OPERATING SYSTEM

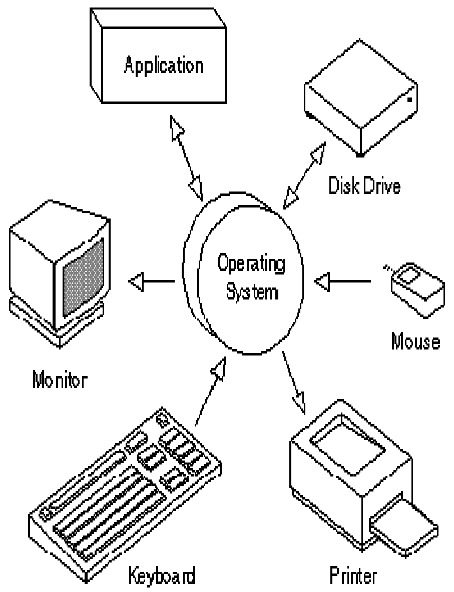
SCSX1011

UNIT I

Introduction-Operating system structures- System Components-OS Services-System Calls-System Structure-Resources- Processes-Threads-Objects-Device Management-Different approaches-Buffering device drivers.

1.Introduction

- Operating System is a system software that acts as an intermediary between a user and Computer Hardware to enable convenient usage of the system and efficient utilization of resources.Also an operating system is a program designed to run other programs on a computer.



- Operating system is the most important program that runs on a computer. OS is considered as the backbone of a computer, managing both software and hardware resources. They are responsible for everything from the control and allocation of memory to recognizing input from external devices and transmitting output to computer displays.

- They also manage files on computer hard drives and control peripherals, like printers and scanners. Operating systems monitor different programs and users, making sure everything runs smoothly, without interference, despite the fact that numerous devices and programs are used simultaneously.
- An operating system also has a vital role to play in security. Its job includes preventing unauthorized users from accessing the computer system.

Operating System Goals

- Execute user programs and make solving user problems easier.
- Make the computer system convenient to use.
- Use the computer hardware in an efficient manner.

Classification of Operating systems

Multi-user OS

Allows two or more users to run programs at the same time. This type of operating system may be used for just a few people or hundreds of them. In fact, there are some operating systems that permit hundreds or even thousands of concurrent users.

Multiprocessing OS

Support a program to run on more than one central processing unit (CPU) at a time. This can come in very handy in some work environments, at schools, and even for some home-computing situations.

Multitasking OS

Allows to run more than one program at a time.

Multithreading OS

Allows different parts of a single program to run concurrently (simultaneously or at the same time).

Real time OS

These are designed to allow computers to process and respond to input instantly. Usually, general-purpose operating systems, such as disk operating system (DOS), are not considered real time, as they may require seconds or minutes to respond to input. Real-time operating systems are typically used when computers must react to the consistent input of information without delay. For example, real-time operating systems may be used in navigation. General-purpose operating systems, such as DOS and UNIX, are not real-time. Today's operating systems tend to have graphical user interfaces (GUIs) that employ pointing devices for input. A mouse is an example of such a pointing device, as is a stylus. Commonly used operating systems for IBM-compatible personal computers include Microsoft Windows, Linux, and Unix variations. For Macintosh computers, Mac OS X, Linux, BSD, and some Windows variants are commonly used.

2. Operating System Structures

An OS provides the environment within which programs are executed. Internally, Operating Systems vary greatly in their makeup, being organized along many different lines. The design of a new OS is a major task. The goals of the system must be well defined before the design begins. The type of system desired is the basis for choices among various algorithms and strategies. An OS may be viewed from several vantage ways.

- By examining the services that it provides.
- By looking at the interface that it makes available to users and programmers.
- By disassembling the system into its components and their interconnections.

3. System Components

We can create a system as large and complex as an operating system by partitioning it into smaller pieces. Each piece should be a well-delineated (represented accurately or precisely) portion of the system with carefully defined inputs, outputs and functions. Even though, not all systems have the same structure. However, many modern operating systems share the same goal of supporting the following types of system components:

- Process Management
- Main Memory Management
- File Management
- I/O System Management
- Secondary Management
- Networking
- Protection System
- Command-Interpreter System

Process Management

The operating system manages many kinds of activities ranging from user programs to system programs like printer spooler, name servers, file server etc. Each of these activities is encapsulated in a process. A program by itself is not a process but a program in execution. For example

- A batch job is a process
- A time-shared user program is a process
- A system task (e.g. spooling output to printer) is a process.

There can be many processes running the same program. A program does nothing unless its instructions are executed by a CPU. The execution of a process must be sequential. The five major activities of an operating system in regard to process management are:

- Creation and deletion of user and system processes.
- Suspension and resumption (Block/Unblock) of processes.
- Providing mechanism for process Synchronization.
- Providing mechanism for process Communication.
- Providing mechanism for process deadlock handling.

Main Memory Management

Main memory is central to the operation of a modern computer system. Primary-Memory or Main-Memory is a large array of words or bytes ranging in size from hundreds of thousands to billion. Each word or byte has its own address. Main-memory provides storage that can be accessed directly by the CPU. The main memory is only large storage device that the CPU is able to address and access directly for a program to be executed, that must in the main memory. To improve both the utilisation of the CPU and the speed of the computer's response to its users, we must keep several programs in memory.

- The major activities of an operating system in regard to memory-management are:
- Monitoring which part of memory are currently being used and by whom.
- Deciding which process are loaded into memory when memory space becomes available.
- Allocating and deallocating memory space as needed.

File Management

File management is one of the most visible components of an OS. Computers can store information on several different types of physical media (e.g. magnetic tap, magnetic disk, CD etc). Each of these media has its own properties like speed, capacity, data transfer rate and access methods.For convenient use of the computer system, the OS provides a uniform logical view of information storage.

- A file is a logical storage unit, which abstracts away the physical properties of its storage device. A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data.
- The operating system is responsible for the following activities in connection with file management:
    - Creation and deletion of files.
    - Creation and deletion of directions.
    - Support of primitives for manipulating files and directions.
    - Mapping of files onto secondary storage.
    - Backing up of files on stable (non volatile) storage media.

I/O System Management

OS hides the peculiarities of specific hardware devices from the user. I/O subsystem consists of:

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

Only the device driver knows the peculiarities of the specific device to which it is assigned.

Secondary-Storage Management

The main purpose of a computer system is to execute programs. These programs, with the data they access ,must be in main memory, or primary storage.

• Systems have several levels of storage, including primary storage, secondary storage and cache storage.

• Since main memory (*primary storage*) is volatile and too small to accommodate all data and programs permanently, the computer system must provide *secondary storage* to back up main memory.

• Most modern computer systems use disks as the principle on-line storage medium, for both programs and data.

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

- Free-space management (paging/swapping)
- Storage allocation (what data goes where on the disk)
- Disk scheduling (Scheduling the requests for memory access).

Networking

A distributed systems is a collection of processors that do not share memory, peripheral devices, or a clock. Instead, each processor has its own local memory and clock, and the processors communicate with one another through various communication lines such as network or high-speed buses.

The processors in a distributed system vary in size and function. They may include small processors, workstations, minicomputers and large, general-purpose computer systems.

The processors in the system are connected through a communication-network ,which are configured in a number of different ways i.e.., Communication takes place using a *protocol.*The network may be fully or partially connected .

- The communication-network design must consider routing and connection strategies, and the problems of contention and security.
- A distributed system provides user access to various system resources.
- Access to a shared resource allows:
    - Computation Speed-up
    - Increased functionality
    - Increased data availability
    - Enhanced reliability

Protection System

- If a computer system has multiple users and allows the concurrent execution of multiple processes, then the various processes must be protected from one another's activities.
- Protection refers to mechanism for controlling the access of programs, files, memory segments, processes(CPU) only by the users who have gained proper authorization from the OS.
- The protection mechanism must:
    - Distinguish between authorized and unauthorized usage.
    - Specify the controls to be imposed.
    - Provide a means of enforcement.

Command Interpreter System

- A command interpreter is one of the important system programs for an OS. It is an interface of the operating system with the user. The user gives commands, which are executed by Operating system (usually by turning them into system calls).
- The main function of a command interpreter is to get and execute the next user specified command. Many commands are given to the operating system by control statements which deal with:

- process creation and management
- I/O handling
- secondary-storage management
- main-memory management
- file-system access
- protection
- networking

4. Operating system services

- An OS provides environment for the execution of programs. It provides certain services to programs and to the users of those programs. The specific services provided differs from one OS to another, but we can identify common classes.
- These OS services are provided for the convenience of the programmer, to make the programming task easier. One set of **operating-system services** provides functions that are helpful to the user are

Program execution

- The system must be able to load a program into memory and to run that program.
- The program must be able to end its execution, either normally or abnormally (indicating error).

I/O operations

- A running program may require I/O, which may involve a file or an I/O device.
- For specific devices, special functions may be desired (rewind a tape drive, or to blank a CRT).
- For efficiency and protection, users usually cannot execute I/O operations directly.
- Therefore Operating system must provide some means to perform I/O.

File-system manipulation

- The file system is of particular interest.
- Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

Communications

- One process needs to exchange information with another process. Such communication can occur in two ways:

- The first takes place between processes that are executing on the same computer.
- The second takes place between processes that are executing on different computers over a network.
- Communications may be implemented via **shared memory** or through **message passing**, in which packets of information moved between the processes by the OS.

Error detection

OS needs to be constantly aware of possible errors. Errors may occur

- In the CPU and memory hardware (such as a memory error or power failure)
- In I/O devices (such as a parity error on tape, a connection failure on a network or lack of power in the printer).
- And in user program (such as arithmetic overflow, an attempt to access illegal memory location, or a too-great use of CPU time).
- For each type of error, OS should take the appropriate action to ensure correct and consistent computing.
- Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system
- Another set of OS functions exists not for helping user, but for ensuring the efficient operation of the system itself via resource sharing. Systems with multiple users can gain efficiency by sharing the computer resources among the users.

