Lecture 3

Introduction/Overview III

Lecture Information

Ceng328 *Operating Systems* at March 2, 2010

Dr. Cem Özdoğan
Computer Engineering Department
Çankaya University

Contents

Operating-System Structures

- We can view an OS from several points.
  - One view focuses on the services that the system provides;
  - Another, on the interface that it makes available to users and programmers;
  - A third, on its components and their interconnections.
- We consider what services an OS provides, how they are provided, and what the various methodologies are for designing such systems.

Operating-System Services I

- One set of operating-system services provides functions that are helpful to the user.
  - **User interface.** UI, CLI, BI, GUI
  - **Program execution**.
  - **I/O operations**.
  - **File-system manipulation**.
  - **Communications**. shared memory or through message passing.
  - **Error detection**. The OS needs to be constantly aware of possible errors. Errors may occur in the hardware and in the user program. For each type of error, the OS should take the appropriate action to ensure correct and consistent computing.

Operating-System Services II

- Another set of operating-system functions exists not for helping the user but rather for ensuring the efficient operation of the system itself.
  - **Resource allocation**.
  - **Accounting**. We want to keep track of which users use how much and what kinds of computer resources.
  - **Protection and security**.

User Operating-System Interface

- There are two fundamental approaches for users to interface with the OS.
  1. One technique is to provide a command-line interface or command interpreter that allows users to directly enter commands that are to be performed by the OS.
  2. The second approach allows the user to interface with the OS via a graphical user interface or GUI.

Command Interpreter

- Some OSs include the command interpreter in the kernel. Others, such as Windows XP 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).
- A highly simplified shell illustrating the use of fork, waitpid, and execve is shown in Fig. 1.

```
#define TRUE 1

while (TRUE) {                                   /* repeat forever */
     type_prompt( );                             /* display prompt on the screen */
     read_command(command, parameters);          /* read input from terminal */

     if (fork( ) != 0) {                         /* fork off child process */
         /* Parent code. */
         waitpid(-1, &status, 0);                /* wait for child to exit */
     } else {
         /* Child code. */
         execve(command, parameters, 0);         /* execute command */
     }
}
```

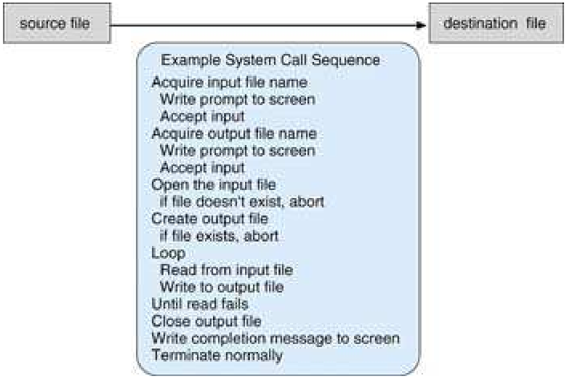
Figure: A stripped-down shell.

Graphical User Interfaces

- A second strategy for interfacing with the OS is through a user-friendly graphical user interface or GUI.
  - Graphical user interfaces first appeared due in part to research taking place in the early 1970s at Xerox PARC research facility. The first GUI appeared on the Xerox Alto computer in 1973.
  - However, graphical interfaces became more widespread with the advent of Apple Macintosh computers in the 1980s.
  - Microsoft's first version of Windows -version 1.O- was based upon a GUI interface to the MS-DOS OS.
  - Traditionally, UNIX systems have been dominated by command-line interfaces, although there are various GUI interfaces available.
- The choice of whether to use a command-line or GUI interface is mostly one of personal preference. As a very general rule, many UNIX users prefer a command-line interface as they often provide powerful shell interfaces.
- Alternatively, most Windows users are pleased to use the Windows GUI environment and almost never use the MS-DOS shell interface.

