Operating Systems
Lecture No. 3

Reading Material

- Computer System Structures, Chapter 2
- Operating Systems Structures, Chapter 3
- PowerPoint Slides for Lecture 3

Summary

- Memory and CPU protection
- Operating system components and services
- System calls
- Operating system structures

Memory Protection

The region in the memory that a process is allowed to access is known as **process address space**. To ensure correct operation of a computer system, we need to ensure that a process cannot access memory outside its address space. If we don't do this then a process may, accidentally or deliberately, overwrite the address space of another process or memory space belonging to the operating system (e.g., for the interrupt vector table).

Using two CPU registers, specifically designed for this purpose, can provide memory protection. These registered are:

- Base register – it holds the smallest legal physical memory address for a process
- Limit register – it contains the size of the process

When a process is loaded into memory, the base register is initialized with the starting address of the process and the limit register is initialized with its size. Memory outside the defined range is protected because the CPU checks that every address generated by the process falls within the memory range defined by the values stored in the base and limit registers, as shown in Figure 3.1.

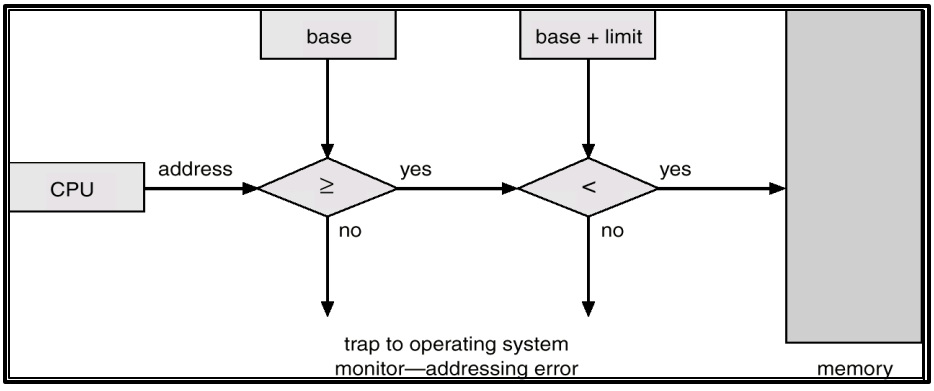

Figure 3.1 Hardware address protection with base and limit registers

In Figure 3.2, we use an example to illustrate how the concept outlined above works. The base and limit registers are initialized to define the address space of a process. The process starts at memory location 300040 and its size is 120900 bytes (assuming that memory is byte addressable). During the execution of this process, the CPU insures (by using the logic outlined in Figure 3.1) that all the addresses generated by this process are greater than or equal to 300040 and less than (300040+120900), thereby preventing this process to access any memory area outside its address space. Loading the base and limit registers are privileged instructions.

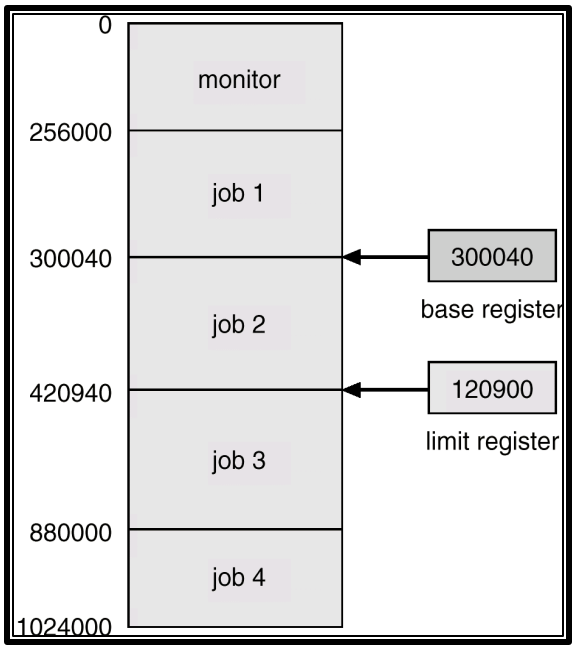

Figure 3.2 Use of Base and Limit Register

CPU Protection

In addition to protecting I/O and memory, we must ensure that the operating system maintains control. We must prevent the user program from getting stuck in an infinite loop or not calling system services and never returning control to the CPU. To accomplish this we can use a **timer**, which interrupts the CPU after specified period to ensure that the operating system maintains control. The timer period may be variable or fixed. A *fixed-rate clock* and a *counter* are used to implement a variable timer. The OS initializes the counter with a positive value. The counter is decremented every clock tick by the clock interrupt service routine. When the counter reaches the value 0, a timer interrupt is generated that transfers control from the current process to the next scheduled process. Thus we can use the timer to prevent a program from running too long. In the most straight forward case, the timer could be set to interrupt every N milliseconds, where N is the **time slice** that each process is allowed to execute before the next process gets control of the CPU. The OS is invoked at the end of each time slice to perform various housekeeping tasks. This issue is discussed in detail under CPU scheduling in Chapter 7.

