KTU B.TECH CSE S4 NOTE
OPERATING SYSTEMS
CS204
MODULE - 1

By

Ms Jasheeda P

CSE Department

MEA Engineering College Peinthalnanna

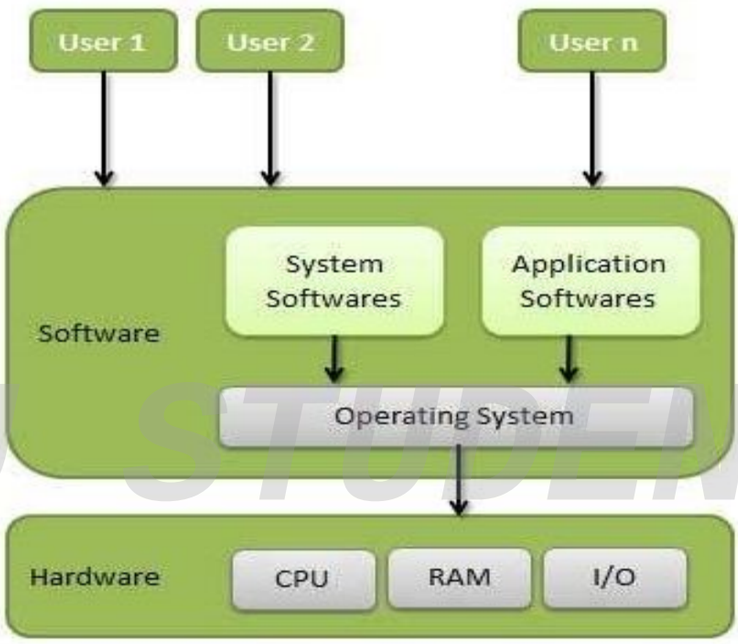
1.1 Introduction: Functions of Operating

An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

Some popular Operating Systems include Linux, Windows, OS X, VMS, OS/400, AIX, z/OS, etc.

Definition

An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.



Following are some of important functions of an operating System.

- Memory Management
- Processor Management
- Device Management
- File Management
- Security
- Control over system performance
- Job accounting
- Error detecting aids
- Coordination between other software and users

Memory Management

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.

Main memory provides a fast storage that can be accessed directly by the CPU. For a program to be executed, it must in the main memory. An Operating System does the following activities for memory management –

- Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part are not in use.
- In multiprogramming, the OS decides which process will get memory when and how much.
- Allocates the memory when a process requests it to do so.
- De-allocates the memory when a process no longer needs it or has been terminated.

Processor Management

In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called process scheduling. An Operating System does the following activities for processor management –

- Keeps tracks of processor and status of process. The program responsible for this task is known as traffic controller.
- Allocates the processor (CPU) to a process.
- De-allocates processor when a process is no longer required.

Device Management

An Operating System manages device communication via their respective drivers. It does the following activities for device management –

- Keeps tracks of all devices. Program responsible for this task is known as the I/O controller.

Decides which process gets the device when and for how much time.

- Allocates the device in the efficient way.
- De-allocates devices.

File Management

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

An Operating System does the following activities for file management –

- Keeps track of information, location, uses, status etc. The collective facilities are often known as file system.
- Decides who gets the resources.
- Allocates the resources.

- De-allocates the resources.

Other Important Activities

Following are some of the important activities that an Operating System performs –

- **Security** – By means of password and similar other techniques, it prevents unauthorized access to programs and data.
- **Control over system performance** – Recording delays between request for a service and response from the system.
- **Job accounting** – Keeping track of time and resources used by various jobs and users.
- **Error detecting aids** – Production of dumps, traces, error messages, and other debugging and error detecting aids.
- **Coordination between other softwares and users** – Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

1.2 Single processor, multiprocessor and clustered systems - Overview

Computer architecture means design or construction of a computer. A computer system may be organized in different ways. Some computer systems have single processor and other have multiprocessors. So computer systems categorized in these ways.