System Calls I

- If a process is running a user program in user mode and needs a system service, such as reading a data from a file, it has to execute a trap instruction to transfer control to the OS.
- To illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file. The system-call sequence is shown in Fig. 2.



```
Example System Call Sequence
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Figure: Example of how system calls are used.

System Calls II

- As we can see, even simple programs may make heavy use of the OS. Frequently, systems execute thousands of system calls per second.
- The relationship between an application program interface (API), the system-call interface, and the OS is shown in Fig. 3, which illustrates how the OS handles a user application invoking the *open()* system call.

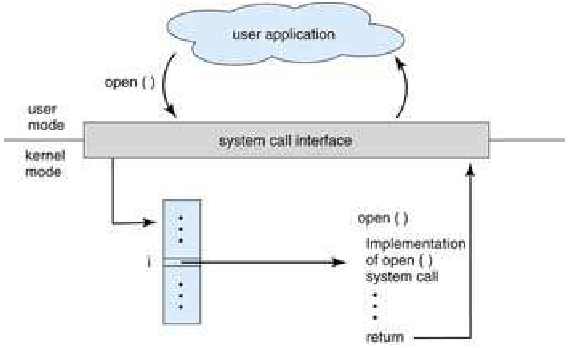


Figure: The handling of a user application invoking the *open()* system call.

System Calls III

- Three general methods are used to pass parameters to the OS.
  1. The simplest approach is to pass the parameters in registers. 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 (see Fig. 4). (Linux and Solaris.)

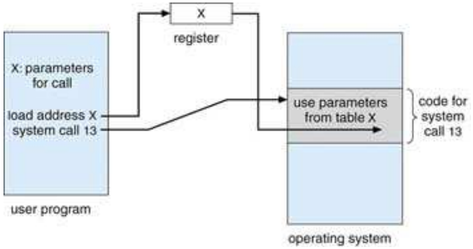


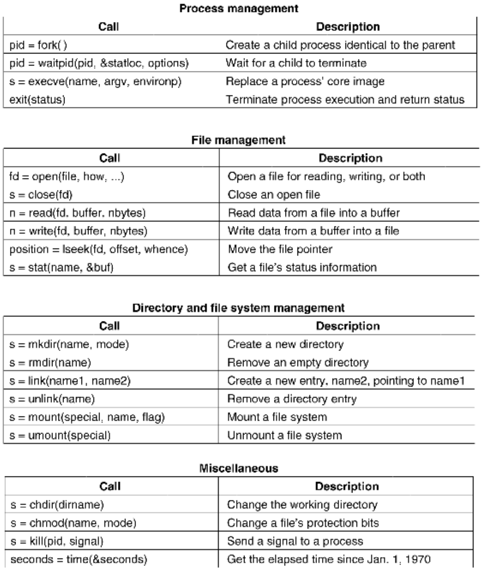
Figure: Passing of parameters as a table.

  2. Parameters also can be placed, or pushed, onto the stack by the program and popped off the stack by the OS.
  3. Some OSs prefer either the block or stack method, because those approaches do not limit the number or length of parameters being passed.

Types of System Calls I

- System calls can be grouped roughly into six major categories:
  1. process control,
  2. file manipulation,
  3. device manipulation,
  4. information maintenance,
  5. communications,
  6. protection.
- Some of the most heavily used POSIX system calls, or more specifically, the library procedures that make those system calls are given in Fig. 5.

Types of System Calls II



Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd. buffer. nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

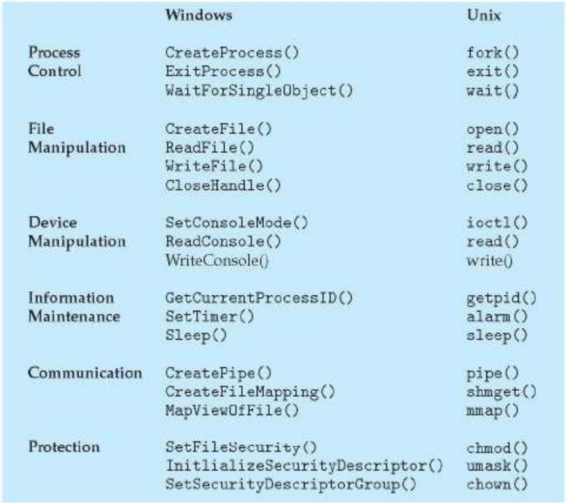
Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Figure: Some of the major POSIX system calls.

Types of System Calls III

Fig. 6 gives some 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()
	InitlializeSecurityDescriptor()
	SetSecurityDescriptorGroup()
		chmod()
		umask()
		chown()

Figure: Examples of Windows and Unix System Calls.

Example of Standard C Library

- As an example, let's assume a C program invokes *printf()* statement.
- The C library intercepts this call and invokes the necessary system call(s) in the OS.
- The C library takes the value returned by *write()* and passes it back to the user program (see Fig. 7).

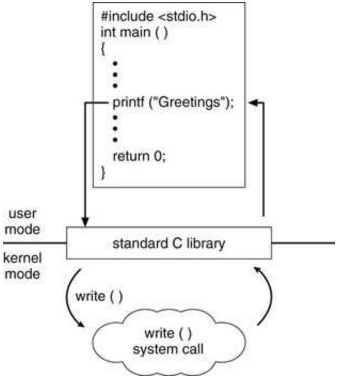


Figure: C library handling of *write()*.

Process Control I

- A process or job executing one program may want to *load* and *execute* another program.
- An interesting question is where to return control when the loaded program terminates. This question is related to the problem of whether the existing program is lost, saved, or allowed to continue execution concurrently with the new program.
- There are so many facets of and variations in process and job control that we next use two examples to clarify these concepts.
  - one involving a single-tasking system
