enough memory is available to hold the entire executable program and its associated data during its execution.
2. The available memory is contiguous.
3. The address range of the available memory is known at the creation time of the executable.

This was the case in closed shop era, when programs were small and the entire memory was available for a single program during its execution. It is hard to maintain the above conditions in modern systems for many obvious reasons. We describe some main reasons here:

- First, these days as most programs are enormous in size, they require large memory.
- Second, typically more than one user or program is allowed to access the computer at any moment. For example, the operating system offers services executing many of its own utility programs simultaneously, and individual users typically execute more than one application program at any given time.
- Third, during the development of a program, neither the timing of its execution nor the range of memory addresses that will then be available for its execution is known. That is, in modern multiprocess- and multiprogram systems it is not realistic to have knowledge in advance of the address range the program may use in its execution.

Moreover, from the resource utilization point of view, enforcing the above conditions is not wise either. These factors bring us to the fundamental question of how modern systems prepare and execute application programs. The objectives of this chapter and the next are to answer this question systematically. The system responsible for preparing- and managing executable programs in the memory during their executions is called the *memory management system*. This system essentially addresses how the three conditions are relaxed in modern systems.

Although these conditions appear independent of one another, their solutions share many common features. The combined solution is far more efficient than individual solutions to the conditions. From the learning point of view, however, relaxing the conditions individually would make it easier to understand the techniques and solutions employed. This chapter addresses the issues of relaxing the conditions 2 and 3 of the list above. Assuring that conditions 2 and 3 are met renders the solution to memory size limitation easier, or else it becomes complicated. The issue of memory size limitation, per se and in combination with relaxing conditions 2 and 3, is discussed in the

next chapter. However, emergent issues such as *memory sharing* and *protection* that arise due to multiprocessing are discussed in this chapter.

The operating system, all the resident application programs, and processes share the main memory, as a whole. Memory management is critical in multiprocess systems. In modern operating systems, memory management is done at two levels: real memory and virtual memory. In this chapter, we describe the techniques of sharing the main memory among application programs and processes. In the next chapter, we discuss virtual memory, a memory management technique to run large programs on limited available main memory.

8.2 Memory-management System

The main memory is a vital hardware resource that stores the programs and data required by the CPU and the I/O devices. The CPU can only execute programs from the memory (and not from the I/O devices). A program must be first brought into the memory before the CPU can execute it. The storing of many applications in the memory at one time enhances utilization of the CPU and other resources, and improves performance of the system in general.

The operating system itself permanently occupies a portion of the memory for its own programs and their static data. This part is referred to as the *static memory*. The remaining portion of the memory stores application programs and their static data, and the dynamic data of processes and of the operating system. This part is the *dynamic memory*. Both application programs and processes, and the operating system use the dynamic memory. In the rest of this chapter and the next, unless specified otherwise, by main memory or memory we mean the dynamic part of the main memory.

» The operating system uses the dynamic memory to store its dynamic data. Kernel modules (both code and data) that are linked to the kernel at runtime are also stored in the dynamic memory. In Fig. 1.4 on page 15, the lower half in the dynamic memory.

The main memory is of limited size. Memory space is recycled to store application programs, process data, and dynamic kernel programs and data. The subsystem that manages the allocation and deallocation of memory space is called the *memory manager*. Its responsibilities are many:

- **Incremental allocation and deallocation:** Its primary responsibility is to incrementally allocate memory to its clients as and when their needs arise, and to reclaim it when they no longer require the allocated memory.
- **Sharing:** Let us examine the meaning of the term memory sharing in different situations. When we say two programs or the operating system share the memory, we mean that they are stored in the memory at different locations: two programs or data *cannot* reside in the same memory locations simultaneously. This is known as *space division sharing* or *space multiplexing*. The memory manager enables the clients to share (i.e., space-multiplex) the memory smoothly. They can reside at the same memory location, though at different times. This is known as time-multiplexing. The other kind of sharing as discussed in Section 6.4.2 is the *shared memory region* that refers to a (dynamically) allocated portion of the memory shared simultaneously by many processes. This is true sharing of memory cells. A key

- Space Multiplexing
- Time "
- Shared Region

responsibility of a memory manager is it must promote, whenever possible, automatic (true) sharing of the same memory cells among multiple processes. Let us review Fig. 4.4 on page 91 now. A process consists of many blocks of information content. In this chapter, the reentrant code and read-only data are collectively called a program code, and the rest are called process data. These days, the program is non-modifiable, and hence, many processes can "truly" share the program blocks without affecting the behavior of one another. Such sharing helps avoid replicating the same program in many memory regions, and thereby improves utilization of the memory. However, other processes cannot automatically share a process's data without the consent of the latter process.

» Codes have an invariant part (instructions and constants) that is not changed during execution, and a variable part (temporary storage and data) that may be changed during an execution. Multiple processes can share the invariant part. The processes each have their private variable part. Some mechanism is necessary to allow the invariant part to access the appropriate variable part of different processes. A few mechanisms are discussed later in this chapter.

Protection: The third responsibility of a memory manager is to protect the operating system and application programs from application processes, and the processes from one another. Operating system programs and data (static or dynamic) must be protected from malicious use and faulty application processes at all costs, otherwise the system may exhibit unpredictable behaviour. Memory protection is utterly necessary for reliability and correct functioning of computer systems. We need to protect memory belonging to one process from unauthorized accesses by other processes.

When a process is created, we may not allocate memory for its entire address space. When the need arises we incrementally add memory to the process; for example, due to expansion of the process stack. The need for automatic memory allocation arises from the need for program modularity (any module can be stored anywhere in the main memory), machine independence (program development should not depend on the quantum of available main memory), and resource sharing (regions of the main memory may be shared by multiple processes). A good memory manager needs to ensure these objectives.

