Memory management

Program must be brought (from disk) into memory and placed within a process for it to be run

- Main memory and registers are the only storage CPU can access directly"
- Memory unit only sees a stream of addresses + read-requests, or address + data and write-requests"
- Register can be accessed in **one** CPU clock (or less)"
- Main memory access can take many cycles.
- **Cache** sits between main memory and CPU registers, typically on CPU chip"
- Protection of memory required to ensure correct operation, between processes from OS and users, and between processes from users"

Not only are we concerned with the relative speed of accessing physical memory, but we also must ensure correct operation. For proper system operation we must protect the operating system from access by user processes. On multiuser systems, we must additionally protect user processes from one another. This protection must be provided by the hardware because the operating system doesn't usually intervene between the CPU and its memory accesses (because of the resulting performance penalty). We first need to make sure that each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 8.1.

The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error (Figure 8.2). This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

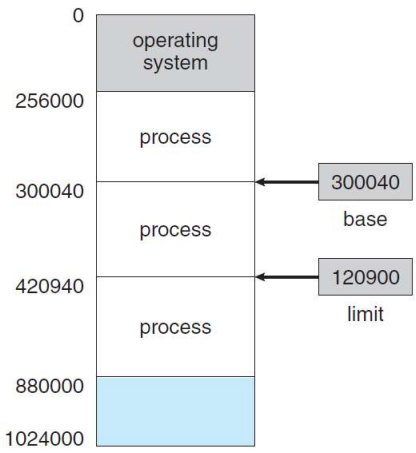

Figure 8.1 A base and a limit register define a logical address space.

The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers.

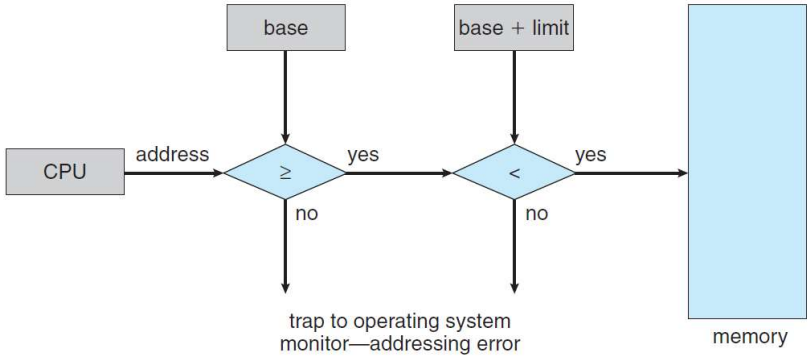

Figure 8.2 Hardware address protection with base and limit registers.

This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents. The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors,

to access and modify parameters of system calls, to perform I/O to and from user memory, and to provide many other services. Consider, for example, that an operating system for a multiprocessing system must execute context switches, storing the state of one process from the registers into main memory before loading the next process's context from main memory into the registers.

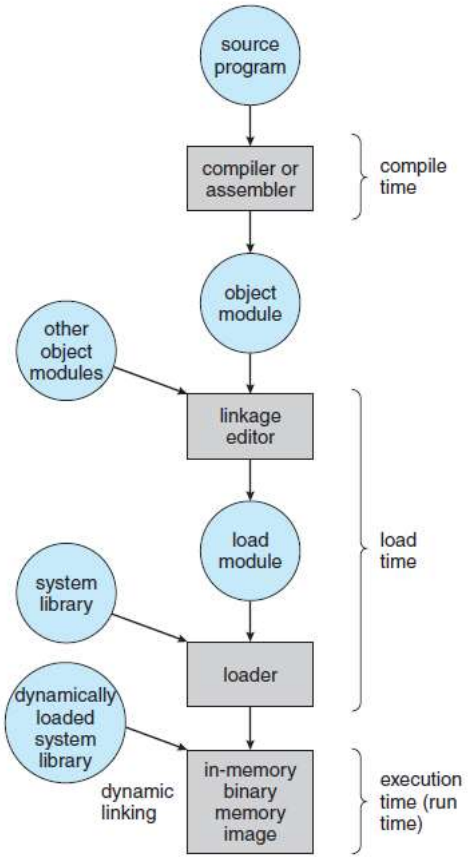

Figure 8.3 Multistep processing of a user program.

Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit—that is, the one loaded into the **memory-address register** of the memory—is commonly referred to as a **physical address**.

The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address**. We use ***logical address*** and ***virtual address***

interchangeably in this text. The set of all logical addresses generated by a program is a **logical address space**. The set of all physical addresses corresponding to these logical addresses is a **physical address space**. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

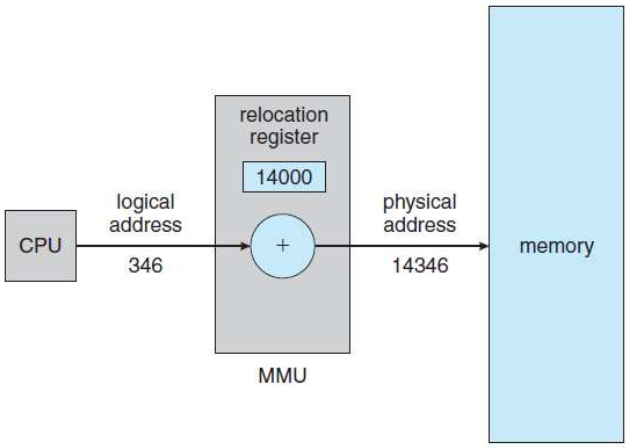

Figure 8.4 Dynamic relocation using a relocation register.