  - the other a multi-tasking system

Process Control II

- The MS-DOS OS is an example of a single-tasking system. It has a command interpreter that is invoked when the computer is started (see Fig. 8(a)).

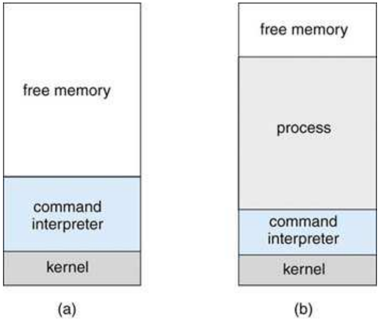


Figure: MS-DOS execution. (a) At system start-up. (b) Running a program.

- 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 (see Fig. 8(b)).
- Next, it sets the instruction pointer to the first instruction of the program.
- The program then runs, and either an error causes a trap, or the program executes a system call to terminate.

Process Control III

- FreeBSD (derived from Berkeley UNIX) is an example of a multitasking system. When a user logs on to the system, the shell of the user's choice is run.

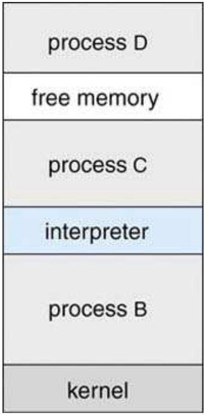


Figure: FreeBSD running multiple programs.

- This shell is similar to the MS-DOS shell in that it accepts commands and executes programs that the user requests.
- However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed (see Fig. 9).
- 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.

File Management

- We first need to be able to *create* and *delete* files. Either system call requires the name of the file and perhaps some of the file's attributes.
- Once the file is created, we need to *open* it and to use it. We may also *read*, *write*, or *reposition* (rewinding or skipping to the end of the file, for example).
- Finally, we need to *close* the file, indicating that we are no longer using it.
- We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system.
- In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary.
- At least two system calls, *get file attribute* and *set file attribute*, are required for this function.

Device Management

- 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.
- The various resources controlled by the OS can be thought of as devices. Some of these devices are physical devices (for example, tapes), while others can be thought of as abstract or virtual devices (for example, files).
- Once the device has been requested (and allocated to us), we can *read*, *write*, and (possibly) *reposition* the device, just as we can with files.
- In fact, the similarity between I/O devices and files is so great that many OSs, including UNIX, merge the two into a combined file-device structure.

Information Maintenance

- Many system calls exist simply for the purpose of transferring information between the user program and the OS.
- For example, most systems have a system call to return the current *time* and *date*.
- In addition, the OS keeps information about all its processes, and system calls are used to access this information.
- Generally, calls are also used to reset the process information (*get process attributes* and *set process attributes*).

Communication

- There are two common models of interprocess communication:
  1. The **message-passing** model. the communicating processes exchange messages with one another to transfer information.
  2. the **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.
- Recall that, normally, the OS tries to prevent one process from accessing another process's memory.
- Shared memory requires that two or more processes agree to remove this restriction. They can then 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.