In other words, memory management is critical in multiprocess operating systems. This chapter and the next discuss policies toward the management of the main memory space. Memory management is centred on a few issues: (1) units of memory allocation, (2) where in the main memory the units are constructed, (3) when the units are allocated and how, (4) how programs/processes reference these units, (5) whether or not units are sharable by different processes, (6) how the units are protected from unauthorized accesses.

» The aim of the memory manager is to maximize memory utilization with minimal overheads while incrementally allocating and deallocating memory in piece-meals to clients. The memory manager is constraint with protection and hardware-based address translation during memory allocation. Also, all memory locations may not be accessible to DMA devices.

Assigning suitable memory addresses to instructions and data is an underlying issue in all these tasks and is handled by introducing various levels of address abstractions and suitable translation schemes. Each program is initially written with their own addressing scheme or address space, which are independent of the one available physical address space. This gives rise to many questions such as what is an address space, what are the different address spaces used in this context, how is the translation from one space to another done, what are the system utilities involved in this process, how are the addresses finally mapped or bound to physical addresses during program execution, how are the spatial and temporal connectivities maintained, etc.

Address handling is a continuous theme of memory management and its various components. This will be discussed throughout this chapter.

The operating system expects some support from the underlying processor hardware for efficient memory management and memory protection. Without hardware support, good memory management may not be possible or would be highly inefficient; some memory management features would not be possible to implement. The hardware circuitry helps the operating system make memory management robust in the event of programming errors or malicious programming practices.

8.3 Memory Hardware

Let us review what roles the main memory plays in its management. Figure 8.1 presents a model of a typical memory device. We can visualize the main memory as a linear array of (memory) cells; the array index starts at 0 and rises to some maximum integer number. The cells are all of the same size, and each cell is a linear array of bits. Cells are primitive storage units, and units of information transfer across memory boundary. When a cell is referenced, all the bits in the cell are referenced as a single indivisible unit.

The memory system is a passive device in the sense that it does not interpret (or act based on) the content of the cells; neither does it modify the content on its own. It treats cell contents as some kind of abstract information. It only guarantees that whatever information is stored in a cell can be repeatedly retrieved later from the cell without alteration. Each cell has its own physical address, from zero up to some maximum integer number. The memory system supports two primitive operations, namely read and write to fetch values from and to store values, respectively, into individual cells. Cells can be referenced in arbitrary order by invoking these primitives on their physical addresses. A write operation on a cell overwrites the current value, and a read operation returns the value stored at the latest write operation.

A memory device receives cell access requests from the processor and the I/O controllers, and carries out these requests on corresponding cells.

» As far as memory management is concerned, a memory device is a transparent hardware unit. The memory hardware is not involved in management decisions. A memory device sees only streams of memory operations, and executes these operations atomically on cells. It may not be interested to know who generates these operation requests, and how they are generated. That is all the memory hardware does.

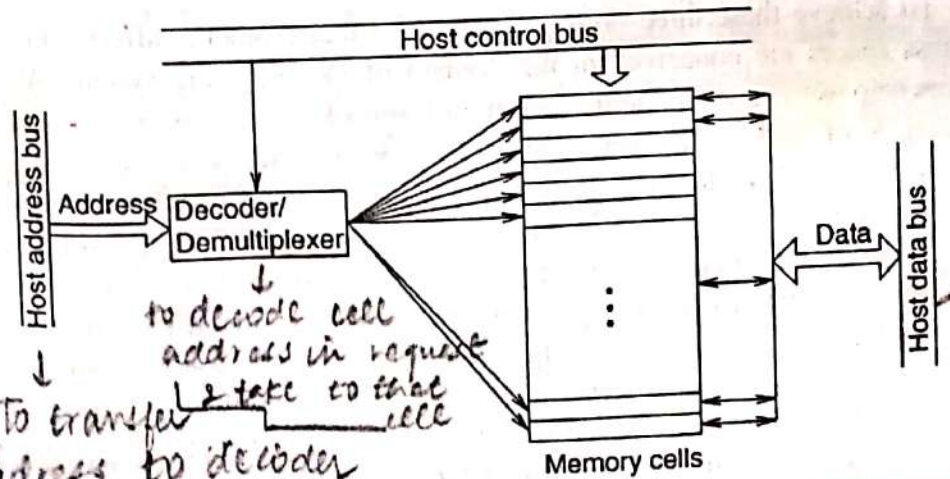


Figure 8.1: *Memory address decoder.*

Each request consists of a cell address and a primitive operation (read or write). A demultiplexer (decoder) reads the cell address from the host address bus, decodes it, and activates the appropriate individual cell to carry out the primitive operation. Cells are connected to the host data bus to transfer data between the cells and the other hardware units.

8.4 Address Spaces and Address Translation

In mathematics, a *space* consists of a set of entities and a set of relations on the entities, but here our only interest is in the set and not in the relations. Each entity in the set is referenced by a distinct name. An *address space* is a space where entities are locations that are referenced by using addresses, that is, it is a set of addresses. An *address* refers to a location (in the address space) that stores a piece of information. Distinct addresses refer to distinct locations in the address space. Again, we do not concern ourselves here with how addresses in the same address space are related to one another.

It was noted in passing in Section 8.2 that memory management facilitates achieving the following three objectives.

1. **Machine independence:** There is no a priori correspondence between an application program and its placement in the main memory. To achieve this, programs must *not* reference memory cells by their physical addresses. Programs use different kinds of addresses to reference the stored contents.
2. **Program modularity:** An application program may be developed as a collection of loosely connected subprogram units instead of a big whole. The units can be placed anywhere in the main memory. They may not be linked together until at the execution request for the application. If need arises, the units may be relocated at runtime without the notice of the program execution.
3. **Program sharing:** Multiple processes may automatically share parts of programs, and the processes would not notice such sharing.

