1. Computer system Operation

1.1 CPU and Process Management

Every program running on a computer, be it a service or an application, is a process. As long as a von Neumann architecture is used to build computers, only one process per CPU can be run at a time. Older microcomputer Operating Systems such as MS-DOS did not attempt to bypass this limit, with the exception of interrupt processing, and only one process could be run under them. Mainframe operating systems have had multitasking capabilities since the early 1960s. Modern operating systems enable concurrent execution of many processes at once via multitasking even with one CPU. Process management is an operating system's way of dealing with running multiple processes. Since most computers contain one processor with one core, multitasking is done by simply switching processes quickly. Depending on the operating system, as more processes run, either each time slice will become smaller or there will be a longer delay before each process is given a chance to run. Process management involves computing and distributing CPU time as well as other resources. Most operating systems allow a process to be assigned a priority which affects its allocation of CPU time. Interactive operating systems also employ some level of feedback in which the task with which the user is working receives higher priority. Interrupt driven processes will normally run at a very high priority. In many systems there is a background process, such as the System Idle Process in Windows, which will run when no other process is waiting for the CPU.

1.2 Memory Management

Current computer architectures arrange the computer's memory in a hierarchical manner, starting from the fastest registers, CPU cache, random access memory and disk storage. An operating system's memory manager coordinates the use of these various types of memory by tracking which one is available, which is to be allocated or de-allocated and how to move data between them. This activity, usually referred to as virtual memory management, increases the amount of memory available for each process by making the disk storage seem like main memory. There is a speed penalty associated with using disks or other slower storage as memory.

Another important part of memory management is managing virtual addresses. If multiple processes are in memory at once, they must be prevented from interfering with each other's memory (unless there is an explicit request to utilize shared memory). This is achieved by having separate address spaces. Each process sees the whole virtual address space, typically from address 0 up to the maximum size of virtual memory, as uniquely assigned to it. The operating system maintains a page table that matches virtual addresses to physical addresses. These memory allocations are tracked so that when a process terminates, all memory used by that process can be made available for other processes.

The operating system can also write inactive memory pages to secondary storage. Under Microsoft Windows, this process is called *paging*.

1.3 I/O Management

Any Input/Output (I/O) devices present in the computer, such as keyboard, mouse, disk drives, printers, displays, etc require a significant amount of management. The Operating System allocates requests from applications to perform I/O to an appropriate device and provides convenient methods for using the device (typically abstracted to the point where the application does not need to know implementation details of the device). To perform useful functions, processes need access to the peripherals connected to the computer, which are controlled by the kernel through device drivers. For example, to show the user something on the screen, an application would make a request to the kernel, which would forward the request to its display driver, which is then responsible for actually plotting the character/pixel.

A kernel must maintain a list of available devices. This list may be known in advance (e.g. on an embedded system where the kernel will be rewritten if the available hardware changes), configured by the user (typical on older PCs and on systems that are not designed for personal use) or detected by the operating system at run time (normally called plug and play).

1.4 Information and Storage Management

All operating systems include support for a variety of file systems. Modern file systems comprise a hierarchy of directories. While the idea is conceptually similar across all general-purpose file systems, some differences in implementation exist. 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. File systems may provide journaling, which provides safe recovery in the event of a system crash. A journaled file system writes information twice: first to the journal, which is a log of file system operations, then to its proper place in the ordinary file system. In the event of a crash, the system can recover to a consistent state by replaying a portion of the journal. In contrast, non-journaled file systems typically need to be examined in their entirety by a utility such as *fsck* or *chkdsk*.

1.5 Network Management

Although not a core part of the operating system, the Network Manager has become essential in modern day computing. Most current operating systems are capable of using the TCP/IP networking protocols. This means that one system can appear on a network of the other and share resources such as files, printers, and scanners using either wired or wireless connections.

Many operating systems also support one or more vendor-specific legacy networking protocols as well, for example, SNA on IBM systems, DECnet on systems from Digital Equipment Corporation, and Microsoft-specific protocols on Windows. Specific protocols for specific tasks may also be supported such as NFS for file access.