Resource allocation

- When multiple users logged on the system or multiple jobs running at the same time, resources must be allocated to each of them.
- Many types of resources are managed by OS. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have general request and release code.

Accounting

To keep track of which users use how much and what kinds of computer resources. This record may be used for accounting (so that users can be billed) or simply for accumulating usage statistics.

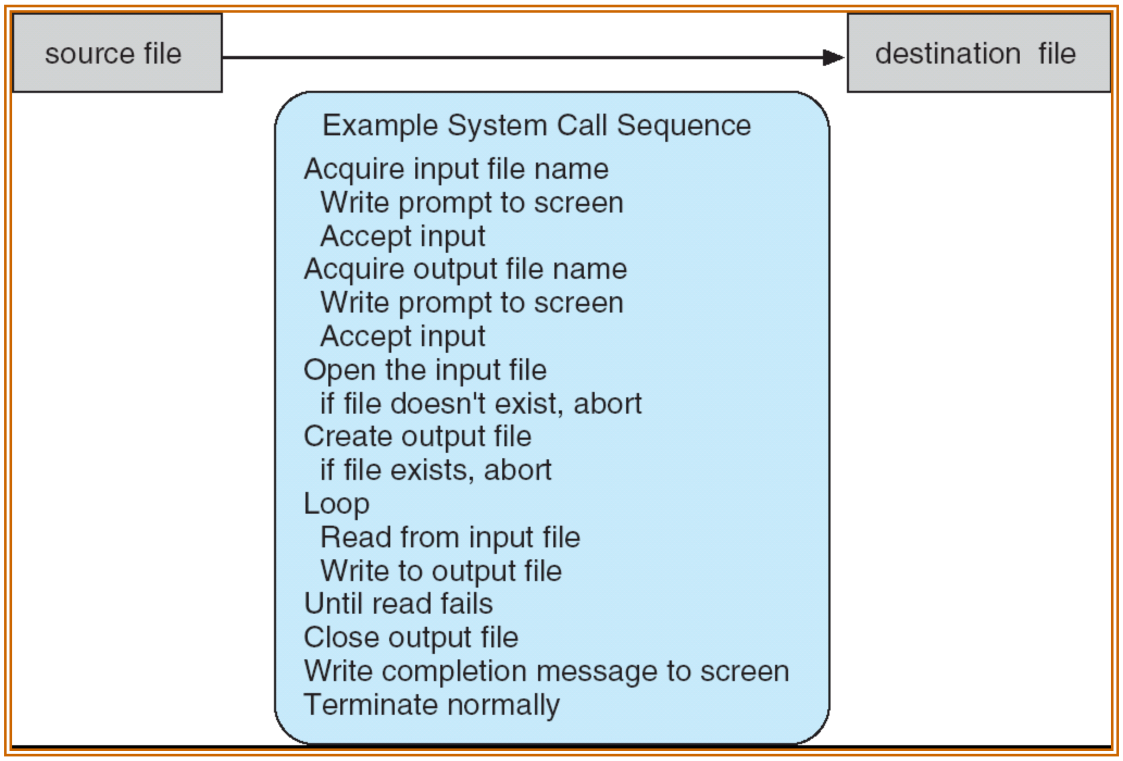
Protection and security

- The owners of information stored in a multi-user or networked computer system may want to control the use of that information. When several disjointed processes execute concurrently, processes should not interfere with each other or with the OS itself.
- Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts. If a system is to be protected and secure, precautions must be instituted throughout. A chain is only as strong as its weakest link.

5. System Calls

- System calls provide the interface between a process and the operating system. These calls are generally available as assembly-language instructions.
- Some systems also allow to make system calls from a high level language, such as C, C++ and Perl (have been defined to replace assembly language for systems programming). As an example of how system calls are used,consider writing a simple program to read data from one file and to copy them to another file.
- The first input that the program will need is the names of the two files:
  - The input file and
  - The output file
- These names can be specified in many ways,depending on the OS design:
  - One approach is for the program to ask the user for the names of the two files.
    I. In an *interactive system*, this approach will require a sequence of system calls, first to write a prompting message on the screen, and then to read from the keyboard the charaters that define the two files.
    II. On *mouse-based window-and-menu systems*, a menu to select the source name and a similar window can be opened to select the destination name.
- Once the two file names are obtained, the program must open the input file and create the output file.
- Each of these operations requires another system call and may encounter possible error conditions.
  - When the program tries to open the file, it may find that *no file of that name exists* or that the *file is protected against access*.

- In these cases, the program should print a message on the console(another sequence of system calls), and then terminate abnormally(another system call).
- If the *input file* exists, then we must create a new *output file*.
    - We may find an *output file* with the same name.
        - This situation may cause the *program to abort* (a system call), or
        - we may *delete the existing file* (another system call).
    - In an *interactive system* another option is to ask the user (a sequence of system calls to output the prompting message and to read response from the keyboard) whether to *replace the existing file* or to *abort the program*.