To achieve these three objectives and also many others, different kinds of address spaces are conceived in the domain of the operating system. Address spaces help us to view a system at different levels of abstractions and evaluate the challenges of memory management tasks. Note that the same piece of information may reside in different address spaces in different forms. For example, the addresses a higher-level application program uses to reference information are distinct from the addresses the memory hardware recognizes to reference memory cells. Higher-level application programs use identifiers to reference information. (We are here using address as a generic notion distinct from memory locations.) What we need is some mapping schemes to correlate addresses, from different address spaces, that refer to the same piece of information. A mapping scheme is also referred to as an *address binding* or *translation scheme*.

The address space is an important concept in operating systems. We need to understand this concept to understand memory management and address translation schemes. In the following subsections, we discuss three different kinds of address spaces: (1) memory or physical address space, (2) program identifier space, and (3) process or logical address space.

8.4.1 Memory-address Space

In a computer system, the bus address space is partitioned into I/O address space and memory address space. The I/O address space represents device registers (I/O ports), and the CPU uses I/O addresses to access I/O ports. The memory address space represents memory cells or locations, and the CPU uses memory addresses to access the cells. The CPU accesses I/O ports by executing IN/OUT kind of instructions, and memory locations by executing read/write/move kind of instructions. In some processor architectures, parts or whole of the I/O address space may be mapped into the memory address space. In such cases, the CPU can access I/O ports by reading and writing values at the corresponding memory mapped I/O addresses.

The elements in the memory address space are called *physical addresses* or *memory addresses*. A physical address refers to a single cell in the main memory. Different physical addresses refer to different cells in the main memory. The processor and I/O devices send out physical addresses on the system address bus to access contents of memory cells. The cells are the real storage units where the information is stored. The memory address space is the real address space where each location stores some information in a physical form that the memory device understands. (Every other address space discussed in this chapter is an abstract address space.) Memory address space size is limited by the largest address the system address bus can hold. If there are n lines in the address bus, the memory space size is at most 2^n.

» A physical address corresponds to electrical signals sent on different wires of the system address bus. The memory hardware understands only these signals.

8.4.2 Program-identifier Space

Application developers write their applications in various higher-level programming languages such as C, C++, Java, etc. An application program consists of many variables and functions, and these are connected by a set of syntax rules supported by the programming language. The rules refer to the variables and functions by their names, popularly called *identifiers*. Users view "application memory" as a placeholder of all program identifiers, where their values are stored in the way the users imagine. (The methods the memory manager adopts to actually allocate physical memory to these identifiers and to represent their values in the physical memory are immaterial to users.) An application program uses the identifiers to reference information stored in the "application memory". For example, when an identifier for a variable is read-referenced in a program, the user expects that the "application memory" system returns the latest value of the variable to the application. A *program identifier space* or *name space* consists of a set of identifiers that are referenced by their names to retrieve- or update information, (see Fig. 8.2). In

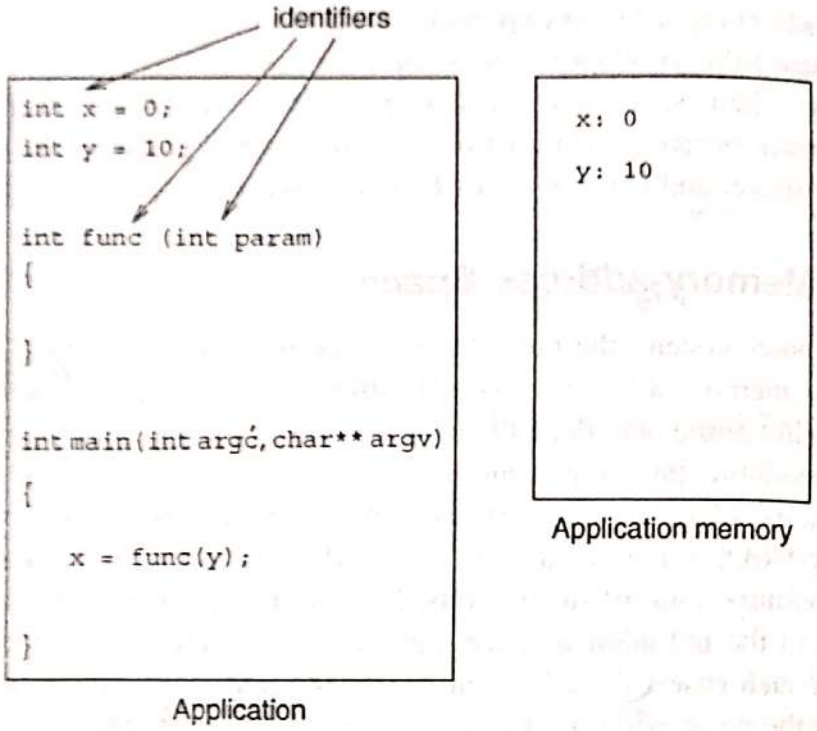


Figure 8.2: *Program identifier space abstraction.*

the figure, when the application reads *y* to evaluate *func*, it expects the "application memory" returns 10. The names are 'symbolic addresses' in the "application memory". Here space elements are program identifiers, and different identifiers usually refer to different elements in the program identifier space. Every application program has its own program identifier space.

» We do not concern ourselves how two or more different identifiers can refer to a single piece of information, as supported by many programming languages.

8.4.3 Address Translation

At one end, users view each application program as a set of identifiers connected by well-defined syntax rules. At the other end, the main memory understands physical addresses, but not the identifier names. Neither does the CPU understand program identifiers. It understands only machine instructions, primitive data, and their addresses. We need to translate each application program into a form that the CPU understands. The translated program is stored in the main memory in a way that the CPU and the main memory can work together to execute the program to produce the correct output.

