CHAPTER 2 – OPERATING-SYSTEMS STRUCTURES

SERVICES

One set of operating system services provides functions that are helpful to the user.

- User interface. command-line interface (CLI), which uses text commands. Another is a batch interface, in which commands and directives to control those commands are entered into files, and those files are executed. Graphical user interface (GUI) where the interface uses windows.
- Program execution. The system must be able to load a program into memory and to run that program.
- I/O operations. A running program may require I/O, which may involve a file or an I/O device.
- File-system manipulation. Programs need to read and write files and directories.
- Communications. Communications may be implemented via shared memory, in which two or more processes read and write to a shared section of memory, or message passing, in which packets of information in predefined formats are moved between processes by the operating system.
- Error detection. The operating system needs to be detecting and correcting errors constantly.

System functions exists not for helping the user but rather for ensuring the efficient operation of the system itself

- Resource allocation. When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them (such as CPU cycles, main memory, and file storage).
- Accounting. keep track of which users use how much and what kinds of computer resources.
- Protection and security. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important (authentication).

USER & OS

Command Interpreters

Some operating systems include the command interpreter in the kernel. Others, such as Windows and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on (on interactive systems). On systems with multiple command interpreters to choose from, the interpreters are known as shells.

Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. The MS-DOS and UNIX shells operate in this way.

SYSTEM CALLS

Provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++ or assembly-language instructions.

Why would an application programmer prefer programming according to an API rather than invoking actual system calls? There are several reasons for doing so. One benefit concerns program portability.

For most programming languages, the run-time support system (a set of functions built into libraries included with a compiler) provides a systemcall interface that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system. Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers.

The system call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call and any return values. Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time support library.

Three general methods are used to pass parameters to the operating system:

- The simplest approach is to pass the parameters in registers.
- When there are more parameters than registers. In these cases, the parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register (Linux and Solaris).
- Parameters also can be placed, or pushed, onto the stack by the program and popped off the stack by the operating system

SYSTEM CALLS TYPES

❖ Process Control

Process control ◦ end, abort ◦ load, execute ◦ create process, terminate process

A running program needs to be able to halt its execution either normally (end()) or abnormally (abort()).

If the program discovers an error in its input and wants to terminate abnormally, it may also want to define an error level. More severe errors can be indicated by a higher-level error parameter. The command interpreter or a following program can use this error level to determine the next action automatically.

Quite often, two or more processes may share data. To ensure the integrity of the data being shared, operating systems often provide system calls allowing a process to lock shared data. Then, no other process can access the data until the lock is released.

The MS-DOS operating system is an example of a single-tasking system. Because MS-DOS is single-tasking, it uses a simple method to run a program and does not create a new process. It loads the program into memory, writing over most of itself to give the program as much memory as possible Since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed. To start a new process, the Shell executes a fork() system call. Then, the selected program is loaded into memory via an exec() system call, and the program is executed. When the process is done, it executes an exit() system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code. This status or error code is then available to the shell or other programs. Processes are discussed in Chapter 3 with a program example using the fork() and exec() system calls.

❖ File Management

File management ◦ create file, delete file ◦ open, close ◦ read, write

❖ Device Management

Device management ◦ request device, release device ◦ read, write, reposition

A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available

❖ Information maintenance

Information maintenance ◦ get time or date, set time or date ◦ get system data, set system data ◦ get process, file, or device attributes ◦ set process, file, or device attributes

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. Another set of system calls is helpful in debugging a program.

Many operating systems provide a time profile of a program to indicate the amount of time that the program executes at a particular location or set of locations. A time profile requires either a tracing facility or regular timer interrupts. At every occurrence of the timer interrupt, the value of the program counter is recorded. With sufficiently frequent timer interrupts, a statistical picture of the time spent on various parts of the program can be obtained.

❖ Communications

Communications ◦ create, delete communication connection ◦ send, receive messages ◦ transfer status information ◦ attach or detach remote devices

There are two common models of interprocess communication:

- Message-passing model: the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. Each process has a process name, and this name is translated into an identifier by which the operating system can refer to the process. Most processes that will be receiving connections are special-purpose daemons, which are system programs provided for that purpose.
- Shared-memory model: processes use shared memory create() and shared memory attach() system calls to create and gain access to regions of memory owned by other processes. The operating system tries to prevent one process from accessing another process's memory. Processes can exchange information by reading and writing data in the shared areas

Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. It is also easier to implement than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication.

❖ Protection

Protection provides a mechanism for controlling access to the resources provided by a computer system. Historically, protection was a concern only on multiprogrammed computer systems with several users.

SYSTEM PROGRAMS

The logical computer hierarchy: At the lowest level is hardware. Next is the operating system, then the system programs, and finally the application programs.

System programs, also known as system utilities, provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls. They can be divided into these categories:

- File management. These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- File modification. Several text editors may be available to create and modify the content of files stored on disk or other storage devices.
- Programming-language support. Compilers, assemblers, debuggers, and interpreters.
- Program loading and execution: Loaders, relocatable loaders, linkage editors, debugging systems.
- Communications
- Background services. Some processes terminate after completing their tasks, while others continue to run until the system is halted. Constantly running system-program processes are known as services, subsystems, or daemons.

DESIGN & GOALS

The design of the system will be affected by the choice of hardware and the type of system: batch, time sharing, single user, multiuser, distributed, real time, or general purpose.