Another use of the timer is to compute the current time. A timer interrupt signals the passage of some period, allowing the OS to compute the current time in reference to some initial time. Load-timer is a privileged instruction.

OS Components

An operating system has many components that manage all the resources in a computer system, insuring proper execution of programs. We briefly describe these components in this section.

✍ Process management

A process can be thought of as a program in execution. It needs certain resources, including CPU time, memory, files and I/O devices to accomplish its tasks. The operating system is responsible for:

- ✍ Creating and terminating both user and system processes
- ✍ Suspending and resuming processes
- ✍ Providing mechanisms for process synchronization
- ✍ Providing mechanisms for process communication
- ✍ Providing mechanisms for deadlock handling

✍ Main memory management

Main memory is a large array of words or bytes (called memory locations), ranging in size from hundreds of thousands to billions. Every word or byte has its own address. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. It contains the code, data, stack, and other parts of a process. The central processor reads instructions of a process from main memory during the machine cycle—fetch-decode-execute.

The OS is responsible for the following activities in connection with memory management:

- ✍ Keeping track of free memory space
- ✍ Keeping track of which parts of memory are currently being used and by whom
- ✍ Deciding which processes are to be loaded into memory when memory space becomes available
- ✍ Deciding how much memory is to be allocated to a process
- ✍ Allocating and deallocating memory space as needed
- ✍ Insuring that a process is not overwritten on top of another

✍ Secondary storage management

The main purpose of a computer system is to execute programs. The programs, along with the data they access, must be in the main memory or **primary storage** during their execution. Since main memory is too small to accommodate all data and programs, and because the data it holds are lost when the power is lost, the computer system must provide **secondary storage** to backup main memory. Most programs are stored on a disk until loaded into the memory and then use disk as both the source and destination of their processing. Like all other resources in a computer system, proper management of disk storage is important.

The operating system is responsible for the following activities in connection with disk management:

- ✍ Free-space management

- ✍ Storage allocation and deallocation
- ✍ Disk scheduling

✍ I/O system management

The I/O subsystem consists of:

- ✍ A memory management component that includes buffering, caching and spooling
- ✍ A general device-driver interface
- ✍ Drivers for specific hardware devices

✍ File management

Computers can store information on several types of physical media, e.g. magnetic tape, magnetic disk and optical disk. The OS maps files onto physical media and accesses these media via the storage devices.

The OS is responsible for the following activities with respect to file management:

- ✍ Creating and deleting files
- ✍ Creating and deleting directories
- ✍ Supporting primitives (operations) for manipulating files and directories
- ✍ Mapping files onto the secondary storage
- ✍ Backing up files on stable (nonvolatile) storage media

✍ Protection system

If a computer system has multiple users and allows concurrent execution of multiple processes then the various processes must be protected from each other's activities.

Protection is any mechanism for controlling the access of programs, processes or users to the resources defined by a computer system.

✍ Networking

A **distributed system** is a collection of processors that do not share memory, peripheral devices or a clock. Instead, each processor has it own local memory and clock, and the processors communicate with each other through various communication lines, such as high- speed buses or networks.

The processors in a communication system are connected through a **communication network**. The communication network design must consider message routing and connection strategies and the problems of contention and security.

A distributed system collects physically separate, possibly heterogeneous, systems into a single coherent system, providing the user with access to the various resources that the system maintains.

✍ Command-line interpreter (shells)

One of the most important system programs for an operating system is the **command interpreter**, which is the interface between the user and operating system. Its purpose is to read user commands and try to execute them. Some operating systems include the command interpreter in the kernel. Other operating systems (e.g. UNIX, Linux, and DOS) treat it as a special program that runs when a job is initiated or when a user first logs on (on time sharing systems). This program is sometimes called the **command-line interpreter** and is often known as the **shell**. Its function is simple: to get the next command statement and execute it. Some of the famous shells for UNIX and Linux are

Bourne shell (sh), C shell (csh), Bourne Again shell (bash), TC shell (tcsh), and Korn shell (ksh). You can use any of these shells by running the corresponding command, listed in parentheses for each shell. So, you can run the Bourne Again shell by running the `bash` or `/usr/bin/bash` command.