1. Single Processor Systems
2. Multiprocessor Systems
3. Clustered Systems

Single Processor

Some computers use only one processor such as microcomputers. On a single processor system, there is only one CPU that perform all the activities in the computer system, However, most of these systems have other special purpose processors, such as I/O Processor that move data rapidly among different components of the computer system. These processors execute only a limited system programs and do not run the user program. So we define single processor systems as " A system that has only one general purpose CPU, is considered as single processor system.

Multiprocessor systems

Some systems have two or more processors. These systems are also known as parallel systems or tightly coupled systems. Mostly the processors of

these systems share the common system bus (electronic path) memory and peripheral (input/output) devices. These systems are fast in data processing and have capability to execute more than one program simultaneously on different processors. This type of processing is known as multiprogramming or multiprocessing. Multiple processors further divided in two types.

i. Asymmetric Multiprocessing Systems (AMS)

ii. Symmetric Multiprocessing Systems (SYS)

Asymmetric multiprocessing systems

The multiprocessing system, in which each processor is assigned a specific task, is known as Asymmetric Multiprocessing Systems. In this system there exists master slave relationship like one processor defined as master and others are slave. The master processor controls the system and other processor executes predefined tasks. The master processor also defined the task of slave processors.

Symmetric multiprocessing system

The multiprocessing system in each processor performs all types of task within the operating system. All processors are peers and no master slave relationship exits. In SMP systems many programs can run simultaneously. But I/O must control to ensure that data reach the appropriate processor because all the processor shares the same memory.

Clustered systems

Clustered systems are another form of multiprocessor system. This system also contains multiple processors but it differs from multiprocessor system. The clustered system is composed of multiple individual systems that connected together. In clustered system, also individual systems or computers share the same storage and liked to gather via local area network. A special type of software is known as cluster to control the node the systems.

Other form of clustered system includes parallel clusters and clustering over a wide area network. In parallel cluster multiple hosts can access the same data on the shared storage. So many operating systems provide this facility but some special software is are also designed to run on a parallel cluster to share data.

1.3 Kernel Data Structures – Operating Systems used in different computing environments.

Kernel Data Structures

The operating system must keep a lot of information about the current state of the system. As things happen within the system these data structures must be changed to reflect the current reality. For example, a new process might be created when a user logs onto the system. The kernel must create a data structure representing the new process and link it with the data structures representing all of the other processes in the system.

Mostly these data structures exist in physical memory and are accessible only by the kernel and its subsystems. Data structures contain data and pointers, addresses of other data structures, or the addresses of routines. Taken all together, the data structures used by the Linux kernel can look very confusing. Every data structure has a purpose and although some are used by several kernel subsystems, they are simpler than they appear at first sight.

Understanding the Linux kernel hinges on understanding its data structures and the use that the various functions within the Linux kernel makes of them. This section bases its description of the Linux kernel on its data structures. It talks about each kernel subsystem in terms of its algorithms, which are its methods of getting things done, and their usage of the kernel's data structures.

Linked Lists

Linux uses a number of software engineering techniques to link together its data structures. On a lot of occasions it uses linked or chained data structures. If each data structure describes a single instance or occurrence of something, for example a process or a network device, the kernel must be able to find all of the instances. In a linked list a root pointer contains the address of the first data structure, or element, in the list, and then each subsequent data structure contains a pointer to the next element in the list. The last element's next pointer would be 0 or NULL to show that it is the end of the list. In a doubly linked list each element contains both a pointer to the next element in the list but also a pointer to the previous element in the list. Using doubly linked lists makes it easier to add or remove elements from the middle of list, although you do need more memory accesses. This is a typical operating system trade off: memory accesses versus CPU cycles.

Hash Tables

Linked lists are handy ways of tying data structures together, but navigating linked lists can be inefficient. If you were searching for a particular element, you might easily have to look at the whole list before you find the one that you need. Linux uses another technique, hashing, to get around this restriction. A hash table is an array or vector of pointers. An array, or vector, is simply a set of things coming one after another in memory. A bookshelf could be said to be an array of books. Arrays are accessed by an index, which is an offset into the array's associated area in memory. Taking the bookshelf analogy a little further, you could describe each book by its position on the shelf; you might ask for the 5th book.

