Lecture **5,6,7,8,9,10**

Topics to Be Covered:

* Structure of os
* System call
* Process
* PCB

**Structure of os**

### Simple Structure

* Layered Approach
* Microkernel
* Modules

### Simple Structure

### Many operating systems do not have well-deﬁned structures; such systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become



ROM BIOS device drivers

resident system program

application program

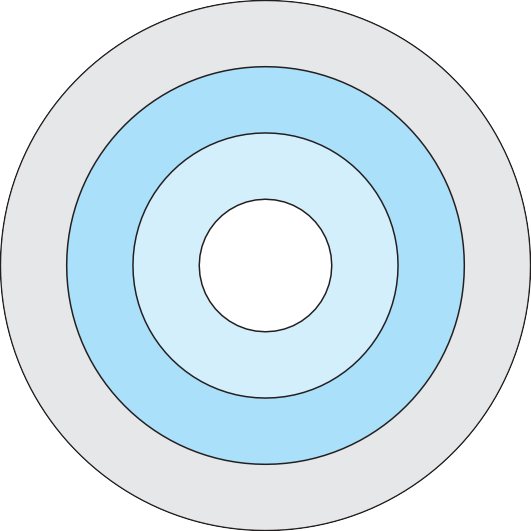
MS-DOS device drivers

### so popular. It was written to provide the most functionality in the least space, so it was not carefully divided into modules 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. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail. Of course, MS-DOS was also limited by the hardware of its era. Another example of limited structuring is the original UNIX operating system. Like MS-DOS, UNIX initially was 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 This monolithic structure was difﬁcult to implement and maintain. It had a distinct performance advantage, however: there is very little overhead in the system call interface or in communication within the kernel. We find this simple, monolithic structure in the UNIX, Linux, and Windows operating systems.

**Layered Approach**

In this 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. This layering structure is shown in Figure 2.2.



layer N user interface

•

•

•

layer 1

layer 0 hardware

**Figure2.2** A layered operating system

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. This approach simpliﬁes debugging and system veriﬁcation. The ﬁrst layer can be debugged without any concern for the rest of the system, because, by deﬁnition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the ﬁrst layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system are simpliﬁed. Each layer is implemented only with operations provided by 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.

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. For example, the device driver for the backing store (disk space used by virtual-memory algorithms) must be at a lower level than the memory-management routines, because memory management requires the ability to use the backing store.

**Microkernel**

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. There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. Typically, however, microkernels provide minima process and memory management, in addition to a communication facility. Figure 2.4 show the architecture of a typical microkernel. 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. Communication is provided through **message passing,** , 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.

One beneﬁt of the microkernel approach is that it makes extending the operating system easier. All

new services are added to user space and consequently do not require modiﬁcation of the kernel. When the kernel does have to be modiﬁed, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another. 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. Some contemporary operating systems have used the microkernel approach. Tru64 UNIX (formerly Digital UNIX) provides a UNIX interfac to the user, but it is implemented with a Mach kernel.

messages

messages

Interprocess Communication

memory managment

CPU

scheduling

microkernel

hardware

Device Driver

File System

Application Program

**Modules**

Perhaps the best current methodology for operating-system design involves using **loadable kernel modules**. Here, 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. It resembles a layered system in that each kernel section has deﬁned, protected interfaces; but it is more ﬂexible than a layered system, because any module can call any other module. The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efﬁcient, because modules do not need to invoke message passing in order to communicate.

The Solaris operating system structure, shown in Figure 2.5, is organized around a core kernel with seven types of loadable kernel modules:

device and bus drivers

scheduling classes

file systems

miscellaneous modules

core Solaris kernel

loadable system calls

STREAMS

modules

executable formats

**Figure 2.5** Solaris loadable modules

**System Call:**

* System calls provide an interface between the process and the operating system.
* System calls allow user-level processes to request some services from the operating system which process itself is not allowed to do.
* For example, for I/O a process involves a system call telling the operating system to read or write particular area and this request is satisfied by the operating system.

The following different types of system calls provided by an operating system:

**Process control**

* end, abort
* load, execute
* create process, terminate process
* get process attributes, set process attributes
* wait for time
* wait event, signal event
* allocate and free memory

**File management**

* create file, delete file
* open, close
* read, write, reposition
* get file attributes, 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 process, file, or device attributes
* set process, file, or device attributes

**Communications**

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

**PROCCESS CONCEPT**

A process is a program in execution. A process is more than the program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers. In addition, a process generally includes the process stack, which contains temporary data (such as method parameters, return addresses, and local variables), and a data section, which contains global variables.