Example System Call Sequence

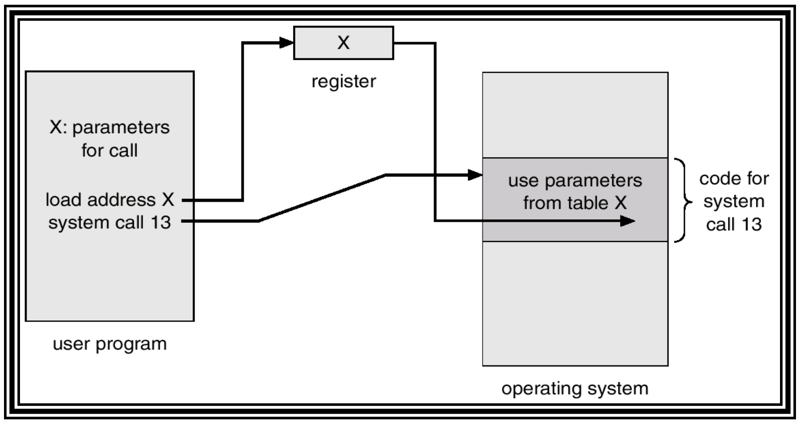
```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close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

- Now that both the files are setup, we enter a ***loop*** that reads from the *input file* (a **system call**) and writes to the *output file* (another **system call**).
- Each *read* and *write* must return status information regarding various possible error conditions.
    - On input,
        - the program may find that the *end of file has been reached*, or
        - that a *hardware failure occured* in the *read* (such as a parity error).
    - On output,
        - Various errors may occur, depending on the output device (such as no more disk space, physical end of tape, printer out of paper).

- Finally, after the entire file is copied,
    - The program may close both files (another **system call**), writes a message to the console(more sytem calls),and finally terminates normall (the final **system call**).
- As we can see, even simple programs may make heavy use of the OS.

- **System calls** occur in different ways, depending on the computer in use.
- Often, more information is required than simply the identity of the desired system call.
- The exact type and amount of information vary according to the particular OS

- **Three general methods** are used to pass parameters between a running program and the operating system.
    - Simplest approach is to pass parameters in *registers*.
    - Store the parameters in a table in memory, and the table address is passed as a parameter in a register (in the cases where parameters are more than registers).
    - *Push* (store) the parameters onto the *stack* by the program, and *pop* off the stack by operating system.

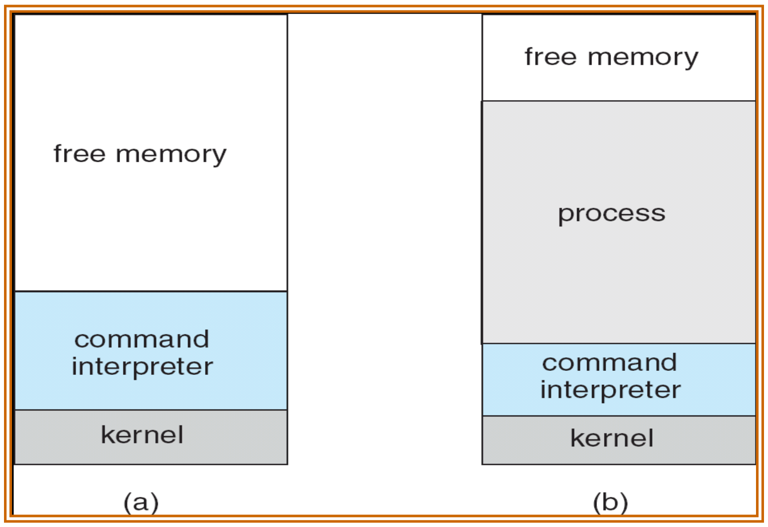


Types of System Calls

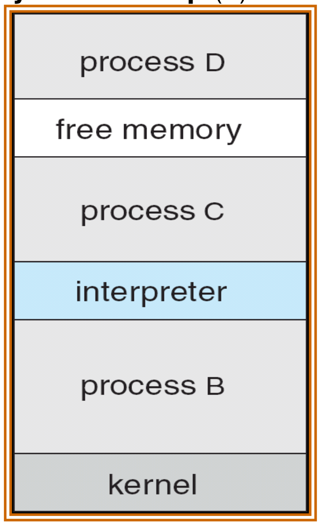
System calls can be grouped roughly in to five categories:

i. **Process control**
ii. **File management**
iii. **Device management**
iv. **Information maintenance**
v. **Communications**

Process control – ***load, execute, abort, end, create process,terminate process, get process attributes,set process attributes, allocate and free memory, wait event ,signal event.***



MS-DOS execution. (a) **At system startup** (b) **running a program**



UNIX Running Multiple Programs

File management – create file, delete file, open, close, read, write,reposition, get file attribute,set file attributes.

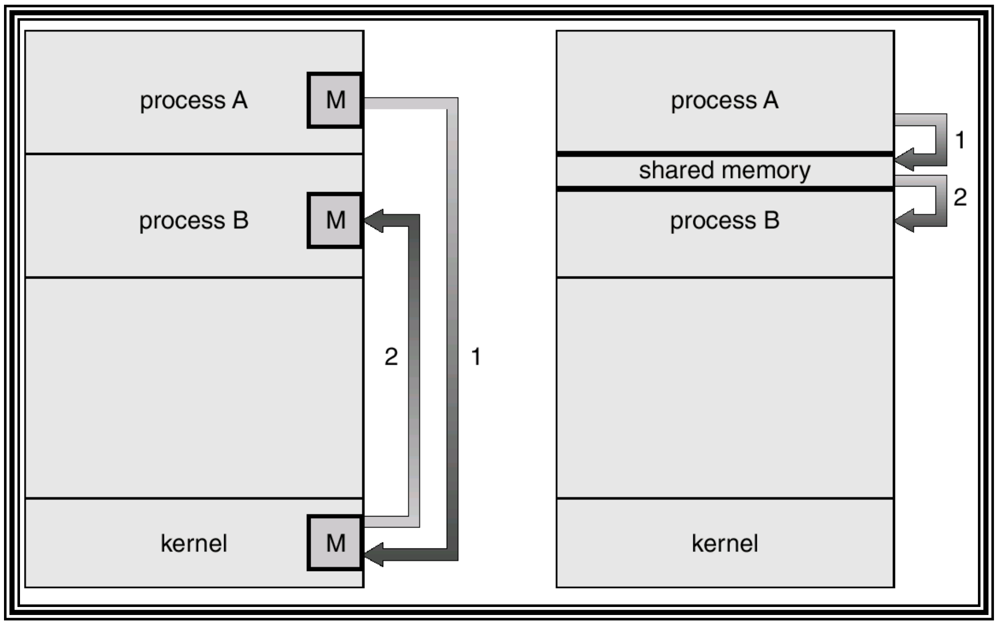
Device management – request device, release device, read, reposition,write,get device attributes,set device attributes, logically attach or detach device.

Information maintenance – get time and date, set time and date, get system data,set system data,get process file or device attributes,set process file or device attributes.

Communications – create, close communication connection, send, receive messages, transfer status information,attach or detach remote devices. Communication may take place using:

i. **message passing model** or

ii. **shared memorymodel**



Communication models: (a) **message passing model** (b) **shared memory model**

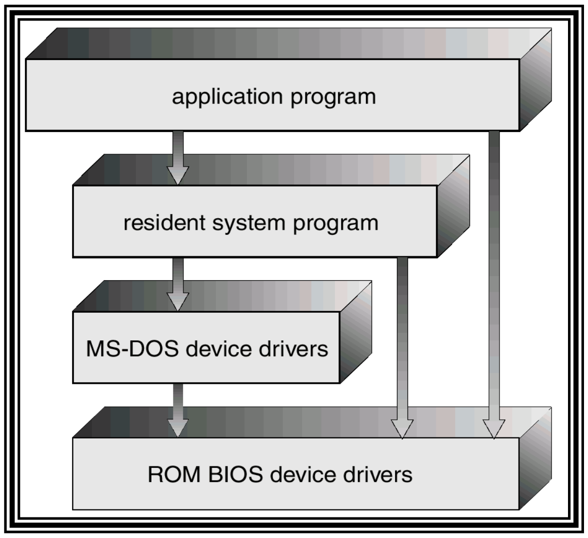
6. System Structure

- A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components rather than have one monolithic system.
- Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions. As modern operating systems are large and complex careful engineering is required.
- There are four different structures that have shown in this document in order to get some idea of the spectrum of possibilities. These are by no means exhaustive, but they give an idea of some designs that have been tried in practice.

Simple Structure

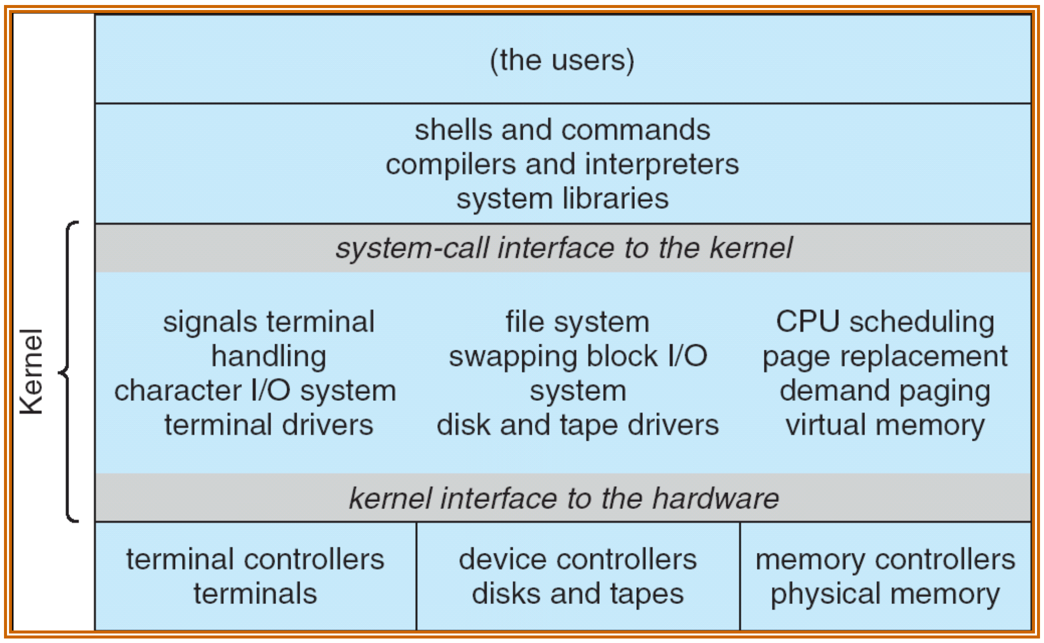
Many commercial systems do not have well-defined structures. Frequently,such operating systems started as small, simple, and limited systems and then grew beyond their original scope. **MS-DOS** is an example of such a system. It was originally designed and implemented by a few people who had no idea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

- MS-DOS
  - was written to provide the most functionality in the least space(because of the limited hardware on which it ran),
  - so it was not divided into modules carefully.
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated.



MS-DOS layer structure

- UNIX
  - UNIX is the another system limited by hardware functionality. It consists of two separable parts.
    - Systems programs
    - The kernel
      - 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.
      - 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.
      - Taken in sum, that is an enormous amount of functionality to be combined into one level.
- This monolithic structure of UNIX was difficult to implement and maintain i.e., changes in one system could adversely effect other areas.



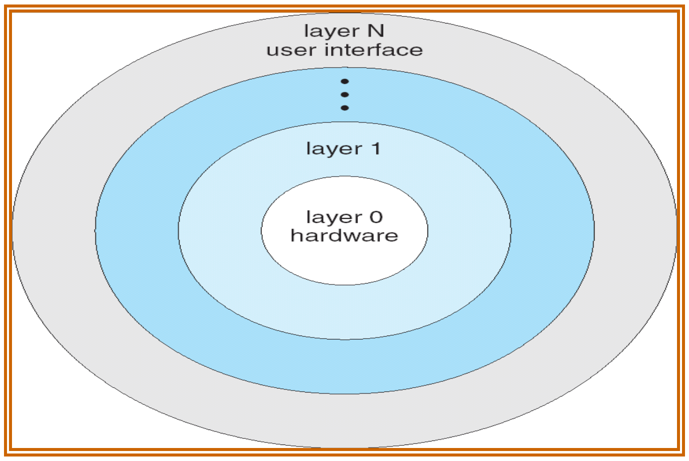
UNIX System Structure

- New versions of UNIX are designed to use more advanced hardware.
- With proper hardware support, operating systems can be broken into pieces that are smaller and more appropriate than those allowed by the original MS-DOS or UNIX systems.
- The operating system can then retain much greater control over the computer and over the applications that make use of that computer.
- Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems.
- Under the top-down approach, the overall functionality and features are determined and are separated into components.