A hash table is an array of pointers to data structures and its index is derived from information in those data structures. If you had data structures describing the population of a village then you could use a person's age as an index. To find a particular person's data you could use their age as an index into the population hash table and then follow the pointer to the data

structure containing the person's details. Unfortunately many people in the village are likely to have the same age and so the hash table pointer becomes a pointer to a chain or list of data structures each describing people of the same age. However, searching these shorter chains is still faster than searching all of the data structures.

As a hash table speeds up access to commonly used data structures, Linux often uses hash tables to implement caches. Caches are handy information that needs to be accessed quickly and are usually a subset of the full set of information available. Data structures are put into a cache and kept there because the kernel often accesses them. The drawback to caches is that they are more complex to use and maintain than simple linked lists or hash tables. If the data structure can be found in the cache (this is known as a cache hit), then all well and good. If it cannot then all of the relevant data structures must be searched and, if the data structure exists at all, it must be added into the cache. In adding new data structures into the cache an old cache entry may need discarding. Linux must decide which one to discard, the danger being that the discarded data structure may be the next one that Linux needs.

Abstract Interfaces

The Linux kernel often abstracts its interfaces. An interface is a collection of routines and data structures which operate in a well-defined way. For example all, network device drivers have to provide certain routines to operate on particular data structures. This way there can be generic layers of code using the services (interfaces) of lower layers of specific code. The network layer is generic and it is supported by device specific code that conforms to a standard interface.

Often these lower layers *register* themselves with the upper layer at boot time. This registration usually involves adding a data structure to a linked list. For example each filesystem built into the kernel registers itself with the kernel at boot time or, if you are using modules, when the filesystem is first used. You can see which filesystems have registered themselves by looking at the file/proc/filesystems.

The registration data structure often includes pointers to functions. These are the addresses of software functions that perform particular tasks. Again, using filesystem registration as an example, the data structure that each filesystem passes to the Linux kernel as it registers includes the address of a filesystem specfic routine which must be called whenever that filesystem is mounted.

Computing Environments

Traditional

Stand-alone general purpose machines

But blurred as most systems interconnect with others (i.e., the Internet)

Portals provide web access to internal systems

Network computers (thin clients) are like Web terminals

Mobile computers interconnect via wireless networks

Networking becoming ubiquitous – even home systems uses firewalls to

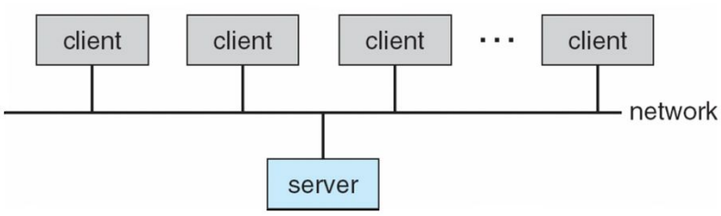
protect home computers from Internet attacks.

Client-Server Computing

Many of today's systems act as server system to satisfy requests generated by client systems. This form of specialized distributed system, called a client-server system.

Compute-server system provides an interface to client to request services (i.e., database)

File-server system provides interface for clients to store and retrieve files



Peer-to-Peer Computing

Another model of distributed system

P2P does not distinguish clients and servers

Instead all nodes are considered peers

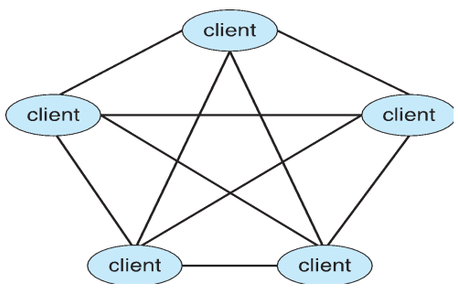
May each act as client, server or both

Node must join P2P network

Registers its service with central lookup service on network, or

Broadcast request for service and respond to requests for service via discovery protocol

Examples include Napster and Gnutella, Voice over IP (VoIP) such as Skype



Web-Based Computing

The Web has become ubiquitous, leading to more access by a wider variety of device including PCs, workstations, handheld PDAs1 and even cell phones as access point.