- Shared memory allows maximum speed and convenience of communication.
- Problems exist, however, in the areas of protection and synchronization between the processes sharing memory.

Operating-System Design and Implementation

- Because an OS is large and complex, it must be created piece by piece. Each of these pieces should be a well delineated portion of the system, with carefully defined inputs, outputs, and functions.
- *Large Systems:* 100k's to millions of lines of code involving 100 to 1000 man-years of work
- *Complex:* Performance is important while there is conflicting needs of different users.
- It is not possible to remove all bugs from such complex and large software. Behavior is hard to predict; tuning is done by guessing.

Design Goals

- At the highest level, the design of the system will be affected by the choice of hardware and the type of system: batch, time shared, single user, multiuser, distributed, real time, or general purpose.
- The requirements can, however, be divided into two basic groups: user goals and system goals.
  1. Users desire certain obvious properties in a system: **The system should be convenient to use, easy to learn and to use, reliable, safe, and fast.**
  2. A similar set of requirements can be defined by those people who must design, create, maintain, and operate the system: **The system should be easy to design, implement, and maintain; it should be flexible, reliable, error free, and efficient.**
- There is, in short, no unique solution to the problem of defining the requirements for an OS.
- The wide range of systems in existence shows that different requirements can result in a large variety of solutions for different environments.

Mechanisms and Policies

- One important principle is the separation of policy from mechanism.
  - Mechanisms determine **how** to do something; policies determine **what** will be done.
  - For example, the timer construct is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.
- The separation of policy and mechanism is important for flexibility. Policies are likely to change across places or over time.
- Policy decisions are important for all resource allocation. 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 I

- Once an OS is designed, it must be implemented.
- The advantages of using a higher-level language, or at least a systems-implementation language, for implementing OSs are the same as those accrued when the language is used for application programs:
    - 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 OS by simple recompilation.
    - Finally, an OS is far easier to port (to move to some other hardware) if it is written in a higher-level language.
        - For example, MS-DOS was written in Intel 8088 assembly language. Consequently, it is available on only the Intel family of CPUs.
        - The Linux OS, in contrast, is written mostly in C and is available on a number of different CPUs, including Inte180X86, Motorola 680XO, SPARC, and MIPS RXOOO.

Implementation II

- The only possible disadvantages of implementing an OS in a higher-level language are reduced speed and increased storage requirements.
- As is true in other systems, major performance improvements in OSs are more likely to be the result of better data structures and algorithms than of excellent assembly-language code.