- This seperation allows programmers to hide information,they are therefore free to implement the low-level routines as they see fit, provided that the external interface of the routine stays unchanged and that the routine itself performs the advertised task.

Layered Approach

- A system can be made modular in many ways. One method is the **layered approach,** in which the operating system is broken up into a number of layers(or levels), each built on top of lower layers.

- The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface as shown in below**.** With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers.



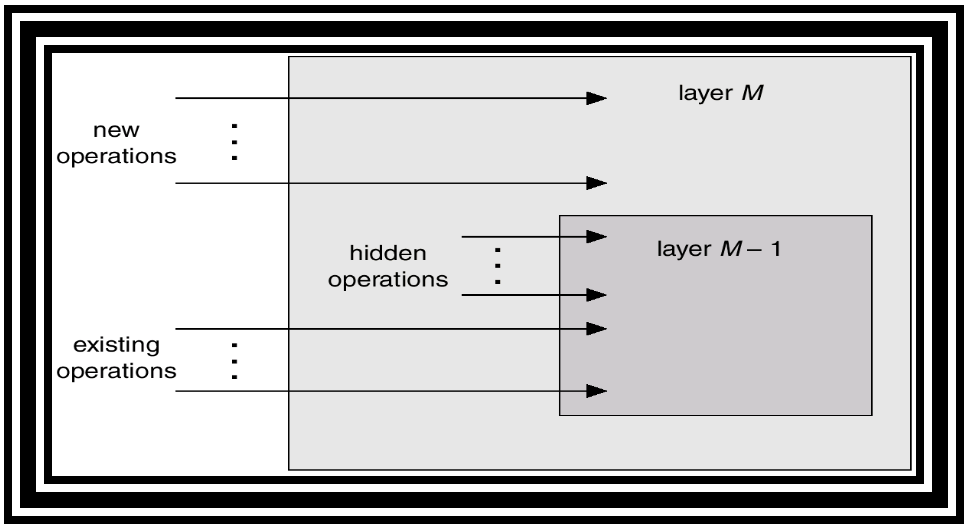
Layered OS

- A layered design was first used in the operating system. Its six layers are as follows:

layer 5:	**user programs**
layer 4:	**buffering for input and output**
layer 3:	**operator-console device driver**
layer 2:	**memory management**
layer 1:	**CPU scheduling**
layer 0:	**Hardware**

- An operating-system layer is an implementation of an abstract object made up of data, and of the operations that can manipulate those data.
- 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 **modularity** (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.



An Operating System Layer

- 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 is simplified when the system is broken down into layers.
- Each layer is implemented with only those operations provided by lowerlevel 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.

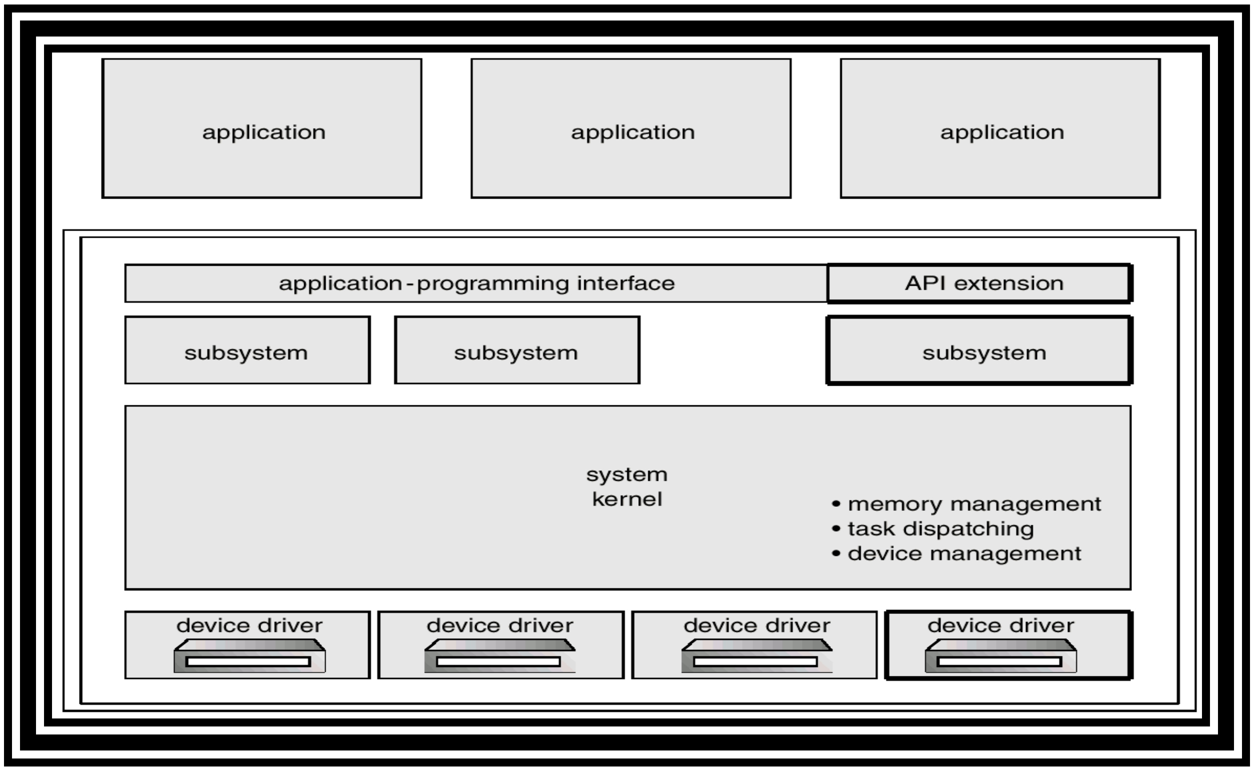
Drawbacks

- 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 disk space used by virtual-memory algorithms must be at a level lower than that of the memory-management routines,because memory management requires the ability to use the disk space.

- A final problem with layered implementations is that they tend to be less efficient than other types.
    - For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware. 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.

- These limitations have caused a small backlash against layering in recent years.
- Fewer layers with more functionality are being designed, providing most of the advantages of modularized code while avoiding the difficult problems of laver definition and interaction.

- For instance, OS/2 shown in fig(f) is a descendent of MS-DOS that adds multitasking and dual-mode operation,as well as other new features.



OS/2 Layer Structure

Microkernels

- We have already seen ,as the UNIX expanded, the kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called Mach that modularized the kernel using the microkernel approach. This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs i.e., moves as much from the kernel into "*user*" space which results is a smaller kernel.
- There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. However, microkernels typically 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.
- Communication takes place between user modules using *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.
- One benefit of the microkernel approach is ease of extending the operating system. 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. Several contemporary operating systems have used the microkernel approach.

Examples

- Tru64 UNIX (formerly Digital UNIX) provides a UNIX interface to the user, but it is implemented with a Mach kernel.
- QNX is a real-time operating system that is also based on the microkernel design. The QNX microkernel provides services for message passing and process scheduling.

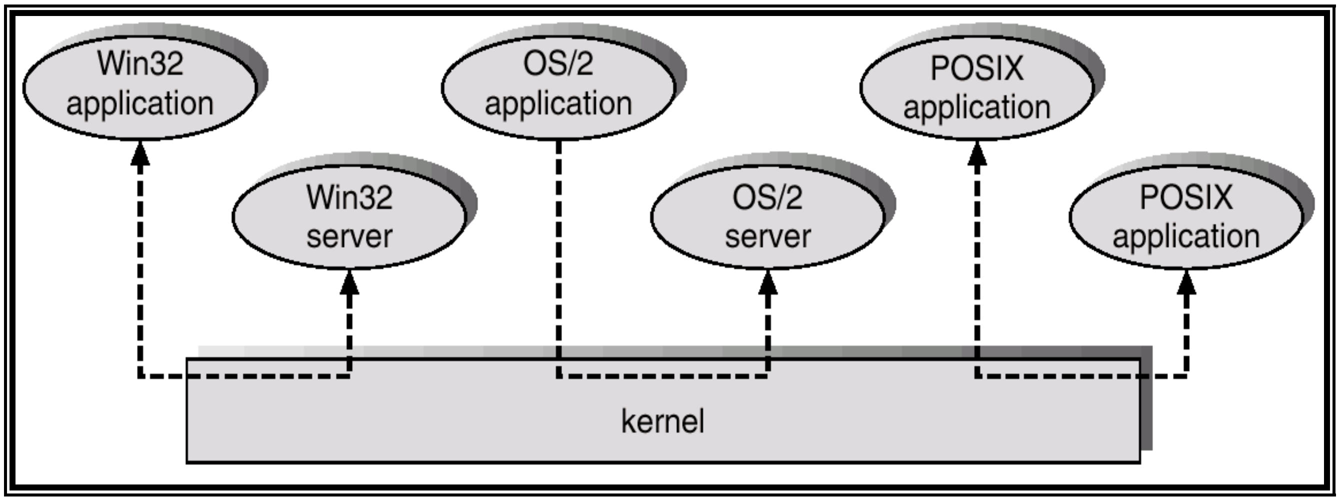
Benefits

- Easier to extend a microkernel

- Easier to port the operating system to new architectures
- More reliable (less code is running in kernel mode)
- More secure

Drawback

- 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.



Windows NT Client-Server Structure

7. Resources

- The OS treats an entity as a resource if it satisfies the below characteristics:
    - A process must request it from the OS.
    - A process must suspend its operation until the entity is allocated to it.
- The most common source is a file. A process must request a file before it can read it or write it. Further, if the file is unavailable,the process must wait until it becomes available. This abstract description of a resource is crucial to the way various entities(such as files,memory and devices) are managed.

Files

- A sequential file is a named,linear stream bytes of memory.
- You can store information by opeening a file

8. Process

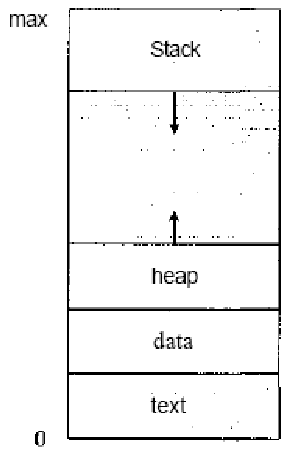
An operating system executes a variety of programs:

- Batch system – jobs
- Time-shared systems – user programs or tasks

Process – a program in execution; process execution must progress in sequential fashion. A process is a program in execution. A process is more than the program code, which is sometimes known as the **text section.** A process includes:

1. Program counter
2. Stack
3. Data section
    - The current activity, as represented by the value of the **Program counter** and the contents of the processor's registers.
    - The process **Stack** contains temporary data (such as function parameters, return addresses, and local variables),
    - **Data section,** which contains global variables.

- A process may also include a **heap,** which is memory that is dynamically allocated during process run time.