Devices that were networked now have faster network connectivity, provided by either improved networking technology, optimized network

implementation code, or both.

The implementation of Web-based computing has given rise to new categories of devices, such as load balancers which distribute network connections among a pool of similar servers.

Open-Source Operating Systems

Open Source operating systems are those made available in source-code format rather than as compiled binary code.

Linux is the most famous open- source operating system, while Microsoft Windows is a well-known example of the opposite closed source approach.

There are many benefits to open-source operating systems/ including a community of interested (and usually unpaid) programmers who contribute to the code by helping to debug it analyze it/ provide support/ and suggest changes.

Open-source code is more secure than closed-source code because many more eyes are viewing the code.

History

In the early days of modern computing (that is, the 1950s), a great deal of software was available in open-source format.

Counter to the copy protection and Digital Rights Management (DRM) movement

Started by Free Software Foundation (FSF), which has "copy left" GNU Public License (GPL)

Examples include GNU/Linux, BSD UNIX (including core of Mac OS X), and Sun Solaris

Linux

As an example of an open-source operating system, consider GNU/Linux.

The GNU project produced many UNIX-compatible tools, including compilers, editors, and utilities, but never released a kernel.

The resulting GNU /Linux operating system has spawned hundreds of unique or custom builds, of the system. Major distributions include RedHat, SUSE, Fedora, Debian, Slackware, and Ubuntu.

BSD UNIX

It started in 1978 as a derivative of AT&T's UNIX.

There are many distributions of BSD UNIX, including FreeBSD, NetBSD, OpenBSD, and DragonflyBSD.

Solaris

Solaris is the commercial UNIX-based operating system. of Sun

Microsystems.

Originally, Sun perating system was based on BSD UNIX.

Solaris can be compiled from the open source and linked with binaries of the close-sourced components, so it can still be explored, modified, compiled, and tested.

Utility

Open-source projects enable students to use source code as a learning tool.

GNU ILinux, BSD UNIX, and Solaris are all open-source operating systems, but each has its own goals, utility, licensing, and purpose.

The advantages of free software and open sourcing are likely to increase the number and quality of open-source projects, leading to an increase in the number of individuals and companies that use these projects.

1.4 Operating System Interfaces and implementation - Approaches

User Operating System Interface

Command Interpreter

- CLI or command interpreter allows direct command entry
  - ➢ Sometimes implemented in kernel, sometimes by systems program
  - ➢ Sometimes multiple flavors implemented – shells
  - ➢ Primarily fetches a command from user and executes it
  - ➢ Sometimes commands built-in, sometimes just names of programs
    - If the latter, adding new features doesn't require shell modification

GUI

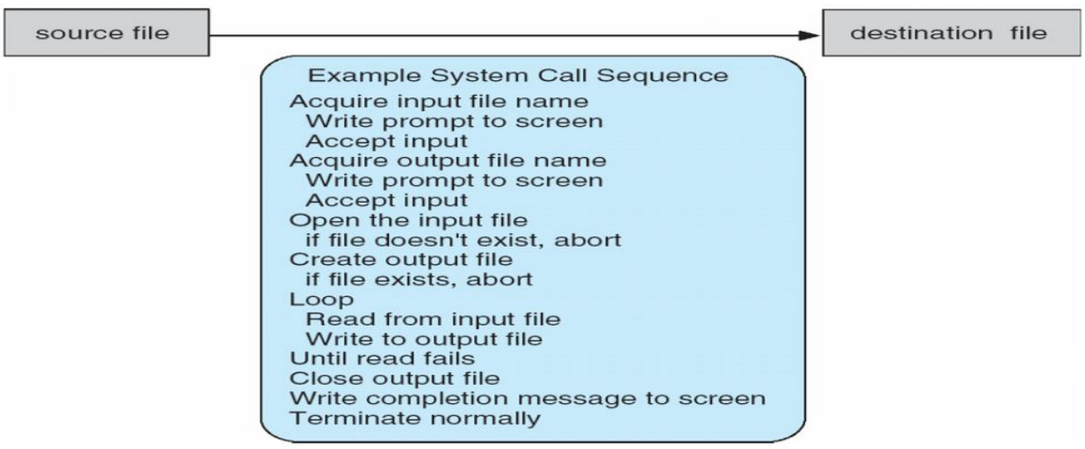
- ➢ User-friendly desktop metaphor interface
- ➢ Usually mouse, keyboard, and monitor
- ➢ Icons represent files, programs, actions, etc
- ➢ Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)
- ➢ Invented at Xerox PARC
- ➢ Many systems now include both CLI and GUI interfaces
- ➢ Microsoft Windows is GUI with CLI "command" shell
- ➢ Apple Mac OS X is "Aqua" GUI interface with UNIX kernel underneath and shells available
- ➢ Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Example of System Calls