Address Binding

In general, there may not be a priori correspondence between a program identifier space and the memory address space. To execute a program we need to map its program identifier space into the memory address space. The identifiers in the program may be placed anywhere in the memory address space; users do not concern themselves with it. Thus, we need to "bind" program identifiers to physical memory addresses before the CPU can reference them. The CPU references the identifiers using the physical memory addresses that are bound to them. The application operates in the program identifier space, and the CPU in the memory address space (see Fig. 8.3).

Address binding is the mechanism of associating entities from two different address spaces, and is a mapping from one address space into another. In computer program development and program execution, address bindings are done at several stages: (1) compile time, (2) load time, and (3) runtime. Depending on the stage of binding, computer systems (the compiler, the loader, the linker, the operating system, and/or the hardware) employ various address mapping schemes.

Translation Utilities

Unless a program is written in the machine language, we need address translation schemes to convert the program identifier space into the memory address space. The translation is done with the help of the system utilities (compiler, assembler, linker), the operating system software (memory manager), and the processor hardware (MMU) at different times. Address translation schemes collectively help applications become machine-independent in the sense that there is no a priori correspondence between a program identifier space and the memory address space. Application programmers are relieved of the burden of resolving memory management issues leaving them to concentrate on the design and development of their applications.

A *compiler* is a special utility that converts plain texts (containing higher-level programs), also called *source code*, into binary objects that are linked together to form an executable (see Fig. 8.4). Executables are often called *binary images* or simply binaries (comprising machine instructions, primitive data, and addresses).

A compiler reads program texts and generates equivalent machine language programs from them. As shown in the Fig. 8.4, in modern programming environments, compilation is done in pieces. The compiler fixes addresses for all program identifiers defined in each binary object, and marks

» As mentioned previously, before a source (higher level) program is executed, it must be transformed into an equivalent machine language program and then loaded into the main memory. In modern systems, the task of loading a translated program into the main memory is logically distinct from the translation of that program.