- In addition, although OSs are large, only a small amount of the code is critical to high performance; the memory manager and the CPU scheduler are probably the most critical routines.

Simple Structure I

- MS-DOS was originally designed and implemented by a few people who had no idea that it would become so popular.
- It was written to provide the most functionality in the least space, so it was not divided into modules carefully. Fig. 10 shows its structure.

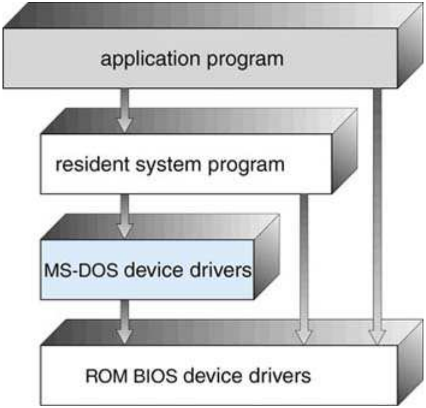


Figure: MS-DOS layer structure.

Simple Structure II

- In MS-DOS, the interfaces and levels of functionality are not well separated.
- For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives.
- Vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail.
- Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible.

Simple Structure III

- Another example of limited structuring is the original UNIX OS. (initially limited by hardware functionality)
- It consists of two separable parts: the kernel and the system programs.
- The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved.
- The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.
- Taken in sum, that is an enormous amount of functionality to be combined into one level. This monolithic structure was difficult to implement and maintain.
- Monolithic systems is the most common organization. The structure is that there is no structure.
- In terms of information hiding, there is essentially none – every procedure is visible to every other procedure.
- Even in monolithic systems, however, it is possible to have at least a little structure, remember the system calls.

Simple Structure IV

- We can view the traditional UNIX OS as being layered, as shown in Fig. 11.

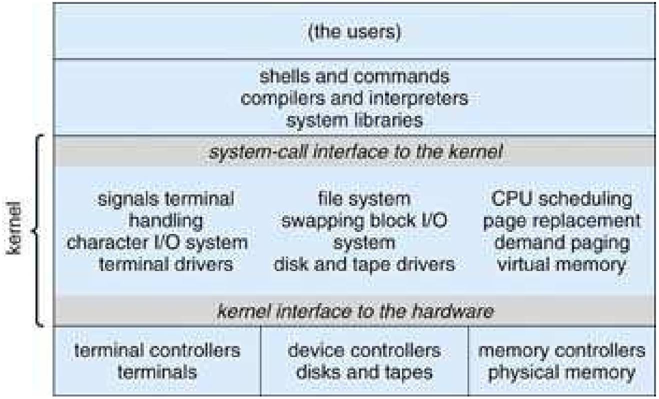


Figure: UNIX system structure.

Layered Approach I

- With proper hardware support, OSs can be broken into pieces that are smaller and more appropriate than those allowed by the original MS-DOS or UNIX systems.
- A system can be made modular in many ways. One method is the **layered approach**, in which the OS is broken up into a number of layers (levels) (seen Fig. 12).

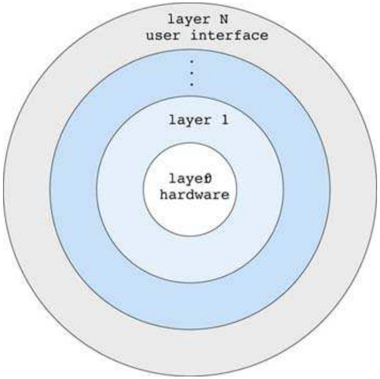


Figure: A layered operating system.

Layered Approach II

- The first system constructed in this way was the THE system built at the Technische Hogeschool Eindhoven in the Netherlands by E. W. Dijkstra (1968) and his students.
- The system had 6 layers, as shown in Fig. 13.

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Figure: Structure of the THE operating system.

Layered Approach III

1. Layer 0 dealt with allocation of the processor, switching between processes when interrupts occurred or timers expired. Layer 0 provided the basic multiprogramming of the CPU.
2. Layer 1 did the memory management. It allocated space for processes in main memory. Layer 1 software took care of making sure pages were brought into memory whenever they were needed.
3. Layer 2 handled communication between each process and the operator console.
4. Layer 3 took care of managing the I/O devices and buffering the information streams to and from them. Above layer 3 each process could deal with abstract I/O devices with nice properties, instead of real devices with many peculiarities.
5. Layer 4 was where the user programs were found. They did not have to worry about process, memory, console, or I/O management.