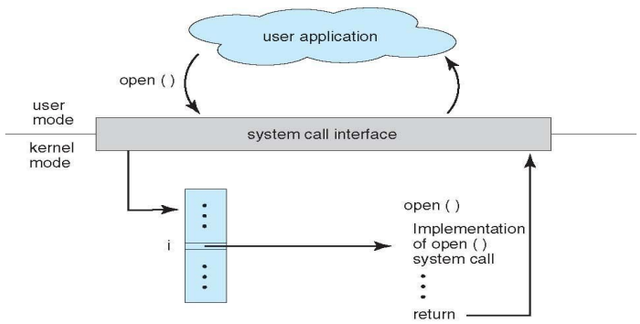
- System call sequence to copy the contents of one file to another file



System Call Implementation

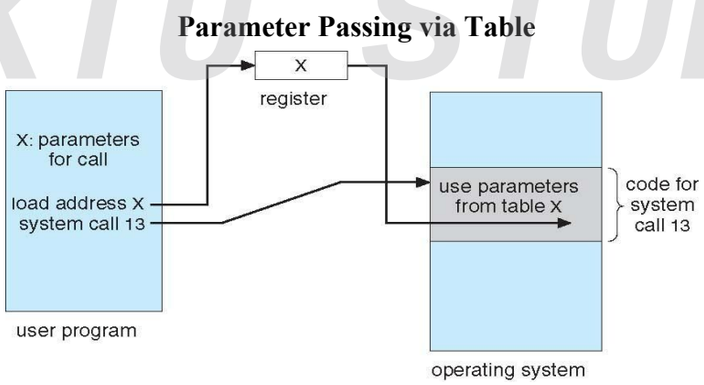
- Typically, a number associated with each system call
- System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
- Just needs to obey API and understand what OS will do as a result call
- Most details of OS interface hidden from programmer by API
- Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
- Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
- Simplest: pass the parameters in registers
- In some cases, may be more parameters than registers
- Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
- This approach taken by Linux and Solaris
- Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
- Block and stack methods do not limit the number or length of parameters being passed



Types of System Calls

- Process control
  - ➢ create process, terminate process
  - ➢ end, abort
  - ➢ load, execute
  - ➢ get process attributes, set process attributes
  - ➢ wait for time
  - ➢ wait event, signal event
  - ➢ allocate and free memory
  - ➢ Dump memory if error
  - ➢ Debugger for determining bugs, single step execution
  - ➢ Locks for managing access to shared data between processes
- File management
  - ➢ create file, delete file
  - ➢ open, close file
  - ➢ read, write, reposition
  - ➢ get and set file attributes