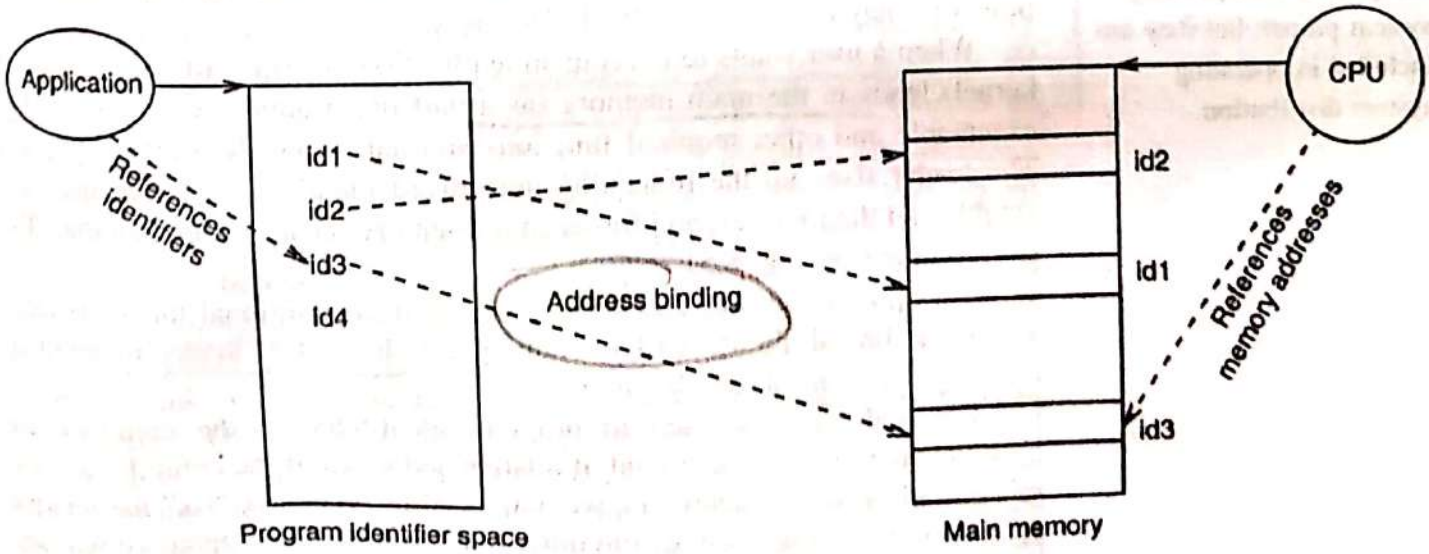


Figure 8.3: *Address binding between program identifiers and memory addresses.*

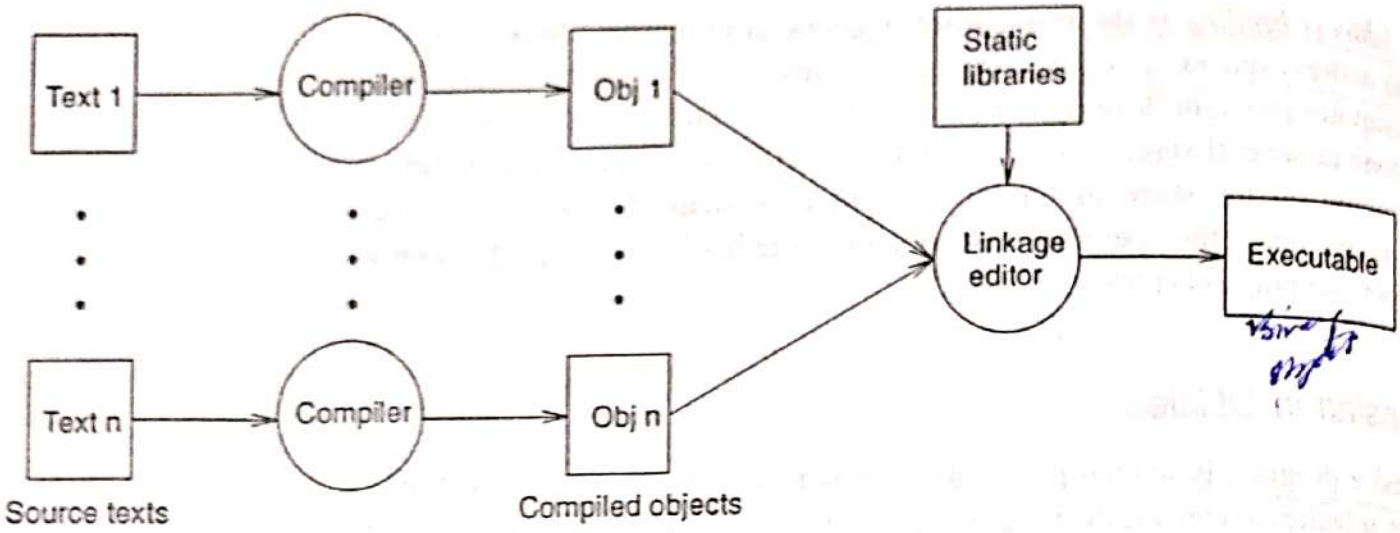


Figure 8.4: *Translation of application programs into an executable.*

» External identifiers are the primary facility through which independently compiled subprogram units communicate with one another.

» A compiler is more programming-language dependent than operating-system dependent. The opposite is true in linker because it has to comply with the load file format of the system. Neither of them is a part of the operating system proper, but they are included in operating system distribution.

identifier not defined in the object as unresolved external identifiers. These identifiers are public or external, and are defined in other program units. The individual binary objects are then combined together later to produce the executable binary image. It is a way to fix positions of program pieces with respect to a common base reference position. Such combination can be done at several junctures: (1) at compilation time, (2) after compilation but before linking time, (3) at linking time, (4) after linking but before execution time, and/or (5) at execution time. We discuss these next.

A *linker* (or linkage editor) is another utility that links together several compiled binary objects and static libraries, as shown in Fig. 8.4, to form a single, coherent binary image. Binary objects contain machine code, primitive data, their addresses, and other information that allow the linker to combine them together to form an executable. The linker fixes up unresolved identifiers between binary objects, where an identifier referenced in one binary object is actually defined in another binary object. Note that an executable, however, may contain unresolved identifiers.

When a user wants to execute an application, a *loader* (usually a part of kernel) loads in the main memory (as a part of an application process) the executable and other required functions and data from the shared libraries. The loader fixes up the remaining unresolved identifiers between the executable and the libraries, and forms an in-memory binary image that the CPU can execute (see Fig. 8.5).

During its execution, an application may need additional functions from a dynamic linked library (DLL). At that point, the DLL is loaded in the main memory and unresolved identifiers are fixed.

The addresses assigned to program identifiers by the compiler are absolute addresses or some kind of relative addresses. If the compiler assigns relative addresses to identifiers, we may need to "cook" or "fix" the relative addresses at the link, load, or run time. In the rest of this chapter we will see how address fixing is done at various stages of address binding.

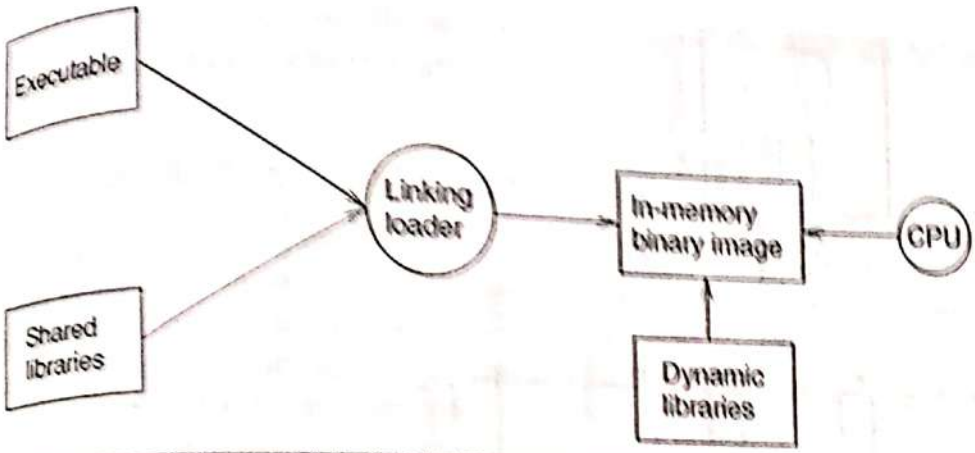


Figure 8.5: *Loading an executable.*

8.4.4 Address-translation Schemes

Application developers should not be burdened with address binding know-how needed at various stages of program translation. The operating system and its utilities (the compiler, the linkage editor, and the loader) and the processor hardware map the program identifiers to the appropriate memory addresses where the identifiers physically reside. Figure 8.6 presents a schematic of a typical end-to-end address-binding scheme: (1) A compiler transforms a higher-level application program into a number of translated units; (2) A linker links the units to form an executable; (3) The operating system allocates physical memory to hold the executable and other shared libraries, and loads them into the allocated memory.

In the figure, the reading of variable x is translated into fetching of physical address 1000. It has to be ensured that by the time the CPU starts executing the program, the binding of program identifiers to physical memory addresses is complete. How this is done is discussed next.

» The gist is that by the time the CPU references an identifier, its address must be fixed, because otherwise chaos ensues in executing the application.