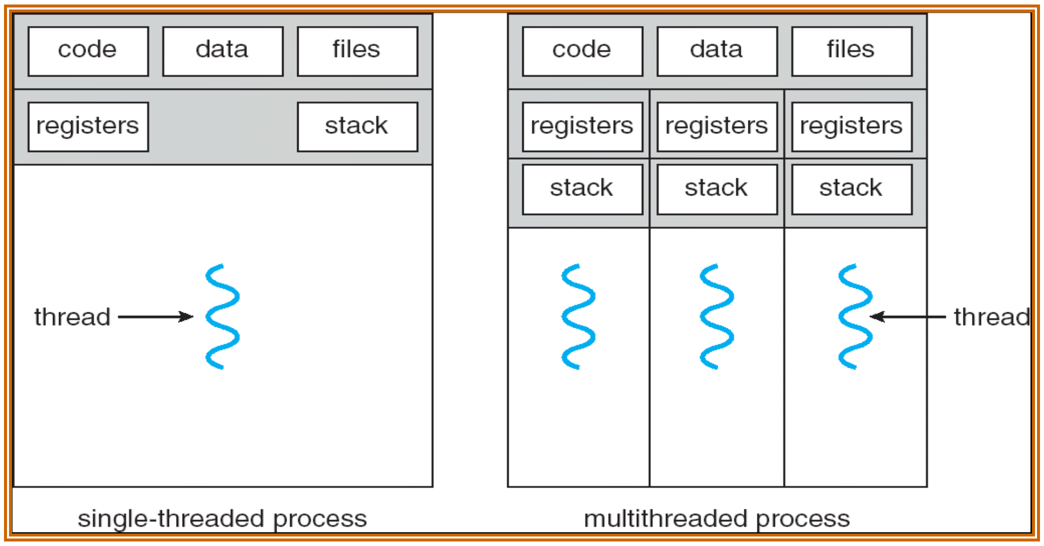
Process in memory

- A program by itself is not a process; a program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file). Whereas a process_is an active entity, with a program counter *s*pecifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.
- Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog. exe or a. out.) Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. Several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program.
- Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs.

9.Threads

A thread is a basic unit of CPU utilization. A thread,sometimes called as **light weight process** whereas a process is a **heavyweight process**.

- Thread comprises:
    - A thread ID
    - A program counter
    - A register set
    - A stack.
- A process is a program that performs a single thread of execution i.e., a process is a executing program with a single thread of control. For example, when a process is running a word-processor program, a single thread of instructions is being executed.
- This single thread of control allows the process to perform only one task at one time. For example, the user cannot simultaneously type in characters and run the spell checker within the same process.
- Traditionally a process contained only a single *thread* of control as it ran, many modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.



Single-threaded and multithreaded processes

- The operating system is responsible for the following activities in connection with process and thread management:
  - The creation and deletion of both user and system processes;
  - The scheduling of processes;
  - The provision of mechanisms for synchronization,
  - Communication,
  - Deadlock handling for processes.

Benefits

The benefits of multi threaded programming can be broken down into four major categories:

Responsiveness

Multi threading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. A multi threaded web browser could still allow user interaction in one thread while an image was being loaded in another thread.

Resource sharing

By default, threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

Economy

- Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads.
- Empirically gauging the difference in overhead can be difficult, but in general it is muc more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.

Utilization of multiprocessor architectures

- The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors.
- A singlethreaded process can only run on one CPU, no matter how many are available.
- Multithreading on a multi-CPU machine increases concurrency.

User and kernel threads

Threads may be provided either at the user level, for **user threads,** or by the kernel, for **kernel threads.**

- **User threads** are supported above the kernel and are managed without kernel support i.e., they are implemented by thread library at the user level.
    - The library porvides support for thread creation,scheduling,and management with no support from the kernel.
    - Because the kernel is unaware of user-level threads,all thread creation and scheduling are done in user space without the need for kernel intervention.
    - Therefore, user-level threads are generally fast to create and manage;they have drawbacks however. If the kernel is single-threaded,then any user-level thread perfroming a blocking system call will cause the entire process to block,even if other threads ae available to run within the application. User-thread libraries include POSIX **Pthreads,**Mach **C-threads**, and Solaris 2 **UI-threads**.

- **Kernel threads** are supported and managed directly by the operating system.
    - The kernel performs thread creation,scheduling,and management in kernel space. Because thread management is done by the operating system, kernel threads are

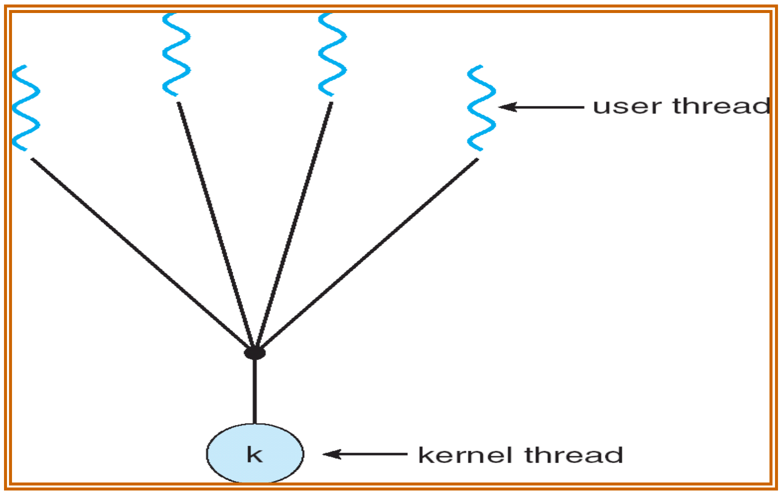
generally slower to create and manage than are user threads. However, since kernel is managing the threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execution.

- Also, in a multiprocessor environment, the kernel can schedule threads on different processors. Most contemporary operating systems—including Windows NT,Windows 2000,Solaris 2,BeOS, and Tru64 UNIX (formerly Digital UNIX)—support kernel threads.

Multithreading Models

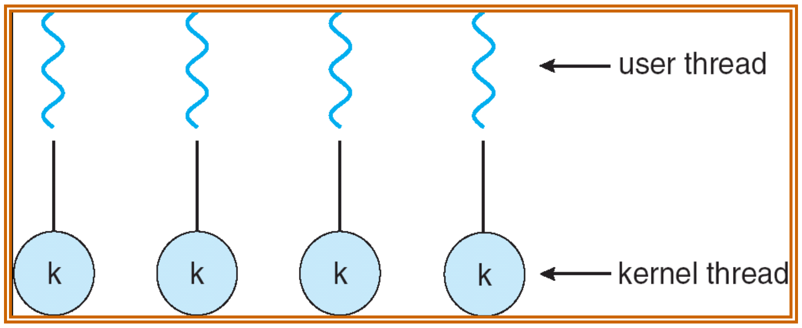
Many systems provide support for both user and kernel threads,resulting in different multi threading models. Three common ways of establishing this relationship are:

Many-to-One Model



- The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call.
- Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors. **Green threads**—a thread library available for Solaris 2—uses this model.
- In addition, user level thread libraries implemented on Operating systems that do not support kernel threads use the many-to-one model.

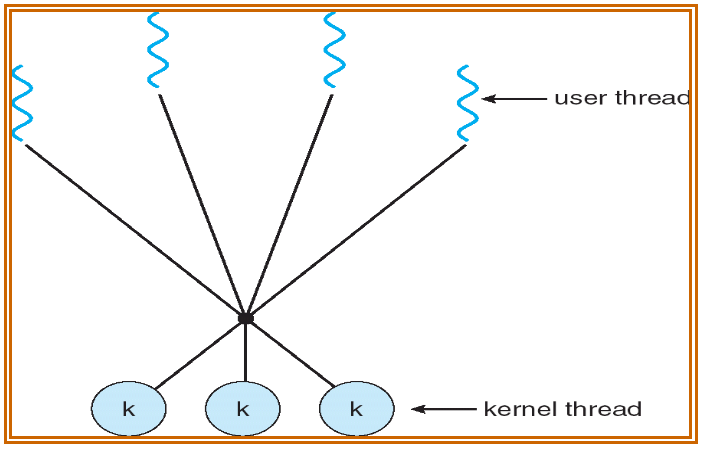
One-to-One Model



- The one-to-one model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.
- Linux, along with the family of Windows operating systems—including Windows 95, 98, NT, 2000, and OS/2— implement the one-to-one model.

Many-to-Many Model

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.



- The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a uniprocessor). Whereas the many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time.
- The one-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create).