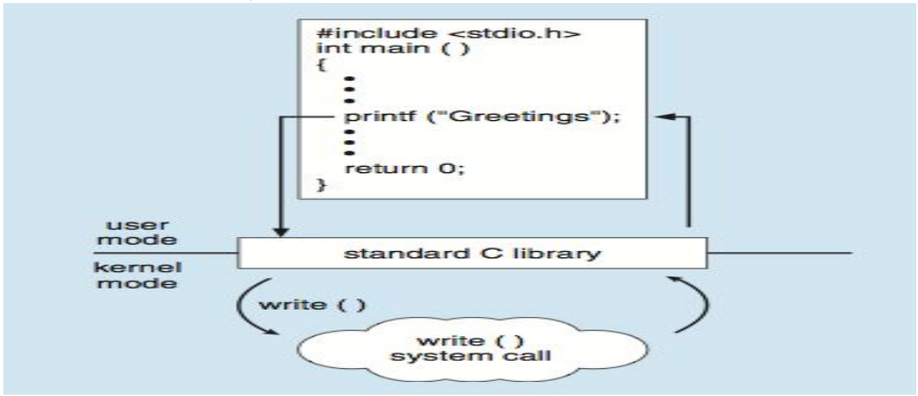
- Device management
  - ➢ request device, release device
  - ➢ read, write, reposition
  - ➢ get device attributes, set device attributes
  - ➢ logically attach or detach devices
- Information maintenance
  - ➢ get time or date, set time or date
  - ➢ get system data, set system data
  - ➢ get and set process, file, or device attributes
- Communications
  - ➢ create, delete communication connection
  - ➢ send, receive messages if message passing model to host name or process name
    - From **client** to **server**
  - ➢ Shared-memory model create and gain access to memory regions
  - ➢ transfer status information
  - ➢ attach and detach remote devices
- Protection
  - ➢ Control access to resources
  - ➢ Get and set permissions
  - ➢ Allow and deny user access

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Standard C Library Example

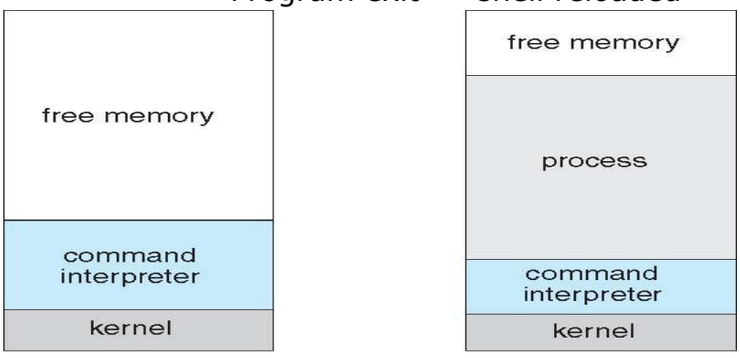
- ➢ C program invoking printf() library call, which calls write() system call



Example: MS-DOS

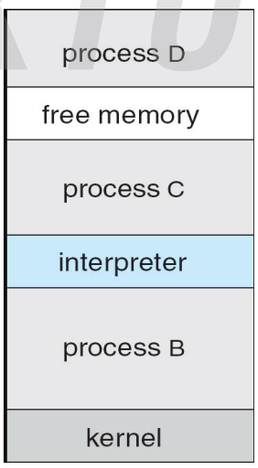
- ➢ Single-tasking

- ➢ Shell invoked when system booted
- ➢ Simple method to run program
- ➢ No process created
- ➢ Single memory space
- ➢ Loads program into memory, overwriting all but the kernel
- ➢ Program exit -> shell reloaded



Example: FreeBSD

- o Unix variant
- o Multitasking
- o User login -> invoke user's choice of shell
- o Shell executes fork() system call to create process
  - ➢ Executes exec() to load program into process
  - ➢ Shell waits for process to terminate or continues with user commands
- o Process exits with:
  - ➢ code = 0 – no error
  - ➢ code > 0 – error code



1.5 Operating System Structure Monolithic, Layered, Micro-kernel, Modular.

Turning away from the user and system call interfaces, let us now take a look at how to implement an operating system. In the next eight sections we will examine some general conceptual issues relating to implementation strategies. After that we will look at some low-level techniques that are often helpful.

12.3.1 System Structure

Probably the first decision the implementers have to make is what the system structure should be. We examined the main possibilities in Sec. 1.7, but will review them here. An unstructured monolithic design is really not a

good idea, except maybe for a tiny operating system in, say, a refrigerator, but even there it is arguable.

Layered Systems

A reasonable approach that has been well established over the years is a layered system. Dijkstra's THE system (Fig. 1-25) was the first layered operating system. UNIX (Fig. 10-3) and Windows 2000 (Fig. 11-7) also have a layered structure, but the layering in both of them is more a way of trying to describe the system than a real guiding principle that was used in building the system.