The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**. We can choose from many different methods to accomplish such mapping. The base register is now called a **relocation register**.

The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory (see Figure 8.4). For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346. The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses—all as the number 346. Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register. The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses.

We now have two different types of addresses: logical addresses (in the range 0 to *max*) and physical addresses (in the range $R + 0$ to $R + max$ for a base value R). The user program generates only logical addresses and thinks that the process runs in locations 0 to *max*. However, these logical addresses must be mapped to physical addresses before they are used. The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

Dynamic Loading

In our discussion so far, it has been necessary for the entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been limited to the size of physical memory. To obtain better memory-space utilization, we can use **dynamic loading**. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine. The advantage of dynamic loading is that a routine is loaded only when it is needed. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.

Swapping

A process must be in memory to be executed. A process, however, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution (Figure 8.5). Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

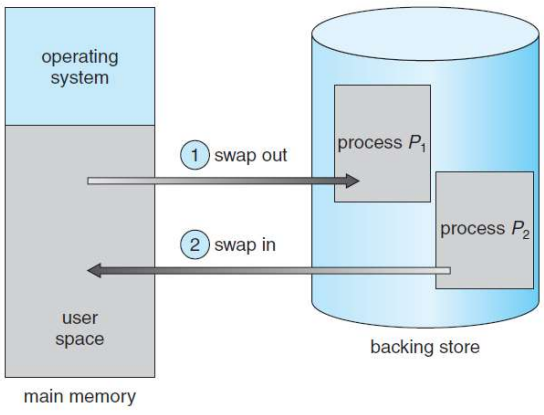

Figure 8.5 Swapping of two processes using a disk as a backing store.

swapping involves moving processes between main memory and a backing store.

Swapping on Mobile Systems

Although most operating systems for PCs and servers support some modified version of swapping, mobile systems typically do not support swapping in any form. Mobile devices generally use flash memory rather than more spacious hard disks as their persistent storage. The resulting space constraint is one reason why mobile operating-system designers avoid swapping. Other reasons include the limited number of writes that flash memory can tolerate before it becomes unreliable and the poor throughput between main memory and flash memory in these devices. Instead of using swapping, when free memory falls below a certain threshold, Apple's iOS asks applications to voluntarily relinquish allocated memory. Read-only data (such as code) are removed from the system and later reloaded from flash memory if necessary. Data that have been modified (such as the stack) are never removed. However, any applications that fail to free up sufficient memory may be terminated by the operating system.

Android does not support swapping and adopts a strategy similar to that used by iOS. It may terminate a process if insufficient free memory is available. However, before terminating a process, Android writes its application state to flash memory so that it can be quickly restarted. Because of these restrictions, developers for mobile systems must carefully allocate and release memory to ensure that their applications do not use too much memory or suffer from memory leaks. Note that both iOS and Android support paging, so they do have memory-management abilities.

Memory Protection

Before discussing memory allocation further, we must discuss the issue of memory protection. We can prevent a process from accessing memory it does not own by combining two ideas previously discussed. If we have a system with a relocation register, together with a limit register, we accomplish our goal. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). Each logical address must fall within the range specified by the limit register. The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory (Figure 8.6).

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process. The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically. This flexibility is desirable in many situations. For example, the operating system

contains code and buffer space for device drivers. If a device driver (or other operating-system service) is not commonly used, we do not want to keep the code and data in memory, as we might be able to use that space for other purposes. Such code is sometimes called **transient** operating-system code; it comes and goes as needed. Thus, using this code changes the size of the operating system during program execution.

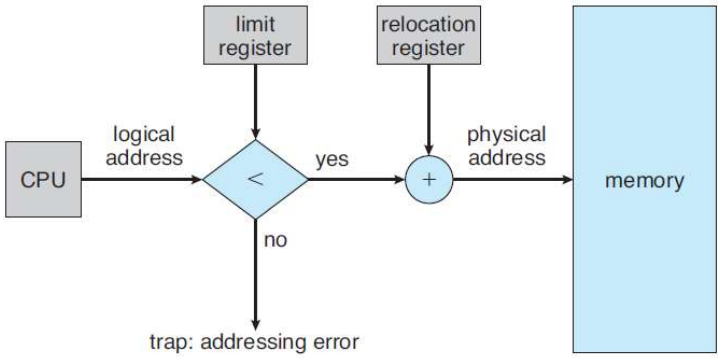

Figure 8.6 Hardware support for relocation and limit registers.