- The many-to-many model suffers from neither of these shortcomings: Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution. Solaris 2, IRIX, HP-UX and Tru64 UNIX support this model.

Processes Vs Threads

How threads differ from processes

- **Threads** differ from traditional multitasking operating system processes:
  - processes are typically independent, while threads exist as subsets of a process. processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources
  - processes have separate address spaces, whereas threads share their address space. Processes interact only through system-provided inter-process communication mechanisms
  - Context switching between threads in the same process is typically faster than context switching between processes. As we mentioned earlier that in many respect threads operate in the same way as that of processes. Some of the similarities and differences are:

Similarities

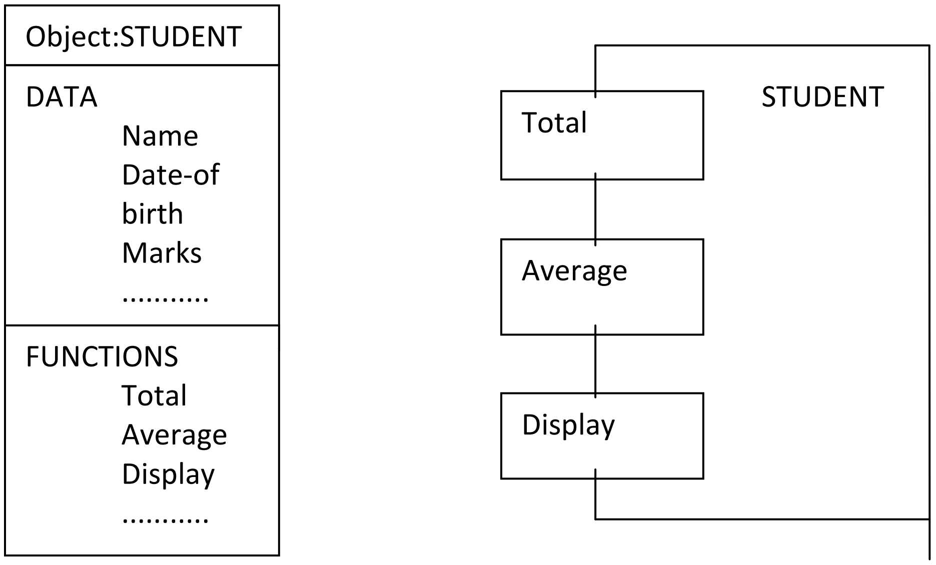
- Like processes threads share CPU and only one thread active (running) at a time.

- Like processes, threads within a processes, threads within a processes execute sequentially.
- Like processes, thread can create children.
- And like process, if one thread is blocked, another thread can run.

Differences

- Unlike processes, threads are not independent of one another.
- Unlike processes, all threads can access every address in the task .
- Unlike processes, thread are design to assist one other. Note that processes might or might not assist one another because processes may originate from different users.

10. **Objects** Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account,a table of data or any item that the program has to handle. Programming problem is analysed in terms of objects and the nature of communication between them. When a program is executed,the objects interact by sending messages to one another . For example,if 'customer' and 'account' are the two objects in a program,then the customer object may send a message to the account object. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each others data or code. It is sufficient to know the type of message accepted,and the type of response returned by the objects.



Two ways of representing an object

- Real-world objects share two characteristics: They all have *state* and *behavior*.
- Desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune).
- You may also notice that some objects, in turn, will also contain other objects.
- These real-world observations all translate into the world of object-oriented programming.

11. Device management

This component of operating system manages hardware devices via their respective drivers. The operating system performs the following activities for device management.

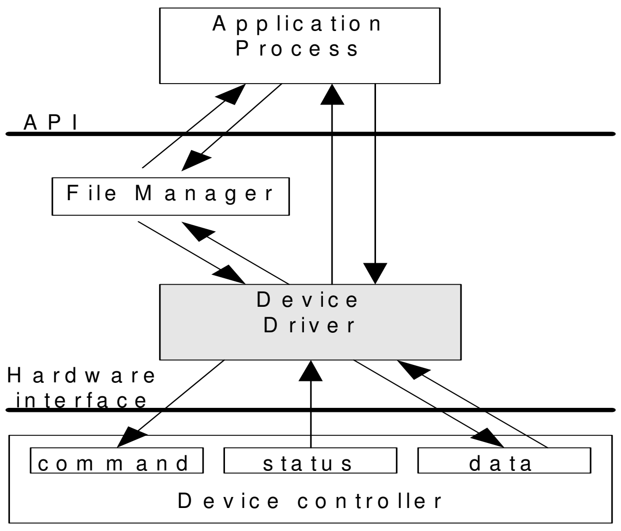
- It keeps track of all the devices. The program responsible for keeping track of all the devices is known as I/O controller.
- It provides a uniform interface to access devices with different physical characteristics.
- It allocates the devices in an efficient manner.
- It deallocates the devices after usage.
- It decides which process gets the device and how much time it should be used.
- It optimizes the performance of each individual device.

Approaches

- *Direct I/O* – CPU software explicitly transfer data to and from the controller's data registers
  - *Direct I/O with polling* – the device management software polls the device controller status register to detect completion of the operation; device management is implemented wholly in the *device driver*, if interrupts are not used
  - *Interrupt driven direct I/O* – interrupts simplify the software's responsibility for detecting operation completion; device management is implemented through the interaction of a *device driver* and interrupt routine
- *Memory mapped I/O* – device addressing simplifies the interface (device seen as a range of memory locations)
  - *Memory mapped I/O with polling* – the device management software polls the device controller status register to detect completion of the operation; device management is implemented wholly in the *device driver*.
  - *Interrupt driven I/O* – interrupts simplify the software's responsibility for detecting operation completion; device management is implemented through the interaction of a *device driver* and interrupt routine

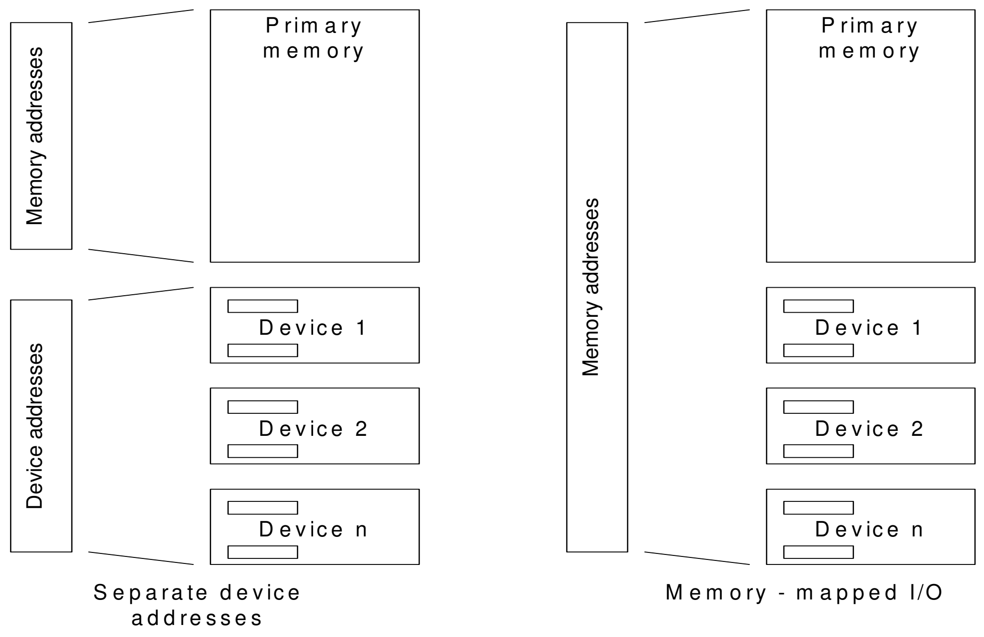
- *Direct memory access* – involves designing of hardware to avoid the CPU perform the transfer of information between the device (controller's data registers) and the memory.

I/O system organization



- An application process uses a device by issuing commands and exchanging data with the device management (device driver).
- Device driver responsibilities:
  - Implement communication APIs that abstract the functionality of the device
  - Provide device dependent operations to implement functions defined by the API
- API should be similar across different device drivers, reducing the amount of info an application programmer needs to know to use the device
- Since each device controller is specific to a particular device, the device driver implementation will be device specific, to
  - Provide correct commands to the controller
  - Interpret the controller status register (CSR) correctly
  - Transfer data to and from device controller data registers as required for correct device operation

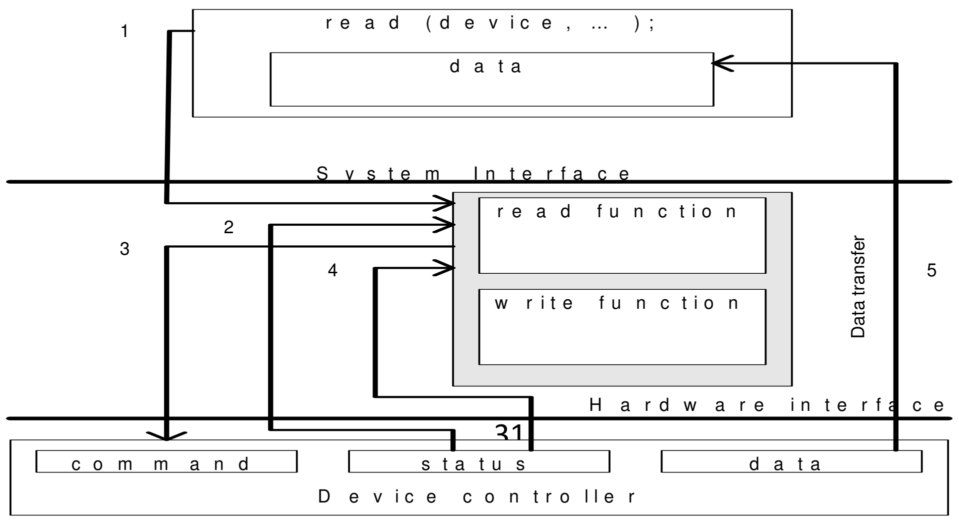