For a new system, designers choosing to go this route should first very carefully choose the layers and define the functionality of each one. The bottom layer should always try to hide the worst idiosyncracies of the hardware, as the HAL does in Fig. 11-7. Probably the next layer should handle interrupts, context switching, and the MMU, so above this level, the code is mostly machine independent. Above this, different designers will have different tastes (and biases). One possibility is to have layer 3 manage threads, including scheduling and inter thread synchronization, as shown in Fig. 12-1. The idea here is that starting at layer 4 we have proper threads that are scheduled normally and synchronize using a standard mechanism (e.g., mutexes).

In layer 4 we might find the device drivers, each one running as a separate thread, with its own state, program counter, registers, etc., possibly (but not necessarily) within the kernel address space. Such a design can greatly simplify the I/O structure because when an interrupt occurs, it can be converted into an unlock on a mutex and a call to the scheduler to (potentially) schedule the newly readied thread that was blocked on the mutex. MINIX uses this approach, but in UNIX, Linux, and Windows 2000, the interrupt handlers run in a kind of no-man's land, rather than as proper threads that can be scheduled, suspended, etc. Since a huge amount of the complexity of any operating system is in the I/O, any technique for making it more tractable and encapsulated is worth considering.

Above layer 4, we would expect to find virtual memory; one or more file systems, and the system call handlers. If the virtual memory is at a lower level than the file systems, then the block cache can be paged out, allowing the virtual memory manager to dynamically determine how the real memory should be divided among user pages and kernel pages, including the cache. Windows 2000 works this way.

Exokernels

While layering has its supporters among system designers, there is also another camp that has precisely the opposite view (Engler et al., 1995). Their view is based on the end-to-end argument (Saltzer et al., 1984). This concept says that if something has to be done by the user program itself, it is wasteful to do it in a lower layer as well.

Consider an application of that principle to remote file access. If a system is worried about data being corrupted in transit, it should arrange for each file to be check summed at the time it is written and the checksum stored along with the file. When a file is transferred over a network from the source disk to the destination process, the checksum is transferred, too, and also recomputed at the receiving end. If the two disagree, the file is discarded and transferred again.

This check is more accurate than using a reliable network protocol since it also catches disk errors, memory errors, software errors in the routers, and other errors besides bit transmission errors. The end-to-end argument says that using a reliable network protocol is then not necessary, since the end point (the receiving process) has enough information to verify the correctness of the file itself. The only reason for using a reliable network protocol in this view is for efficiency, that is, catching and repairing transmission errors earlier.

The end-to-end argument can be extended to almost all of the operating system. It argues for not having the operating system do anything that the user program can do itself. For example, why have a file system? Just let the user read and write a portion of the raw disk in a protected way. Of course, most users like having files, but the end-to-end argument says that the file system should be a library procedure linked with any program that needs to use files. This approach allows different programs to have different file systems. This line of reasoning says that all the operating system should do is securely allocate resources (e.g., the CPU and the disks) among the competing users. The Exokernel is an operating system built according to the end-to-end argument (Engler et al., 1995).

Client-Server Systems

A compromise between having the operating system does everything and the operating system do nothing is to have the operating system do a little bit. This design leads to a microkernel with much of the operating system running as user-level server processes as illustrated in Fig. 1-27. This is the most modular and flexible of all the designs. The ultimate in flexibility is to have each device driver also run as a user process, fully protected against the kernel and other drivers. Getting the drivers out of the kernel would eliminate the largest source of instability in any operating system—buggy third-party drivers—and would be a tremendous win in terms of reliability.

Of course, device drivers need to access the hardware device registers, so some mechanism is needed to provide this. If the hardware permits, each driver process could be given access to only those I/O devices it needs. For example, with memory-mapped I/O, each driver process could have the page for its device mapped in, but no other device pages. If the I/O port space can be partially protected, the correct portion of it could be made available to each driver.