Operating System Services

An operating system provides the environment within which programs are executed. It provides certain services to programs and users of those programs, which vary from operating system to operating system. Some of the common ones are:

- **Program execution:** The system must be able to load a program into memory and to run that programs. The program must be able to end its execution.
- **I/O Operations:** A running program may require I/O, which may involve a file or an I/O device. For efficiency and protection user usually cannot control I/O devices directly. The OS provides a means to do I/O.
- **File System Manipulation:** Programs need to read, write files. Also they should be able to create and delete files by name.
- **Communications:** There are cases in which one program needs to exchange information with another process. This can occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communication may be implemented via shared **memory** or **message passing**.
- **Error detection:** The OS constantly needs to be aware of possible errors. Error may occur in the CPU and memory hardware, in I/O devices and in the user program. For each type of error, the OS should take appropriate action to ensure correct and consistent computing.

In order to assist the efficient operation of the system itself, the system provides the following functions:

- **Resource allocation:** When multiple users are logged on the system or multiple jobs are running at the same time, resources must be allocated to each of them. There are various routines to schedule jobs, allocate plotters, modems and other peripheral devices.
- **Accounting:** We want to keep track of which users use how many and which kinds of computer resources. This record keeping may be used for accounting or simply for accumulating usage statistics.
- **Protection:** The owners of information stored in a multi user computer system may want to control use of that information. When several disjointed processes execute concurrently it should not b possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled.

Entry Points into Kernel

As shown in Figure 3.3, there are four events that cause execution of a piece of code in the kernel. These events are: interrupt, trap, system call, and signal. In case of all of these events, some kernel code is executed to service the corresponding event. You have

discussed interrupts and traps in the computer organization or computer architecture course. We will discuss system calls execution in this lecture and signals subsequent lectures. We will talk about many UNIX and Linux system calls and signals throughout the course.

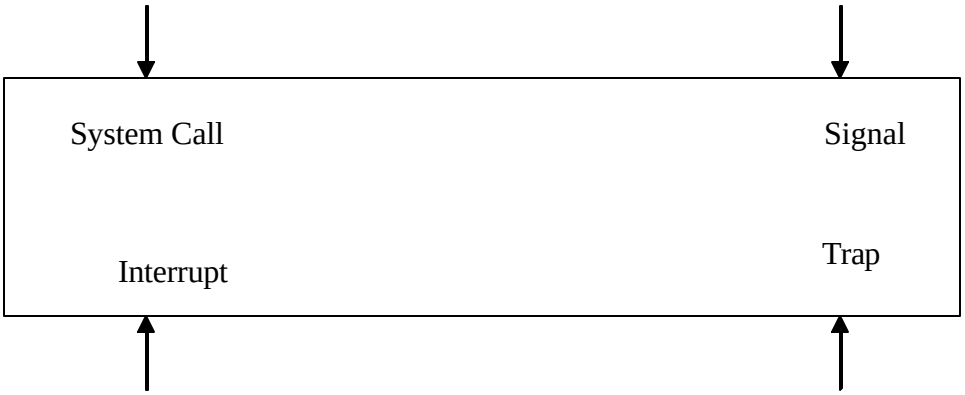

Figure 3.3 Entry points into the operating system kernel

System Calls

System calls provide the interface between a process and the OS. These calls are generally available as assembly language instructions. The system call interface layer contains entry point in the kernel code; because all system resources are managed by the kernel any user or application request that involves access to any system resource must be handled by the kernel code, but user process must not be given open access to the kernel code for security reasons. So that user processes can invoke the execution of kernel code, several openings into the kernel code, also called *system calls*, are provided. System calls allow processes and users to manipulate system resources such as files and processes.

System calls can be categorized into the following groups:

- Process Control
- File Management
- Device Management
- Information maintenance
- Communications

Semantics of System Call Execution

The following sequence of events takes place when a process invokes a system call:

- The user process makes a call to a library function
- The library routine puts appropriate parameters at a well-known place, like a register or on the stack. These parameters include arguments for the system call, return address, and call number. Three general methods are used to pass parameters between a running program and the operating system.
  - Pass parameters in *registers*.
  - Store the parameters in a table in the main memory and the table address is passed as a parameter in a register.
  - *Push* (store) the parameters onto the *stack* by the program, and *pop* off the stack by operating system.

- A `trap` instruction is executed to change mode from user to kernel and give control to operating system.
- The operating system then determines which system call is to be carried out by examining one of the parameters (the call number) passed to it by library routine.
- The kernel uses call number to index a kernel table (the **dispatch table)** which contains pointers to service routines for all system calls.
- The service routine is executed and control given back to user program via return from trap instruction; the instruction also changes mode from system to user.
- The library function executes the instruction following trap; interprets the return values from the kernel and returns to the user process.