Direct I/O versus memory mapped I/O



I/O with polling

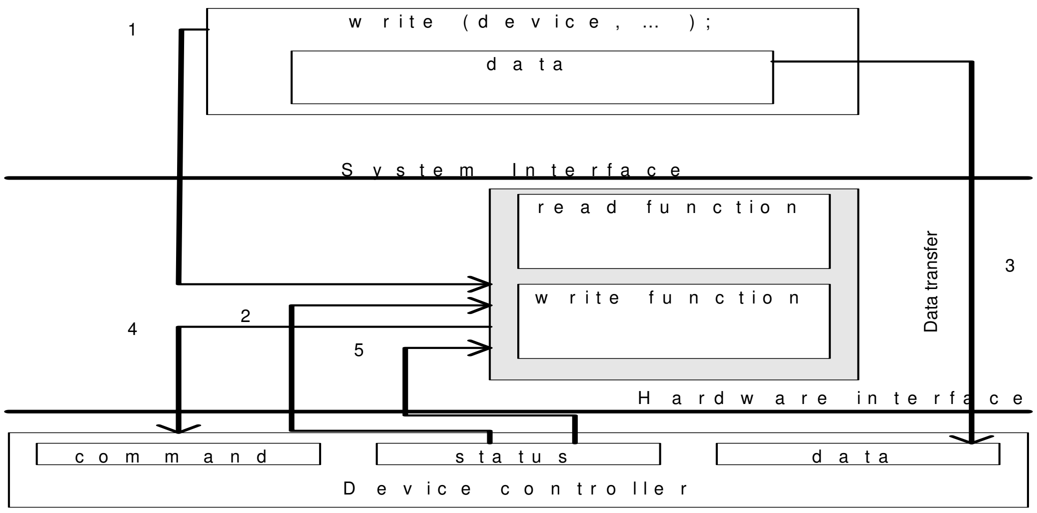
- Each I/O operation requires that the software and hardware coordinate their operations to accomplish desired effect
- In direct I/O pooling this coordination is done in the device driver;
- While managing the I/O, the device manager will poll the *busy/done* flags to detect the operation's completion; thus, the CPU starts the device, then polls the CSR to determine when the operation has completed
- With this approach is difficult to achieve high CPU utilization, since the CPU must constantly check the controller status;

I/O with polling – read



1. Application process requests a read operation
2. The device driver queries the CSR to determine whether de device is idle; if device is busy, the driver waits for it to become idle
3. The driver stores an input command into the controller's command register, thus starting the device
4. The driver repeatedly reads the content of CSR to detect the completion of the read operation
5. The driver copies the content of the controller's data register(s) into the main memory user's process's space.

I/O with polling – write



1. The application process requests a write operation

2. The device driver queries the CSR to determine if the device is idle; if busy, it will wait to become idle

3. The device driver copies data from user space memory to the controller's data register(s)

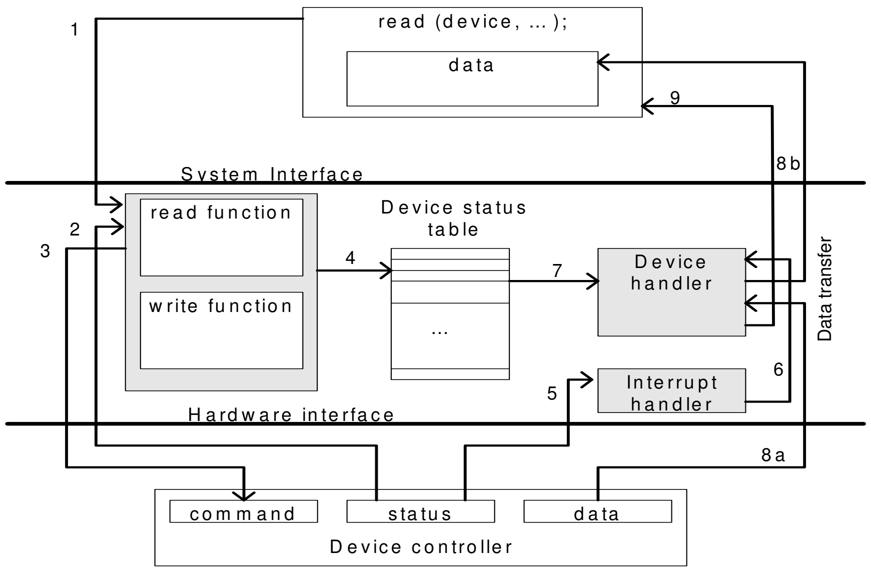
4. The driver stores an output command into the command register, thus starting the device

5. The driver repeatedly reads the CSR to determine when the device completed its operation.

Interrupt driven I/O

- In a multiprogramming system the wasted CPU time (in polled I/O) could be used by another process; because the CPU is used by other processes in addition to the one waiting for the I/O operation completion, in multiprogramming system may result a sporadic detection of I/O completion; this may be remedied by use of interrupts

- The reason for incorporating the interrupts into a computer hardware is to eliminate the need for a device driver to constantly poll the CSR
- Instead polling, the device controller "automatically" notifies the device driver when the operation has completed.



1. The application process requests a read operation
2. **T**he device driver queries the CSR to find out if the device is idle; if busy, then it waits until the device becomes idle
3. **T**he driver stores an input command into the controller's command register, thus starting the device
4. When this part of the device driver completes its work, it saves information regarding the operation it began in the device status table; this table contains an entry for each device in system; the information written into this table contains the return address of the original call and any special parameters for the I/O operation; the CPU, after is doing this, can be used by other program, so the device manager invokes the scheduler part of the process manager. It then terminates

5. The device completes the operation and interrupts the CPU, therefore causing an *interrupt handler* to run
6. The interrupt handler determines which device caused the interrupt; it then branches to the *device handler* for that device
7. The device driver retrieves the pending I/O status information from the device status table
8. **(a,b) T**he device driver copies the content of the controller's data register(s) into the user process's space
9. the device handler returns the control to the application process (knowing the return address from the device status table). Same sequence (or similar) of operations will be accomplished for an output operation.

Overlapping CPU execution with device operation

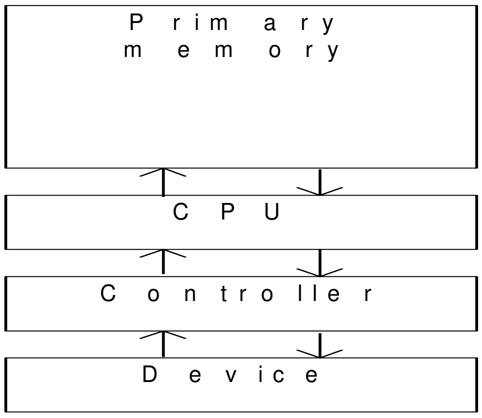
- The software interface to an I/O device usually enables the operating system to execute alternative processes when any specific process is waiting for I/O to complete, ***while preserving serial execution semantics for an individual process***
- That means that whenever the programmers will use *read* statement in a program, they will know that the next instruction will not execute until the read instruction has completed.

Consider the following code:

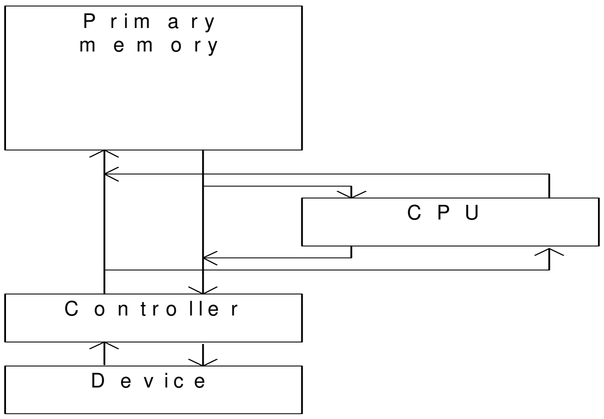
```
…
read (device, "%d", x);
y=f(x);
….
```

Direct memory access

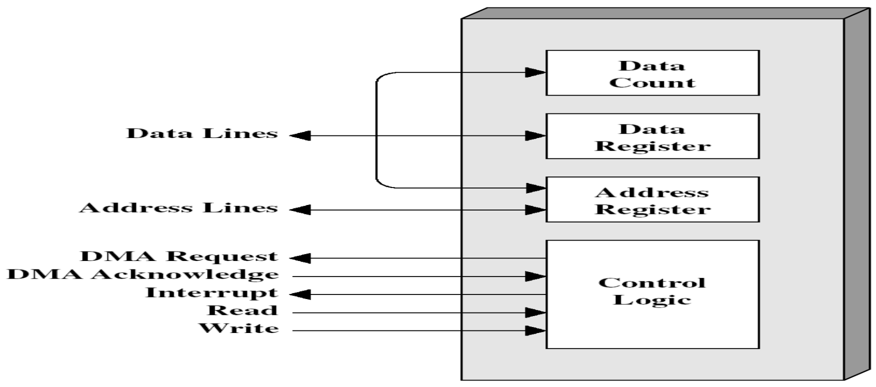
- Traditional I/O
  - Polling approach:
    - CPU transfer data between the controller data registers and the primary memory
    - Output operations - device driver copies data from the application process data area to the controller; vice versa for input operations
  - Interrupt driven I/O approach - the interrupt handler is responsible for the transfer task



- DMA controllers are able to read and write information directly from /to primary memory, with no software intervention
- The I/O operation has to be initiated by the driver
- DMA hardware enables the data transfer to be accomplished without using the CPU at all
- The DMA controller must include an address register (and address generation hardware) loaded by the driver with a pointer to the relevant memory block; this pointer is used by the DMA hardware to locate the target block in primary memory

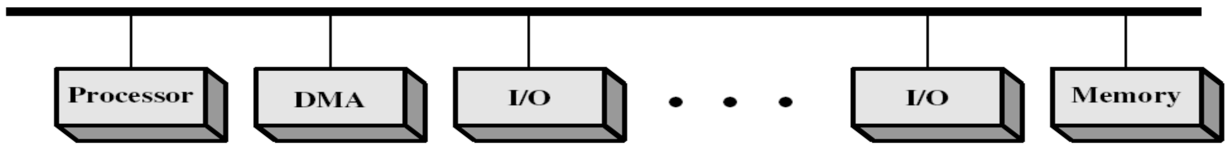


Typical DMA

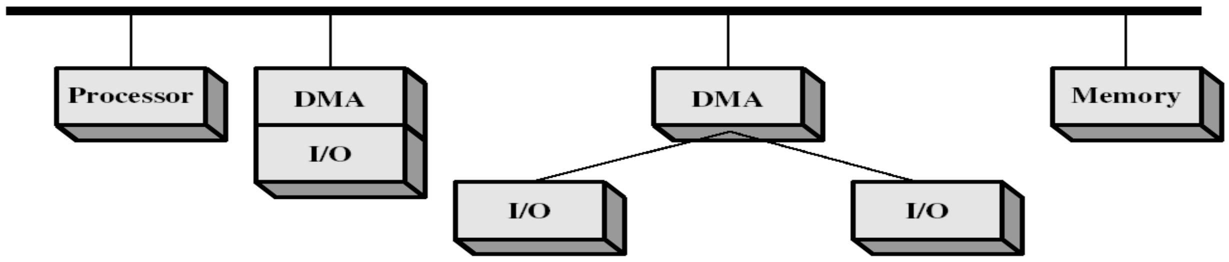


- Mimics the processor. Transfers data to/from memory over system bus