Compile Time Address Translation

If at the time of a program compilation we know where in the main memory the program will be loaded, the (absolute) physical addresses for all identifiers can be resolved at the compilation time itself, and there is no need for load-time or runtime address translation support. The compiler appropriately binds program identifiers to physical addresses, such that all addresses in the executable are physical memory addresses. Memory regions have to be earmarked for a program at the compile time. This is called *static allocation*. The correspondences between program identifiers and their physical memory addresses are fixed at compile time, and do not change at link, load, or runtime. The executable is called the *absolute binary image*. The process of loading an absolute binary at the designated memory area is very simple. The loader is called *binary loader*; it does no cooking of the addresses referenced in the binary. At runtime, the translated program directly uses physical addresses to reference values for program identifiers. Once translated, a program must always be loaded at the same physical memory region if it is to execute correctly. Otherwise, the program

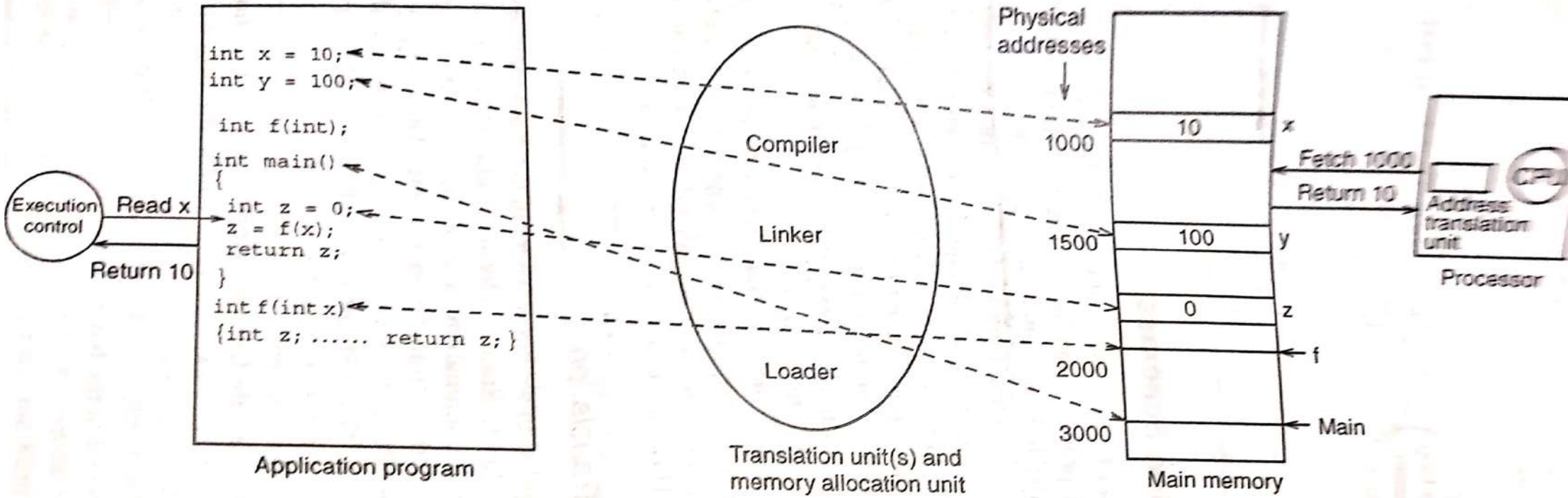


Figure 8.6: *A general address-binding scheme.*

will not produce correct results unless we recompile the program before loading it into a new memory region.

Load Time Address Translation

In multiprogramming systems, it may not be possible to know in advance where in the main memory a binary image will be loaded for execution. It should be possible to load a binary anywhere in the main memory depending on the availability of the required quantum of memory. This is called *dynamic allocation*. (Sharing of programs and data also calls for their dynamic allocation.) As there is no a priori knowledge of the memory region where a compiled unit will be loaded, there cannot be any static correspondences between program identifiers and physical memory addresses. That is, there should be no dependence of instructions and data on the current location of the program in the main memory. Consequently, the compiler (and linker) will not be able to bind program identifiers to absolute physical addresses. Instead, it produces what is called a *relocatable binary image* that can be loaded anywhere in the main memory with minimum address cooking effort. That is, the executables are relocatable binary images.

A relocatable binary image does not refer to any physical memory addresses. All addresses are normally specified relative to zero, and are called *relative addresses*. If the image is blindly loaded in the main memory starting at physical memory address 0, the program will execute correctly. For any other placement of the binary image, the image needs appropriate cooking (of relative addresses). Each binary image contains relocation information that indicates which address fields in the image must be relocated. The final binding of program identifiers to physical addresses is delayed until load time and/or runtime. A *relocatable loader* is responsible for loading into the main memory a relocatable binary image and updating all relative addresses depending on where in the main memory the image is loaded.

For each program unit, the compiler associates relative addresses to program identifiers. The translated program references the identifiers within the same unit by their relative addresses. The identifiers that are defined outside the unit are marked unresolved by the compiler. Each compiled unit or binary image contains a symbol table defining the external symbols. When these units are combined, the linker fixes all those unresolved identifiers to form a single loadable unit. These units can be loaded anywhere in the main memory without any recompilation of the original programs.

If a computer system does not have runtime address translation support, translation of the relative addresses to absolute physical memory addresses at load time is needed. For this purpose, the operating system employs a special facility, called *linking loader*, that fixes physical addresses for all the relative addresses based on where in the main memory the units are loaded. Once loaded and linked together, the units cannot be relocated in the main memory during their execution. The address cooking is fixed for the duration of the program execution, and is called *static relocation*. All bindings are resolved by the load

time (and in-memory link time). The CPU then generates physical addresses when it executes the program, and no runtime address translation is needed.