6. The system operator process was located in layer 5.

Layered Approach IV

- A further generalization of the layering concept was present in the MULTICS (Multiplexed Information and Computing Service, an extremely influential early time-sharing OS, 1964) system. (concentric rings)
- 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.
- A layer does not need to know how these operations are implemented; it needs to know only what these operations do.
- Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.
- A problem with layered implementations is that they tend to be less efficient than other types.
- 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 I

- Ten bugs per thousand lines of code. This means that a monolithic OS of five million lines of code is likely to contain something like 50000 kernel bugs.
- Modularizing the kernel using the **microkernel** approach: **This method structures the OS by removing all nonessential components from the kernel and implementing them as system and user-level programs.**
- If the hardware provides multiple privilege levels, then the microkernel is the only software executing at the most privileged level.
- The result is a *smaller* kernel. Typically, microkernels provide minimal process and memory management, in addition to a communication facility.
- The main function of the microkernel is to provide *a communication facility between the client program and the various services* that are also running in user space.

Microkernels II

- Communication is provided by message passing.
- For example, if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.
- Unfortunately, microkernels can suffer from performance decreases due to increased system function overhead. Consider the history of Windows NT.
  - The first release had a layered microkernel organization. However, this version delivered low performance compared with that of Windows 95.
  - Windows NT 4.0 partially redressed the performance problem by moving layers from user space to kernel space and integrating them more closely.
  - By the time Windows XP was designed, its architecture was more monolithic than microkernel.

Microkernels III

- The MINIX 3 microkernel is only about 3200 lines of C and 800 lines of assembler for very low-level functions such as catching interrupts and switching processes.
- The C code manages and schedules processes, handles interprocess communication (by passing message between processes), and offers a set of about 35 kernel calls. The process structure of MINIX 3 is shown in Fig. 14

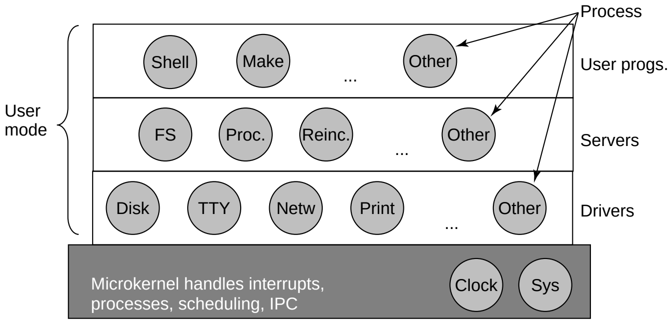


Figure: Structure of the MINIX 3 system.

Modules I

- Perhaps the best current methodology for operating-system design involves using object-oriented programming techniques to create a **modular kernel**.
- Here, the kernel has a set of core components and dynamically links in additional services *either during boot time or during run time*. (dynamically loadable modules)
- Such a design allows the kernel to provide core services yet also allows certain features to be implemented dynamically.
- For example, device and bus drivers for specific hardware can be added to the kernel, and support for different file systems can be added as loadable modules.
- The overall result resembles a layered system in that each kernel section has defined, protected interfaces;
