2.5 Hardware Protection

2.5.1 Dual-Mode Operation

To ensure proper operation, we must protect the operating system and all other programs and their data from any malfunctioning program. Protection is needed for any shared resource. The approach taken by many operating systems provides hardware support that allows us to differentiate among various modes of execution.

At the very least, we need two separate modes of operation: user mode and monitor mode (also called supervisor mode, system mode, or privileged mode). A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: monitor (0) or user (1). With the mode bit, we are able to distinguish between a task that is execute on behalf of the operating system, and one that is executed on behalf of the user.

At system boot time, the hardware starts in monitor mode. The operating system is then loaded, and starts user processes in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to monitor mode. Thus, whenever the operating system gains control of the computer, it is in monitor mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to auser program.

The dual mode of operation provides us with the means for protecting the operating system from errant users, and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as privileged instructions. The hardware allows privileged instructions to be executed only in monitor mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction, but rather treats the instruction as illegal and traps it to the operating system.

The concept of privileged instructions also provides us with the means for the user to interact with the operating system by asking the system to perform some designated tasks that only the operating system should do. Each such request is invoked by the user executing a privileged instruction. Such a request is known as a system call.

When a system call is executed, it is treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to monitor mode. The system-call service routine is a part of the operating system. The monitor examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The monitor verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call.

2.5.2 I/O Protection

A user program may disrupt the normal operation of the system by issuing illegal I/O instructions, by accessing memory locations within the operating system itself, or by refusing to relinquish the CPU. We can use various mechanisms to ensure that such disruptions cannot take place in the system.

To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus, users cannot issue I/O instructions directly; they must do it through the operating system. For I/O protection to be complete, we must be sure that a user program can never gain control of the computer in monitor mode. If it could, I/O protection could be compromised.

Consider a computer executing in user mode. It will switch to monitor mode whenever an interrupt or trap occurs, jumping to the address determined from the interrupt vector. If a user program, as part of its execution, stores a new address in the interrupt vector, this new address could overwrite the previous address with an address in the user program. Then, when a corresponding trap or interrupt occurred, the hardware would switch to monitor mode, and would transfer control through the (modified) interrupt vector to the user program!

The user program could gain control of the computer in monitor mode. In fact, user programs could gain control of the computer in monitor mode in many other ways. The operating system, executing in monitor mode, checks that the request is valid, and (if the request is valid) does the I/O requested. The operating system then returns to the user.

2.5.3 Memory Protection

To ensure correct operation, we must protect the interrupt vector from modification by a user program. In addition, we must also protect the interrupt-service routines in the operating system from modification. Even if the user did not gain unauthorized control of the computer, modifying the interrupt service routines would probably disrupt the proper operation of the computer system and of its spooling and buffering.

We see then that we must provide memory protection at least for the interrupt vector and the interrupt-service routines of the operating system. In general, we want to protect the operating system from access by user programs, and, in addition, to protect user programs from one another. This protection must be provided by the hardware.

To separate each program's memory space, we need the ability to determine the range of legal addresses that the program may access, and to protect the memory outside that space. We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 2.6.

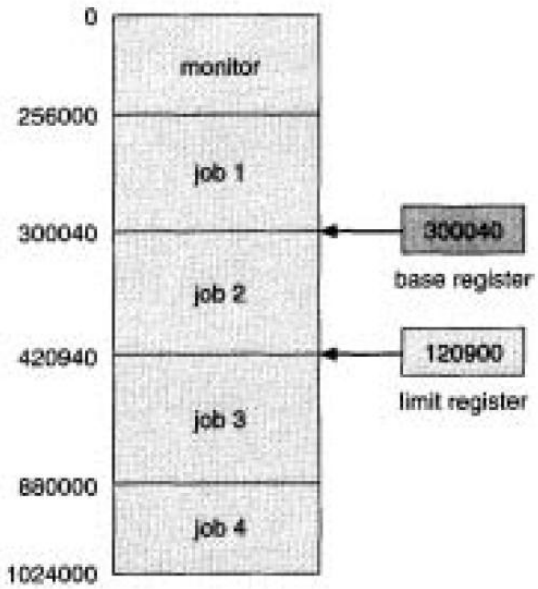

Fig. 2.6 A base and a limit register define a logical address space.

The base register holds the smallest legal physical memory address; the limit register contains the size of the range. For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 inclusive.

This protection is accomplished by the CPU hardware comparing every address generated in user mode with the registers. Any attempt by a program executing in user mode to access monitor memory or other users' memory results in a trap to the monitor, which treats the attempt as a fatal error . This scheme prevents the user

program from modifying the code or data structures of either the operating system or other users.

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

This scheme allows the monitor to change the value of the registers, but prevents user programs from changing the registers' contents.

The operating system, executing in monitor mode, is given unrestricted access to both monitor 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, and so on.

2.5.4 CPU Protection

In addition to protecting I/O and memory, we must ensure that the operating system maintains control. We must prevent a user program from getting stuck in an infinite loop or not calling system services, and never returning control to the operating system. To accomplish this goal, we can use a timer. A timer can be set to interrupt the computer after a specified period. The period may be fixed or variable. A variable timer is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs.

Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the

program more time. Clearly, instructions that modify the operation of the timer are privileged. Thus, we can use the timer to prevent a user program from running too long. **A** simple technique is to initialize a counter with the amount of time that a program is allowed to run. A program with a 7-minute time limit, for example, would have its counter initialized to 420. Every second, the timer interrupts and the counter is decremented by 1. As long as the counter is positive, control is returned to the user program. When the counter becomes negative, the operating system terminates the program for exceeding the assigned time limit.

A more common use of a timer is to implement time sharing. In the most straightforward case, the timer could be set to interrupt every N milliseconds, where N is the time slice that each user is allowed to execute before the next user gets control of the CPU. The operating system is invoked at the end of each time slice to perform various housekeeping tasks, such as adding the value *N* to the record that specifies (for accounting purposes) the amount of time the user program has executed thus far. The operating system also saves registers, internal variables, and buffers, and changes several other parameters to prepare for the next program to run. This procedure is known as a context switch. Following a context switch, the next program continues with its execution from the point at which it left off (when its previous time slice ran out).

Another use of the timer is to compute the current time. A timer interrupt signals the passage of some period, allowing the operating system to compute the current time in reference to some initial time.