Runtime Address Translation

Static relocation hampers performance in some systems. In these systems, we may need to relocate programs at runtime. That is, compiled units may be relocated anywhere in the main memory during their executions. This is called *dynamic relocation*. Consequently, we cannot bind relative addresses to physical addresses at load time or in-memory linking time. When the CPU executes the program, it generates relative addresses and not memory addresses. The relative addresses are bound to physical addresses at runtime depending on where the translated units are relocated at that point of time. If a compiled unit is indeed relocated at runtime, we have to update the corresponding address binding information. The binding information is dynamic. The processor hardware automatically translates the relative address at runtime. Without processor hardware support, dynamic relocation is not possible.

» If you had ever used a debugger such as gdb, you see the addresses of identifiers in relative addresses.

8.4.5 Logical Address Space

As is evident from the discussion in the previous subsection, in modern computer systems, normally where in the main memory a binary image will be loaded for execution is not known in advance. The compiler/linker cannot bind program identifiers to absolute physical memory addresses. Instead, they assign relative addresses to the identifiers. Modern operating systems also prefer dynamic relocation. When the CPU executes the binary, it references the identifiers by their relative addresses. These CPU-generated addresses are called *logical addresses* in contrast to physical memory addresses. The logical addresses collectively comprise what is called a *logical address space*. The CPU-generated logical addresses are translated into the corresponding physical addresses by the processor hardware at runtime.

» Some authors refer to relative address as *virtual addresses*. In this book, however, we avoid referring to them as virtual addresses.

In modern operating systems, address binding is done in two steps as shown in Fig. 8.7. The compiler and linker do the first step, and the processor hardware the second step. Each binding step acts as a mapping between the two address spaces: the first between the program identifier space and the logical address space, and the second between the logical address space and the memory address space.

» There might be a little confusion here. Logical address is a generic name here. Note that the CPU executes instructions in the process context. When a process is in the user mode, a logical address refers to its private address space, when in the kernel mode, it refers to the kernel space. You have now been warned about the confusion!

Program executions (i.e., processes) operate in their individual private address spaces. The operating system associates with each process a distinct private address space containing all the logical addresses that the process is allowed to reference. As far as an individual process is concerned, there is no physical memory. It always executes instructions and accesses data from its private address space using logical addresses. The process does not concern itself about where and how its private address space is placed in the main memory. There may not be any a priori correspondences between the private address spaces and the memory address space. If using compile or load time

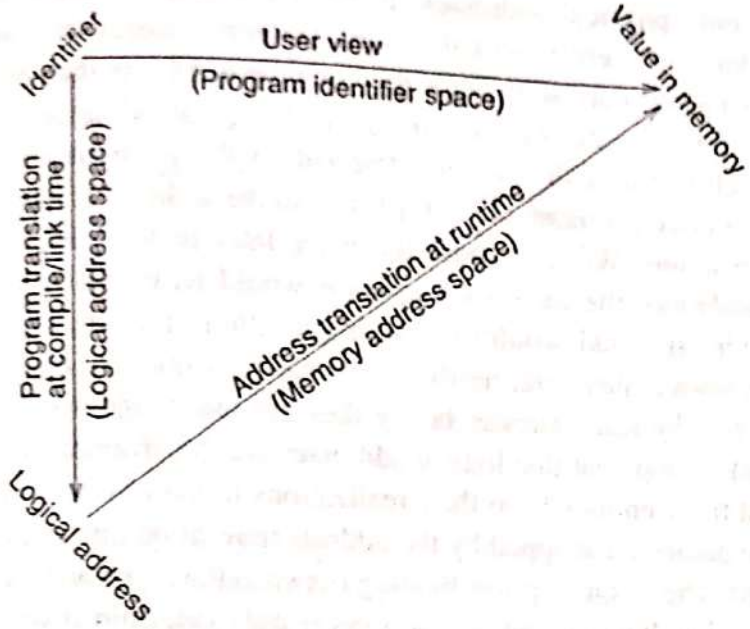


Figure 8.7: *Address binding through logical space.*

static binding forms a private address space, logical addresses are the same as physical addresses. Otherwise, they are usually different. Henceforth, in this chapter, we will deal with logical address spaces and the memory address space, and the mappings from logical address spaces to the memory address space.

» Readers may think that logical addresses are names of information, and memory addresses are locations in the main memory that hold the information.

In summary, application developers deal with program identifiers in their applications. The compiler and linker deal with logical (or relative) addresses and do the binding between the program identifiers and logical addresses. The address translation units in the processor do the binding between logical addresses and physical addresses at runtime.

Runtime address translation has added benefits. Program loading is simple in the sense that no cooking of logical addresses to physical addresses is needed. Programs can be relocated anywhere in the main memory at runtime. In the extreme, if the operating system decides to suspend a program execution for a long duration, the entire address space can be moved to a secondary storage device. This task is called *swap out*. Later, the address space will be swapped into (anywhere in) the main memory when the operating system decides to resume the suspended program execution. Address translation for private address spaces is process-specific, and the operating system maintains the individual address binding information. All the operating system needs is to keep track which parts of a private address space are placed where, and this distribution information is stored in memory management data structures. The data structures aid the processor hardware in address translation.