Even if no hardware assistance is available, the idea can still be made to work. What is then needed is a new system call, available only to device driver processes, supplying a list of (port, value) pairs. What the kernel does is first check to see if the process owns all the ports in the list. If so, it then copies the corresponding values to the ports to initiate device I/O. A similar call can be used to read I/O ports in a protected way.

This approach keeps device drivers from examining (and damaging) kernel data structures, which is (for the most part) a good thing. An analogous set of calls could be made available to allow driver processes to read and write kernel tables, but only in a controlled way and with the approval of the kernel.

The main problem with this approach, and microkernels in general, is the performance hit all the extra context switches cause. However, virtually all work on microkernels was done many years ago when CPUs were much slower. Nowadays, applications that use every drop of CPU power and cannot tolerate a small loss of performance are few and far between. After all, when running a word processor or Web browser, the CPU is probably idle 90% of the time. If a microkernel-based operating system turned an unreliable 900-MHz system into a reliable 800-MHz system, probably few users would complain. After all, most of them were quite happy only a few years ago when they got their previous computer, at the then-stupendous speed of 100 MHz.

Extensible Systems

With the client-server systems discussed above, the idea was to get as much out of the kernel as possible. The opposite approach is to put more modules into the kernel, but in a protected way. The key word here is protected, of course. We studied some protection mechanisms in Sec. 9.5.6 that were initially intended for importing applets over the Internet, but are equally applicable to inserting foreign code into kernel. The most important ones are sandboxing and code signing as interpretation is not really practical for kernel code.

Of course, an extensible system by itself is not a way to structure an operating system. However, by starting with a minimal system consisting of little more than a protection mechanism and then adding protected modules to the kernel one at a time until reaching the functionality desired, a minimal system can be built for the application at hand. In this view, a new operating system can be tailored to each application by including only the parts it requires. Paramecium is an example of such a system (Van Doorn, 2001).

Kernel Threads

Another issue relevant here is that of system threads, no matter which structuring model is chosen. It is sometimes convenient to allow kernel threads to exist, separate from any user process. These threads can run in the background, writing dirty pages to disk, swapping processes between main memory and disk, and so on. In fact, the kernel itself can be structured entirely of such threads, so that when a user does a system call, instead of the user's thread executing in kernel mode, the user's thread blocks and passes control to a kernel thread that takes over to do the work.

In addition to kernel threads running in the background, most operating systems start up many daemon processes in the background, too. While these are not part of the operating system, they often perform ''system'' type activities. These might including getting and sending email and serving various kinds of requests for remote users, such as FTP and Web pages.

1.6 System Boot process

After an operating system is generated, it must be made available for use by the hardware. But how does the hardware know where the kernel is or how

to load that kernel? The procedure of starting a computer by loading the kernel is known as booting the system. On most computer systems, 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. Some computer systems, such as PCs, use a two-step process in which a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel.

When a CPU receives a reset event-for instance, when it is powered up Or rebooted -the instruction register is loaded with a predefined memory location, and execution starts there. At that location is the initial bootstrap program. This program is in the form of read-only memory (ROM), because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot easily be infected by a computer virus.

The bootstrap program can perform a variety of tasks. Usually, one task is to run diagnostics to determine the state of the machine. If the diagnostics pass, the program can continue with the booting steps. It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system.

Some systems-such as cellular phones, PDAs, 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, and rugged operation. 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), which is read only except when explicitly given a command to become writable. 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 (including most general-purpose operating Systems like Windows, Mac OS X, and UNIX) or for systems that change frequently, the bootstrap loader is stored in firmware, and the operating system is on disk. In this case, the bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that block. The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it fits in a single disk block) and knows only the address on disk and length of the remainder of the bootstrap program. is an example of an open-source bootstrap program for Linux systems. All of the disk-bound bootstrap, and the operating system itself, can be easily changed by writing new versions to disk. A disk that has a boot partition (more on that in Section 12.5.1) is called a boot disk or system disk.

Now that the full bootstrap program has been loaded, it can traverse the file system to find the operating system kernel, load it into memory, and start its execution. It is only at this point that the system is said to be running.