1.6 User Interface

All operating systems need to provide an interface to communicate with the user. This could be a Command Line Interface or a Graphical User Interface.

A **command line interface** or **CLI** is a method of interacting with an operating system or software using a command line interpreter. This command line interpreter may be a text terminal, terminal emulator, or remote shell client. The concept of the CLI originated when teletype machines (TTY) were connected to computers in the 1950s, and offered results on demand, compared to ‘batch’ oriented mechanical punch card input technology. Dedicated text-based CRT terminals followed, with faster interaction and more information visible at one time, and then graphical terminals enriched the visual display of information. Currently personal computers encapsulate both functions in software.

A **graphical user interface** (**GUI**) is a type of user interface which allows people to interact with a computer and computer-controlled devices which employ graphical icons, visual indicators or special graphical elements called *widgets,* along with text, labels or text navigation to represent the information and actions available to a user. The actions are usually performed through direct manipulation of the graphical elements. Today, most modern operating systems contain GUIs. A few older operating systems tightly integrated the GUI to the kernel—for example, the original implementations of Microsoft Windows and Mac OS the Graphical subsystem was actually part of the operating system. More modern operating systems are modular, separating the graphics subsystem from the kernel (as is now done in Linux and Mac OS X) so that the graphics subsystem is not part of the OS at all.

1.7 Program Execution

The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating an error). This involves locating the executable file on the disk or other secondary storage media and loading its content into the memory. These steps may further include processing by another parser or interpreter as in the case of .NET Platform, in which each program is compiled to MSIL (*Microsoft Intermediate Language*, now called CIL or *Common Intermediate Language*) and then parsed to assembly upon execution by the .NET JIT (*Just In Time Compiler)*.

1.8 Security

There are two generic levels of security, internal and external. Internal security can be thought of as protecting the computer's resources from the programs concurrently running on the system. Most operating systems set programs running natively on the computer's processor, so the problem arises of how to stop these programs doing the same task and having the same privileges as the operating system (which is after all just a program too). Processors used for general purpose operating systems generally have a hardware concept of privilege. Generally less privileged programs are automatically blocked from using certain hardware instructions, such as those to read or write from external devices like disks. Instead, they have to ask the privileged program (operating system kernel) to read or write. The operating system therefore gets the chance to check the program's identity and allow or refuse the request.

Typically an operating system offers (or hosts) various services to other network computers and users. These services are usually provided through ports or numbered access points beyond the operating systems network address. Services include offerings such as file sharing, print services, email, web sites, and file transfer protocols (FTP), most of which can have compromised security. These threats are categorized under external threats and are usually dealt with using add-on software like firewalls and antivirus programs.

1.8 Device Management

To perform useful functions, processes need access to the peripherals connected to the computer, which are controlled by the kernel through device drivers. For example, to show the user something on the screen, an application would make a request to the kernel, which would forward the request to its display driver, which is then responsible for actually plotting the character/pixel.

In a plug and play system, a device manager first performs a scan on different hardware buses, such as Peripheral Component Interconnect (PCI) or Universal Serial Bus (USB), to detect installed devices, then searches for the appropriate drivers.

As device management is a very OS-specific topic, these drivers are handled differently by each kind of kernel design, but in every case, the kernel has to provide the I/O to allow drivers to physically access their devices through some port or memory location. Very important decisions have to be made when designing the device management system, as in some designs accesses may involve context switches, making the operation very CPU-intensive and easily causing a significant performance overhead.

1.9 Resource Allocation and Accounting

When multiple users or multiple jobs running are concurrently on the operating system, resources must be allocated to each of them. Some (such as CPU cycles, main memory, and file storage) may have special allocation code and rules, while others (such as I/O devices) may have general request and release code. To keep track of which users use how much and what kinds of computer resources, the OS should also implement an Accounting scheme.