8.4.6 Runtime Address-translation Schemes

In the rest of this chapter and the next, we concentrate on two kinds of address spaces: many logical address spaces and the one physical memory address space. The CPU executes instructions from logical address spaces, and always generates logical addresses. The memory hardware, however,

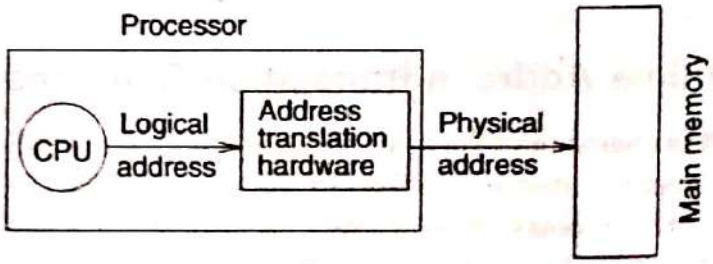
recognizes only physical addresses. In early computer systems, logical and physical addresses were identical, but in modern systems they are distinct. There is no a priori correspondence between them. Unless they are the same, we need a runtime translation unit to map logical addresses to physical addresses before the addresses are directed to the main memory. (We, of course, pay a price for increased complexity in the addressing mechanism and address translation. We talk about the price later in this chapter.) Without address translation, the memory hardware would treat logical addresses as physical addresses, and would access information at wrong memory cells. Processor hardware, therefore, needs to automatically translate logical addresses to their correct physical addresses before they are sent to the main memory.

It would be apparent that logical addresses are the abstract names of some entities, and these entities form their realizations in the physical addresses. At runtime, the names are mapped by the address translation unit in the processor (see Fig. 8.8). There is no a priori binding between the name and the form. The target physical address of a logical address is only determined when the CPU produces the logical address. This is called *runtime* or *execution time binding*.

Theoretically, any logical address may be mapped to any memory address. We can represent this map by a linear array of physical memory addresses. Let us call the array a *MapTable*. Figure 8.9 presents a schematic of address translation via the MapTable. When the CPU generates a logical address, the address is used as an index to the MapTable to obtain the corresponding physical address. The physical address is sent to the memory device to complete the memory reference by the CPU. The CPU operates on the logical address space, and the main memory on the physical memory address space. The MapTable binds the two address spaces at runtime. Two consecutive logical addresses need not be stored in two consecutive memory addresses; they can be placed anywhere in the main memory. It is a very simple address translation scheme.

Processes for their executions do not require MapTables. The tables are required by the memory management subsystem. As each process has its own private address space, it must have its own MapTable to store the mapping information. The MapTables are stored at a priori known region in the main memory. The process descriptor in the kernel contains a reference to its MapTable. When the operating system selects a process to run, it sets appropriate parameters for the hardware translation unit so that the unit accesses information from the right MapTable.

The aforementioned address translation scheme, though very simple, is inefficient on three counts. (1) A MapTable is as large as the logical address space it represents. Thus, for each process we need an additional amount of



re 8.8: *Runtime ad-*
s translation.

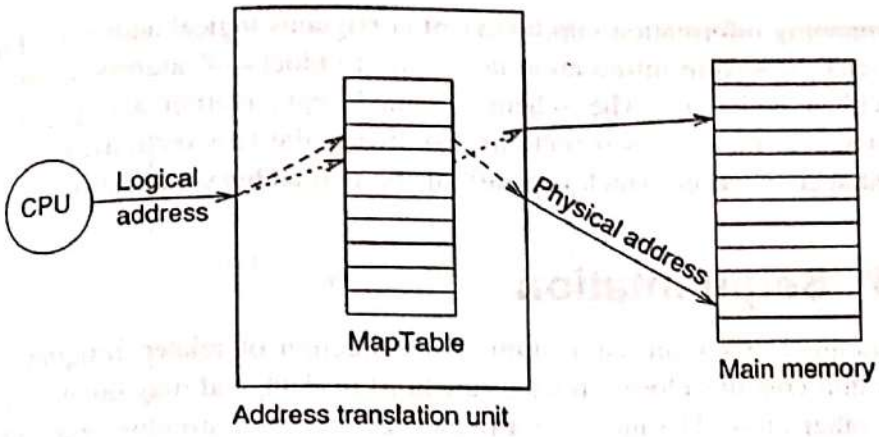


Figure 8.9: *A naive runtime address translation scheme.*

physical memory that is as large as the size of its address space. We, therefore, need to reduce the size of MapTables to make address translation space efficient. (2) Address translation slows down memory reference speed by a factor of two. Every logical address referenced by the CPU forces the processor to make two physical memory references: one to get a MapTable entry, and the other to access the actual referenced entity from the main memory. As no logical address directly refers to any physical memory location, every logical address needs to go through the address translation scheme at runtime. The translation scheme must be very fast as otherwise it will slow down computation speed by a factor of two. (3) The memory manager has to maintain information about each memory cell, that is, whether it is free or mapped to some logical address space. The amount of information required for cell tracking has to be reduced.

The mapping scheme described above is very general. It is designed to help in mapping any logical address to any physical address. This generality may need to be restricted to enable runtime address translation space efficient, faster, and cost effective. At one extreme, an address space is consecutively loaded in the main memory starting at a base address. The address translation unit is informed about the base address. Every logical address generated by the CPU is added to the base address by the translation unit before it is sent to the main memory. The schematic of this address translation scheme is presented in Fig. 2.8(b) on page 59. This scheme does not need a MapTable, and the address translation is as fast as the speed of an adder. The scheme, however, is very restrictive for efficient memory management, because it is necessary to allocate consecutively an entire address space in the main memory. As one base register will also prevent code protection and sharing, we need at least two—one for the invariant part (reentrant code) and the other for the data part to promote code sharing. To make code and/or data more sharable, we need many more than two base registers so that we can share many independently written programs and many private data areas.

In the following two sections we will study two practical feasible address translation schemes that require considerably lesser mapping information, and that are extensively used in modern operating systems. Each scheme groups

the mapping information into blocks of contiguous logical addresses. Entries in MapTables store information pertaining to blocks of addresses instead of individual addresses. The schemes, called segmentation and paging, are discussed in the next two sections. Following the two sections, we present paged segmentation, which is a mix of the two address translation schemes.