Beyond this highest design level, the requirements may be much harder to specify. The requirements can, however, be divided into two basic groups: user goals and system goals.

MECHANISMS & POLICIES

Mechanisms determine how to do something; policies determine what will be done. Policies are likely to change across places or over time. Microkernel-based operating systems take the separation of mechanism and policy to one extreme by implementing a basic set of primitive building blocks. These blocks are almost policy free, allowing more advanced mechanisms and policies to be added via user-created kernel modules or user programs themselves.

Whenever it is necessary to decide whether or not to allocate a resource, a policy decision must be made. Whenever the question is how rather than what, it is a mechanism that must be determined.

IMPLEMENTATION

The advantages of using a higher-level language: the code can be written faster, is more compact, and is easier to understand and debug. In addition, improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation. Finally, an operating system is far easier to port. The only possible disadvantages are reduced speed and increased storage requirements.

SYSTEM STRUCTURE

SIMPLE

Systems that started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. Like MS-DOS, UNIX initially was limited by hardware functionality. It consists of two separable parts: the kernel and the system programs. Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.

LAYERED APPROACH

A system can be made modular in many ways. One method is the layered approach, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.

A typical operating-system layer—say, layer M—consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M, in turn, can invoke operations on lower-level layers. The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers.

The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary. A final problem with layered implementations is that they tend to be less efficient than other types. At each layer, the parameters may be modified, data may need to be passed, and so on. Each layer adds overhead to the system call. The net result is a system call that takes longer than does one on a nonlayered system.

MICROKERNELS

This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel. The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space. The resulting operating system is easier to port. The microkernel also provides more security and reliability, since most services are running as user—rather than kernel— processes. If a service fails, the rest of the operating system remains untouched. Unfortunately, the performance of microkernels can suffer due to increased system-function overhead.

MODULES

The kernel has a set of core components and links in additional services via modules, either at boot time or during run time. This type of design is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X, as well as Windows. The idea of the design is for the kernel to provide

core services while other services are implemented dynamically, as the kernel is running. Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made.

HYBRIDS

In practice, very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues.

DEBBUGGING

FAILURE ANALYSIS

If a process fails, most operating systems write the error information to a log file to alert system operators or users that the problem occurred. The operating system can also take a core dump—a capture of the memory of the process— and store it in a file for later analysis. A failure in the kernel is called a crash.

PERFORMANCE TUNNING

Performance tuning seeks to improve performance by removing processing bottlenecks. To identify bottlenecks, we must be able to monitor system performance. All interesting events are logged with their time and important parameters and are written to a file. Later, an analysis program can process the log file to determine system performance and to identify bottlenecks and inefficiencies.

DTRACE

DTrace is a facility that dynamically adds probes to a running system, both in user processes and in the kernel. These probes can be queried via the D programming language to determine an astonishing amount about the kernel, the system state, and process activities

The DTrace provides a dynamic, safe, low-impact debugging environment. DTrace runs on production systems—systems that are running important or critical applications—and causes no harm to the system. It slows activities while enabled, but after execution it resets the system to its pre-debugging state. It is also a broad and deep tool. It can broadly debug everything happening in the system. DTrace features a compiler that generates a byte code that is run in the kernel. This code is assured to be "safe" by the compiler. Only users with DTrace "privileges" (or "root" users) are allowed to use DTrace, as it can retrieve private kernel data.

GENERATION

The system must be configured or generated for each specific computer site, a process sometimes known as system generation SYSGEN. This SYSGEN program reads from a given file, or asks the operator of the system for information concerning the specific configuration of the hardware system, or probes the hardware directly to determine what components are there. The following kinds of information must be determined.

System generation involves simply creating the appropriate tables to describe the system. The major differences among these approaches are so the size and generality of the generated system and the ease of modifying it as the hardware configuration changes.

BOOT

The procedure of starting a computer by loading the kernel is known as booting the system. A small piece of code known as the bootstrap program or bootstrap loader locates the kernel, loads it into main memory, and starts its execution. The program is in the form of read-only memory (ROM). ROM is convenient because it needs no initialization. Usually, one task is to run diagnostics to determine the state of the machine.

Some systems—such as cellular phones, tablets, and game consoles—store the entire operating system in ROM. Storing the operating system in ROM is suitable for small operating systems, simple supporting hardware. A problem with this approach is that changing the bootstrap code requires changing the ROM hardware chips. Some systems resolve this problem by using erasable programmable read-only memory (EPROM).

All forms of ROM are also known as firmware, since their characteristics fall somewhere between those of hardware and those of software. A problem with firmware in general is that executing code there is slower than executing code in RAM. Some systems store the operating system in firmware and copy it to RAM for fast execution. A final issue with firmware is that it is relatively expensive, so usually only small amounts are available. For large operating systems or for systems that change frequently, the bootstrap loader is stored in firmware, and the operating system is on disk.

GRUB is an example of an open-source bootstrap program for Linux systems.

** **NOTES**

How are iOS and Android similar? How are they different?

They are similar in that both are based on existing kernels. Both allow developers to have a framework. Both use stacks. They are different in that IOS is closed-source while Android is open-source. Android uses a virtual machine unlike IOS. Androids applications are coded in Java while IOS applications are coded in Objective-C.