  but it is more flexible than a layered system in that any module can call any other module.

Modules II

For example, the Solaris OS structure, shown in Fig. 15, is organized around a core kernel with seven types of loadable kernel modules.

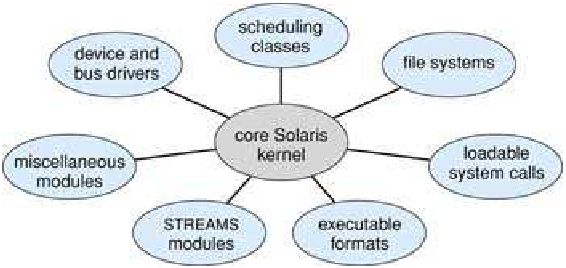


Figure: Solaris loadable modules.

Virtual Machines I

- The layered approach described in Section 2 is taken to its logical conclusion in the concept of a **virtual machine**.
- The fundamental idea behind a virtual machine is *to abstract the hardware of a single computer* (the CPU, memory, disk drives, network interface cards, and so forth) *into several different execution environments*, thereby creating the illusion that each separate execution environment is running its own private computer.
- By using CPU scheduling and virtual-memory techniques, an OS can create the illusion that a process has its own processor with its own (virtual) memory.
- There are several reasons for creating a virtual machine, all of which are fundamentally related to being able *to share the same hardware yet run several different execution environments* (that is, different OSs) concurrently.

Virtual Machines II

- Normally, a process has additional features, such as system calls and a file system, that are not provided by the bare hardware.
- The virtual-machine approach does not provide any such additional functionality but rather provides an interface that is *identical* to the underlying bare hardware.
- Each process is provided with a (virtual) copy of the underlying computer (see Fig. 16).

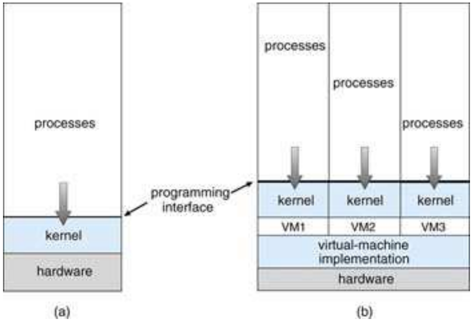


Figure: System models. (a) Nonvirtual machine. (b) Virtual machine.

Examples - VMware

- Despite the advantages of virtual machines, they received little attention for a number of years after they were first developed.
- VMware is a popular commercial application that abstracts Intel 80X86 hardware into isolated virtual machines.
- VMware runs as an application on a host OS and allows this host system to concurrently run several different **guest OSs** as independent virtual machines (see Fig. 17).

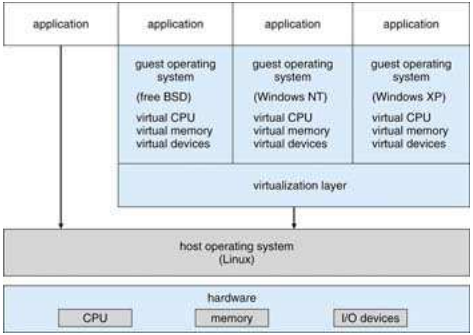


Figure: VMware architecture.

Examples - The Java Virtual Machine I

- Java is a popular object-oriented programming language introduced by Sun Microsystems in 1995. In addition to a language specification and a large API library, Java also provides a specification for a Java virtual machine or JVM.
- Java objects are specified with the class construct; a Java program consists of one or more classes.
- For each Java class, the compiler produces an architecture-neutral **bytecode** output (*.class*) file that will run on any implementation of the JVM.
- The JVM also automatically manages memory by performing **garbage collection** - the practice of reclaiming memory from objects no longer in use and returning it to the system.
- Much research focuses on garbage collection algorithms for increasing the performance of Java programs in the virtual machine.

Examples - The Java Virtual Machine II

The JVM is a specification for an abstract computer. It consists of a **class loader** and a Java interpreter that executes the architecture-neutral bytecodes, as diagrammed in Fig. 18.

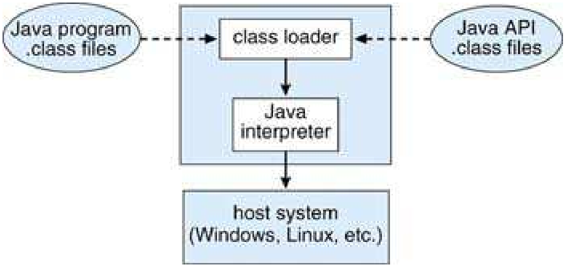


Figure: The Java virtual machine.