Alternative DMA configurations



(a) Single-bus, detached DMA



(b) Single-bus, Integrated DMA-I/O

12. Buffering

Buffering is a technique by which a device manager keeps the slower I/O devices busy when a process is not requiring I/O operations.

- **Input buffering** is the process of reading the data into the primary memory before the process requests it.
- **Output buffering** is the process of saving the data in the memory and then writing it to the device while the process continues its execution.

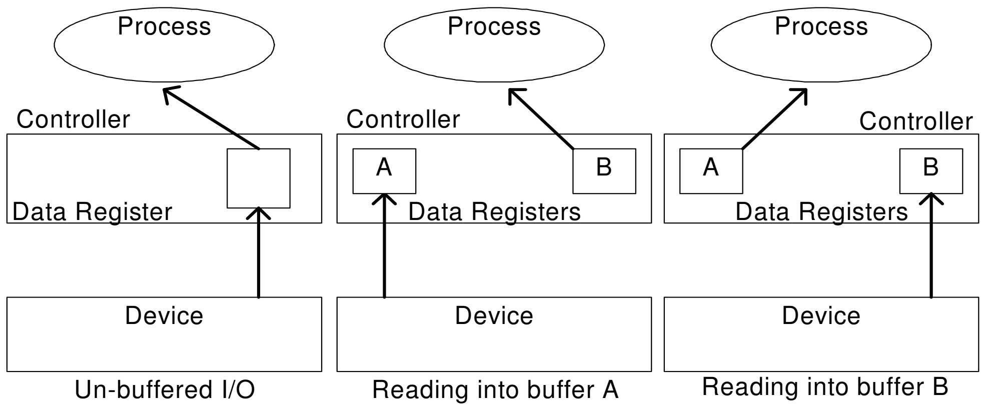
Hardware level buffering

Consider a simple character device controller that reads a single byte form a modem for each input operation.

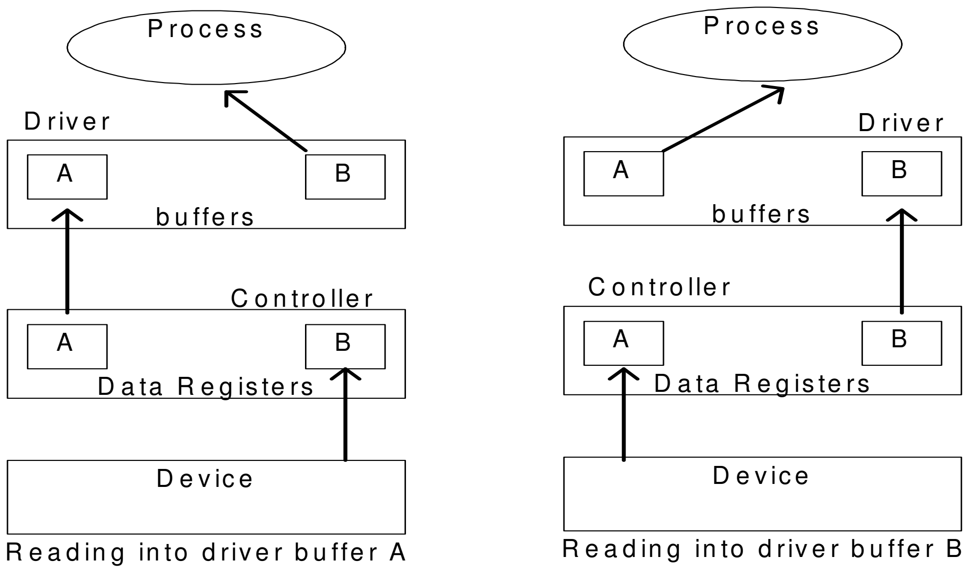
– *Normal operation*: read occurs, the driver passes a read command to the controller; the controller instructs the device to put the next character into one-byte data controller's register; the process calling for byte waits for the operation to complete and the retrieves the character from the data register

Add a hardware buffer to the controller to decrease the amount of time the process has to wait

– *Buffered operation*: the next character to be read by the process has already been placed into the data register, even the process has not yet called for the read operation

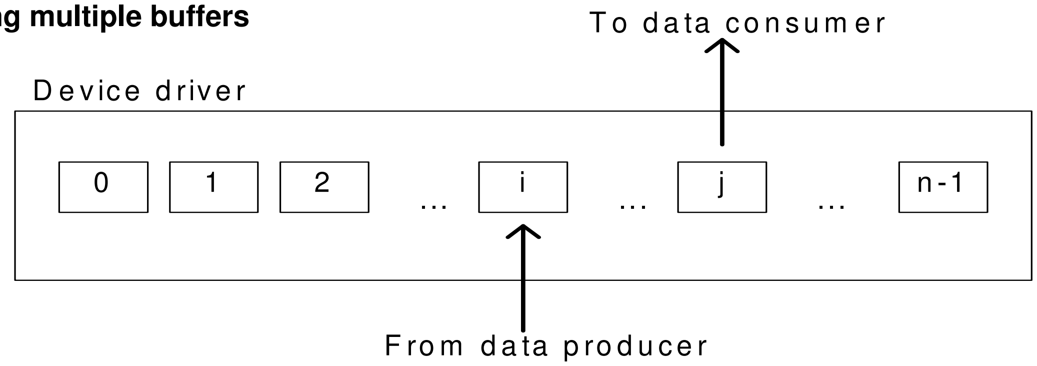


Driver level buffering



This is generally called double buffering. One buffer is for the driver to store the data while waiting for the higher layers to read it. The other buffer is to store data from the lower level module. This technique can be used for the block-oriented devices (buffers must be large enough to accommodate a block of data).

Using multiple buffers



- The number of buffers is extended from two to n. The data producer (controller in read operations, CPU in write operations) is writing into buffer i while the data consumer (the CPU in read operations, the controller in write operations) is reading form buffer j.
- In this configuration buffers j+1 to n-1 and 0 to i-1 are full. This is known as *circular buffering* technique.

13. Device driver

- It is a software program that controls a particular type of device attached to the computer. It provides an interface to the hardware devices without the requirement to know the precise information about the hardware.
- A device driver communicates with the device through a bus or communication sub system.

Responsibilities

1. Initialize devices
2. Interpreting the commands from the operating system
3. Manage data transfers
4. Accept and process interrupts
5. Maintain the integrity of driver and kernel data structures

Device driver interface

- Each operating system defines an architecture for its device management system. The designs are different from operating system to operating system; there is no universal organization
- Each operating system has two major interfaces to the device manager:
  - The driver API
  - The interface between a driver and the operating system kernel.

Driver API

- Provides a set of functions that an programmer can call to manage a device (usually comms or storage). The device manager
  - Must track the state of the device: when it is idle, when is being used and by which process
  - Must maintain the information in the device status table
  - May maintain a device descriptor to store other information about the device
- *open/close* functions to allow initiate/terminate of the device's use
  - *open* – allocates the device and initializes the tables and the device for use

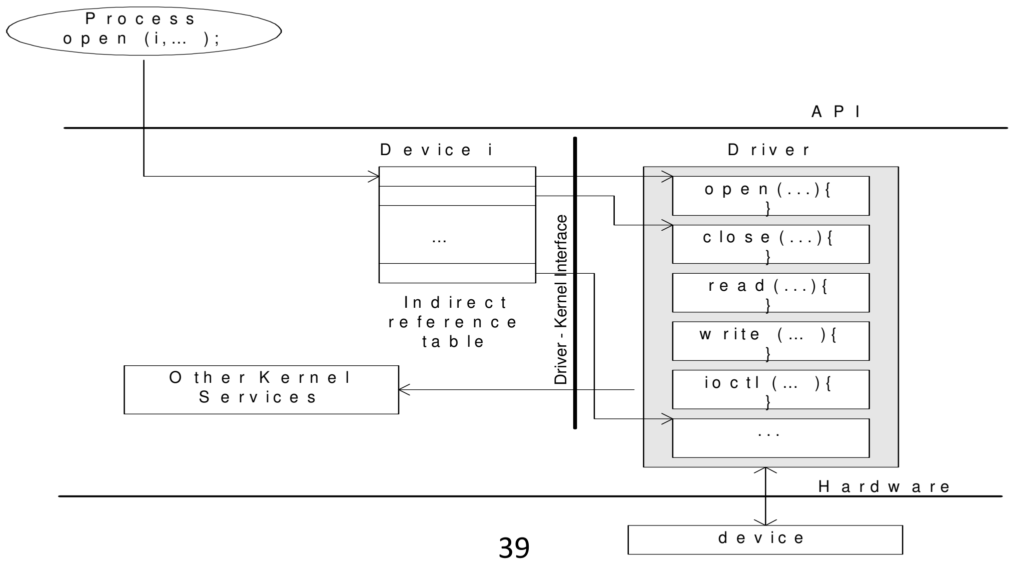
  - *close* – releases dynamic tables entries and releases the device
- *read/write* functions to allow the programmer to read/write from/to the device
  - A consistent way to do these operations across all the devices is not possible; so a concession by dividing the devices into *classes* is made:
    - o such as character devices or block devices
    - o Sequential devices or randomly accessed devices
- *ioctl* function to allow programmers to implement device specific functions.

The driver - kernel interface

The device driver must execute privileged instructions when it starts the device; this means that the device driver must be executed as part of the operating system rather than part of a user program. The driver must also be able to read/write info from/to the address spaces of different processes, since same device driver can be used by different processes

- Two ways of dealing with the device drivers
  - Old way: driver is part of the operating system, to add a new device driver, the whole OS must have been complied
  - Modern way: drivers installation is allowed without re-compilation of the OS by using reconfigurable device drivers; the OS dynamically binds the OS code to the driver functions.
    - o A reconfigurable device driver has to have a fixed, standardized API
    - o The kernel has to provide an interface to the device driver to allocate/de-alocate space for buffers, manipulate tables in the kernel, etc.

Reconfigurable device drivers



- The OS uses an indirect reference table to access the different driver entry points, based on the device identifier and function name. The indirect reference table is filled with appropriate values whenever the device driver loads (because the detection of a PnP device or at boot time)
- When a process performs a system call, the kernel passes the call onto the device driver via the indirect reference table.