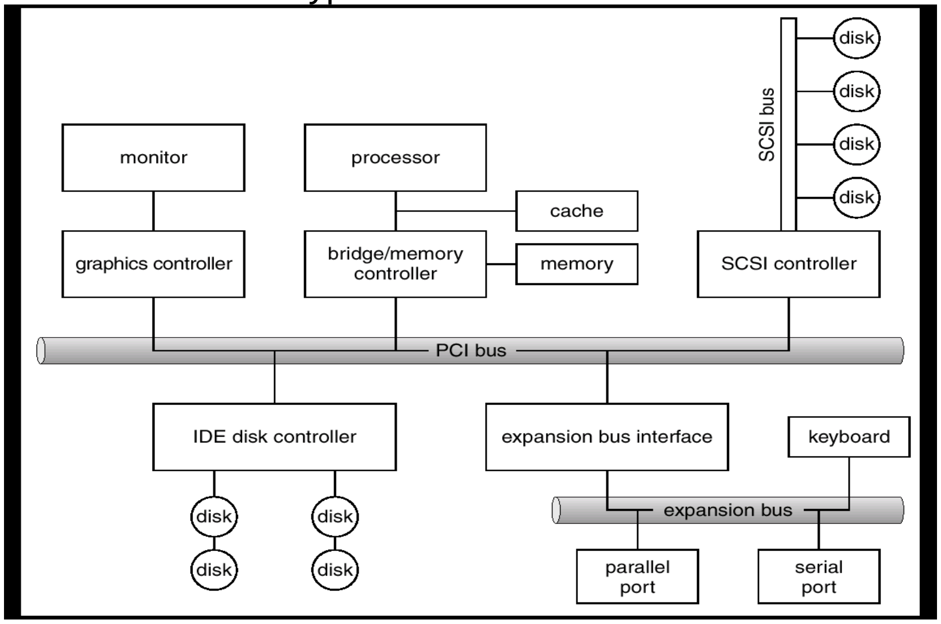
2. Computer-System Operation

I/O devices and the CPU can execute concurrently.
Each device controller is in charge of a particular device type.
Each device controller has a local buffer.
CPU moves data from/to main memory to/from local buffers
I/O is from the device to local buffer of controller.
Device controller informs CPU that it has finished its operation by causing an *interrupt*.

3. I/O structure

Port: a connection point between a peripheral device and the computer.
Bus: a set of wires shared by one or more devices, which communicate with the system using a rigid protocol.
Daisy chain: every device has two ports; either one port connects directly to the system and the other to another device, or the two ports connect to other devices. The chain usually operates as a bus.

A Typical PC Bus Structure



I/O devices such as hard disks write to memory all the time, and with a normal system setup, their requests go through to the CPU first and then the CPU reads/writes the data from/to the memory sequentially. When such a request involves polling/busy wait loops, it is known as **programmed I/O**. The problem with polling is that it wastes a lot of CPU resources as it sits in a tight loop checking for changes in values. The second method for I/O devices to access memory is called interrupt-driven. The problem now is that servicing interrupts is expensive, in the sense that when an interrupt signal is sent by an I/O device, before or after a memory access, the following has to be done every time:

- The data for the currently running process in the CPU (PC/SR/etc...) is to be saved to the stack.
- An Interrupt Service Routine is allowed to handle the interrupt
- Data is sent back to the I/O controller
- The data for the previously running process is restored from the stack.

Such a method has a very large overhead. The state information for the current process has to be saved before the I/O operation can use the CPU, and then restored after it has finished.

A much better way of doing things would be to take away the CPU from the picture all together and have the I/O devices talking directly to the memory. This is where DMA comes in. The DMA controller sits on the shared system bus containing the memory and CPU and allows up to 7 I/O devices to connect to it. Once it has all the information it needs about a particular I/O device, such as the number of words (word = 16 bits) to transfer and the memory address of the first word of input or output (Memory Address Counter), the DMA controller then transmits the data directly to the memory via the shared system bus.

Advantages of DMA

The most important is that the processor does not have to worry about I/O operations between computer peripherals and memory. Another advantage is the fact that transfers are much simpler since they do not require the CPU to execute specific instructions to do the transfer or have to deal with interrupts being signaled in from I/O devices. All the DMA controller needs is a start indicator to let it know when it can start transferring data to memory, a counter for how many words are left and a stop indicator to clean up at the end.