Figure 3.4 gives a pictorial view of the above steps.

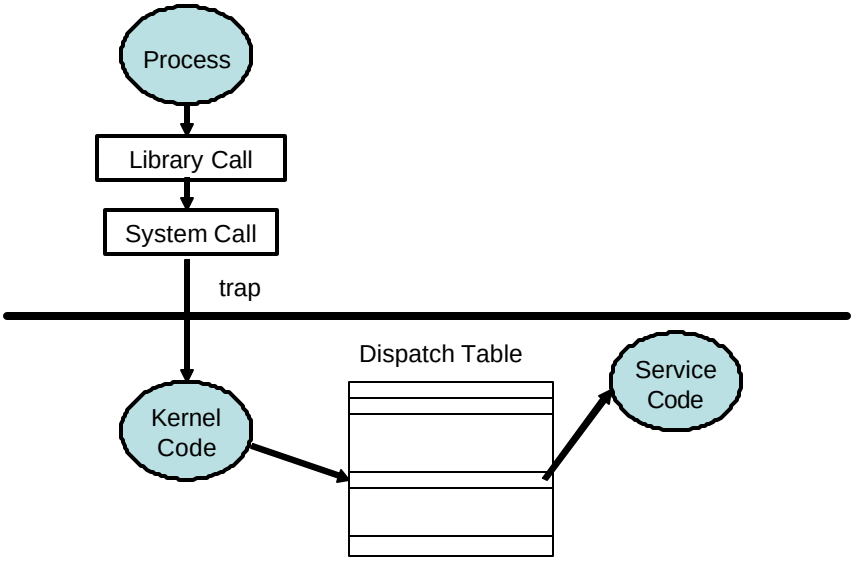

Figure 3.4 Pictorial view of the steps needed for execution of a system call

Operating Systems Structures

Just like any other software, the operating system code can be structured in different ways. The following are some of the commonly used structures.

- Simple/Monolithic Structure

In this case, the operating system code has not structure. It is written for functionality and efficiency (in terms of time and space). DOS and UNIX are examples of such systems, as shown in Figures 3.5 and 3.6. UNIX consists of two separable parts, the kernel and the system programs. The kernel is further separated into a series of interfaces and devices drivers, which were added and expanded over the years. Every thing below the system call interface and above the physical hardware is the kernel, which provides the file system, CPU scheduling, memory management and other OS functions through system calls. Since this is an enormous amount of functionality combined in one level, UNIX is difficult to enhance as changes in one section could adversely affect other areas. We will discuss the various components of the UNIX kernel throughout the course.

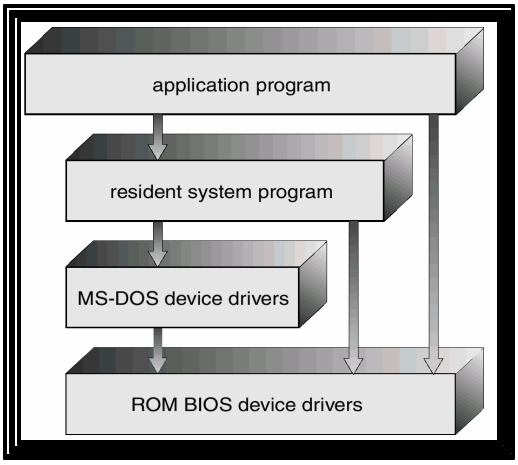

Figure 3.5 Logical structure of DOS

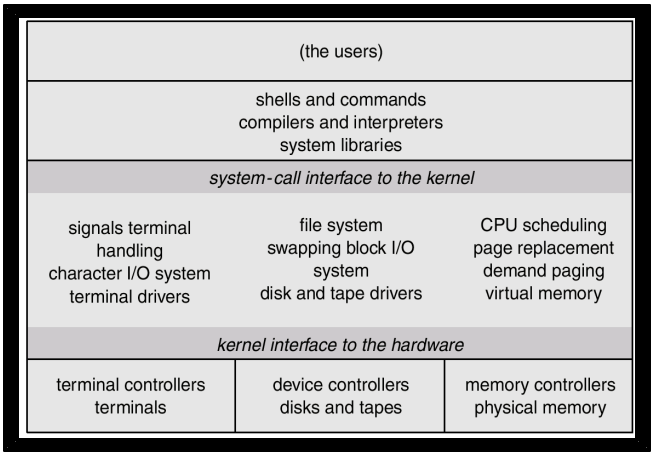

Figure 3.6 Logical structure of UNIX