An operating system executes a variety of programs:

Batch system – jobs

Time-shared systems – user programs or tasks

Process – a program in execution; process execution must progress in sequential fashion.

A process includes: program counter , stack, data section

**What is the difference between process and program?**

Both are same beast with different name or when this beast is sleeping (not executing) it is called program and when it is executing becomes process.

Program is a static object whereas a process is a dynamic object.

A program resides in secondary storage whereas a process resides in main memory.

The span time of a program is unlimited but the span time of a process is limited.

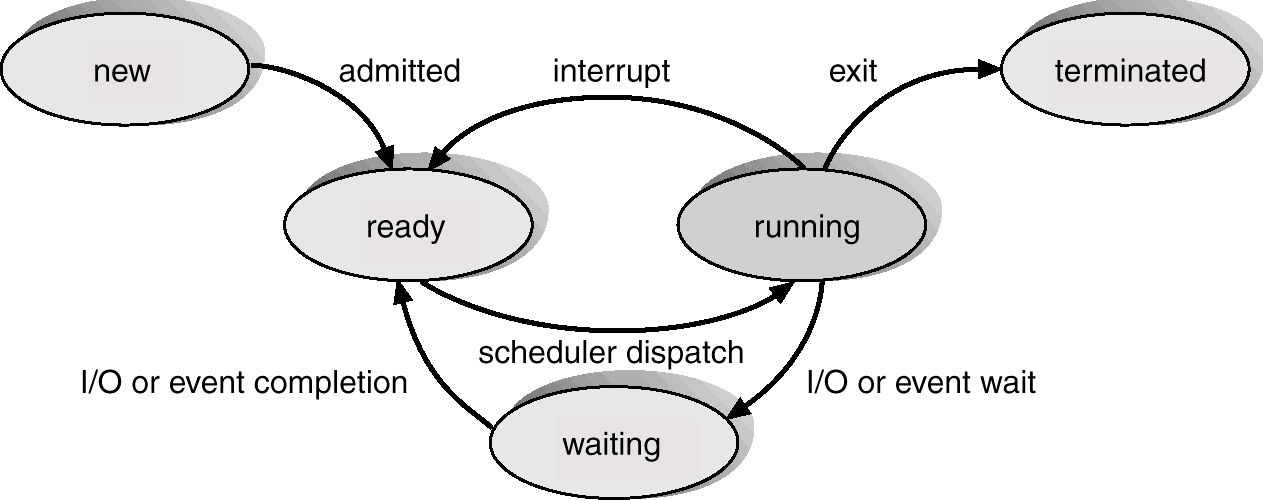
A process is an 'active' entity whereas a program is a 'passive' entity.

A program is an algorithm expressed in programming language whereas a process is expressed in assembly language or machine language.

**Process State**

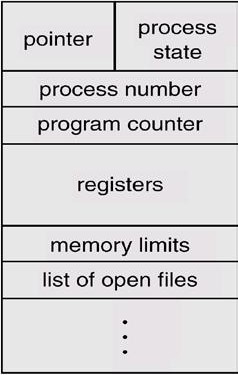
As a process executes, it changes state

* **New State:** The process is being created.
* **Running State:** A process is said to be running if it has the CPU, that is, process actually using the CPU at that particular instant.
* **Blocked (or waiting) State:** A process is said to be blocked if it is waiting for some event to happen such that as an I/O completion before it can proceed. Note that a process is unable to run until some external event happens.
* **Ready State:** A process is said to be ready if it needs a CPU to execute. A ready state process is runnable but temporarily stopped running to let another process run.
* **Terminated state:** The process has finished execution.



**Process Control Block (PCB)**

Information associated with each process



**Process state:** The state may be new, ready, running, waiting, halted, and SO on.

**Program counter:** The counter indicates the address of the next instruction to be executed for this process.

**CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

**CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

**Memory-management information:** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

**Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

**Status information:** The information includes the list of I/O devices allocated to this process, a list of open files, and so on.

**The PCB simply serves as the repository for any information that may vary from process to process.**

**PART A**

Q.1 Explain the various states of the process using suitable diagram.                                          **(RTU-2013)**

Q.2 What is a process? What is the difference between a program and a process? Explain PCB using a suitable example.

**(RTU-14)**

Q.3 What is the main difference between monolithic and microkernel architecture?

Q.4 Explain the difference between microkernel and macro kernel?

Q.5 What is the main advantage of the layered approach to system design?

Q.6 What is the purpose of system calls?