DMA Controllers and Channels

The DMA **controller** is built into a system's chipset and connects a single I/O device to memory. To lower system costs, a DMA controller may allow multiple I/O devices to connect to it. IBM calls this type of controller a **channel** (Dowsing et. al., 2000). When an I/O device wants to connect to the DMA controller, it does so through one of 7 channels available on the controller (confused?). There are actually 8 channels (0-7), but channel 4 is reserved for the system.

There are two types of channels (remember, they're DMA controllers that allow multiple peripherals -I/O- devices to connect to them):

***Multiplexer*:** Connects to several low/medium speed devices **all at once**. Scans all devices in turn, collecting each device's data (memory address, word count) into a buffer.

***Selector*:** Connects to several high speed devices, **but only one at a time**. Stays with one device until data transfer is complete.

There is a good diagram of multiplexer and selector controllers on slide 8 here (these are not the notes of Ian Marshall).

How does the controller work?

Remember that the DMA controller, memory and CPU are all on a shared system bus. Therefore, the DMA controller (DMAC) and CPU cannot read/write to memory at the same time. There are two ways for a DMAC to access memory:

Cycle Stealing: When their is a conflict and both the CPU and DMAC want to access memory, the DMAC steals cycles from the CPU, thus stalling it, temporarily until it's done.

Bus Mastering: This is the start of a DMAC's formal method of transferring data between memory and some peripheral. It determines which peripheral, connected to one of its channels, may use the bus.

Before a DMA transfer can take place, the DMAC has to be configured/programmed by the CPU:

1. The CPU loads the device ID that will be transferring the data and its function (read/write).

2. It loads the **Memory Address Counter** of the DMAC with the start address in memory.

3. It loads the **Word Counter** of the DMAC with the transfer length.

4. It signals that it may proceed with the transfer.

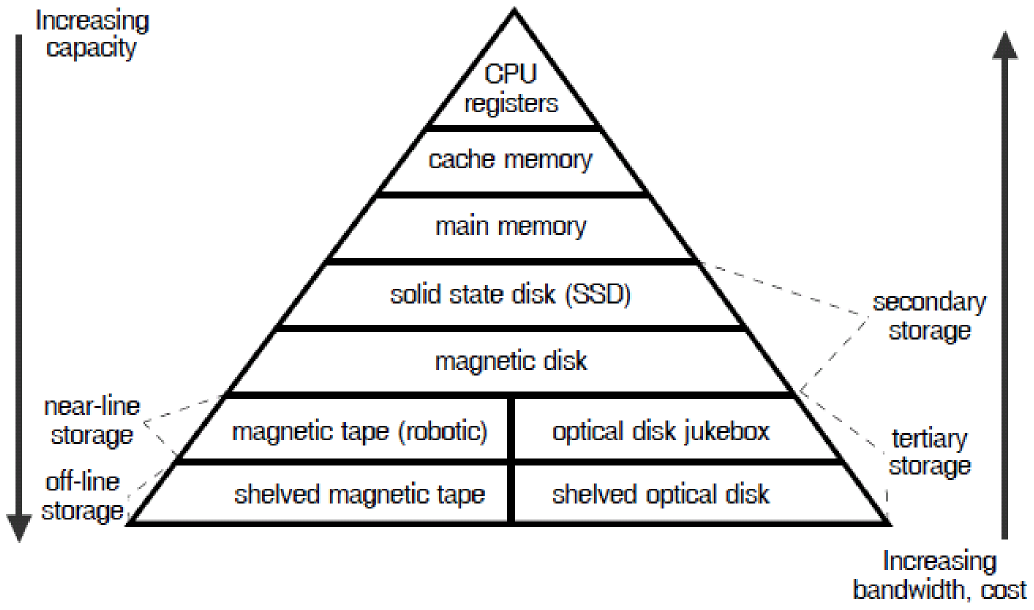
At this point, the CPU does not get involved at all until it receives an interrupt signal from the DMAC when it has finished its transfer. Memory accesses might also bump into each other on the system bus, in which case, the conflict is usually resolved in favor of the DMAC.

The DMAC loops until its Word Counter is 0. It transfers each word of data, decrementing the Word Counter and incrementing the Memory Access Counter each time.

4. Storage Devices

The first step in discussing storage systems is to describe the devices that such systems use. Figure shows the “storage pyramid,” which relates all the forms of storage to each other. Generally, capacity increases towards the bottom of the pyramid, while bandwidth is higher for storage at the top. Cost per byte is lowest for devices lower on the pyramid.

Data storage devices are classified into one of two categories depending on whether the device has easily interchangeable media. Magnetic disk and solid state disk (SSD) are termed *secondary storage* devices, and require one device per storage medium. A magnetic disk drive is an integrated unit; it is impossible to switch media on today’s disk drives. All other storage devices are called *tertiary storage* devices. Optical disks and magnetic and optical tapes are all tertiary storage devices, as a single read/write device can read any of a large number of storage media.



The storage pyramid

This pyramid shows the range of storage devices from CPU registers through shelved magnetic tape and optical disk. Devices and media at the bottom of the pyramid are considered tertiary storage devices, and have the lowest cost per byte and the longest access times. Secondary storage, such as magnetic disk, has a somewhat higher cost per megabyte and shorter access time. Tertiary storage is further divided into offline and nearline storage. Offline storage is accessible only via human operator intervention, while nearline storage is robotically managed. As a result, nearline storage is usually faster than offline storage. However nearline storage is more expensive, as it is difficult to build storage for many thousands of tapes that is robotically-accessible.

Tertiary storage is further broken down into *off-line* and *near-line* storage. Both types of tertiary storage use the same media, but they access those media in different ways. Near-line storage is available with very little latency, as the media are loaded by robots. The StorageTek Automated Cartridge System 4400 [48] is an example of such a system. Each tape silo stores 6,000 IBM 3490 (1/2 inch) tapes. Two robotic arms inside the silo may pick any tape and insert it into one of several IBM 3490 tape drives. Once the tape has been placed into a drive, the robotic arm is free to move other tapes. These arms can rapidly move a tape between drive and storage slot; an arm in the STK 4400 requires an average of 10 - 20 seconds to pick a single tape. The entire process of selecting a tape, moving it between its slot and the tape drive, and transferring data is automatic. No human need intervene in the process, keeping latencies low.

Off-line storage, on the other hand, is not accessible without human intervention. A mass storage center such as the National Center for Atmospheric Research (NCAR) [65] stores 25 terabytes or more, but only has sufficient tape silos for a fraction of that data. The remainder of the media are stored on rows of shelves in the data center and retrieved by the mass storage system's operators. Off-line accesses are considerably slower than nearline accesses for two reasons. First, human operators simply cannot move as quickly as robot arms. Second, humans may make mistakes both in picking the incorrect cartridge and in replacing a cartridge in the incorrect slot. For these reasons, off-line storage is slower than near-line storage. Nevertheless, data centers must store data off-line because it costs less to build shelves than buy robotic silos.

The remainder of this section discusses various types of secondary and tertiary storage.

Magnetic disk is the only type of secondary storage covered; while solid-state disk (SSD) is used in many mass storage systems, its characteristics are those of a large array of dynamic RAM. Tertiary storage media are based either on magnetic or optical technologies.

Since scientific computation centers use magnetic tape for tertiary storage far more than either optical tape or optical disk, it will be covered in more detail. Similarly, robots exist to manipulate both magnetic tape and optical disk, but tape robots dominate because of the prevalence of magnetic tape over optical technologies. The section concludes with a discussion of possible future directions for storage technologies including holographic optical storage.

5. 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. In handling the trap, the operating system will enter in the kernel mode, where it has access to privileged instructions, and can perform the desired service on the behalf of user-level process. It is because of the critical nature of operations that the operating system itself does them every time they are needed. 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.

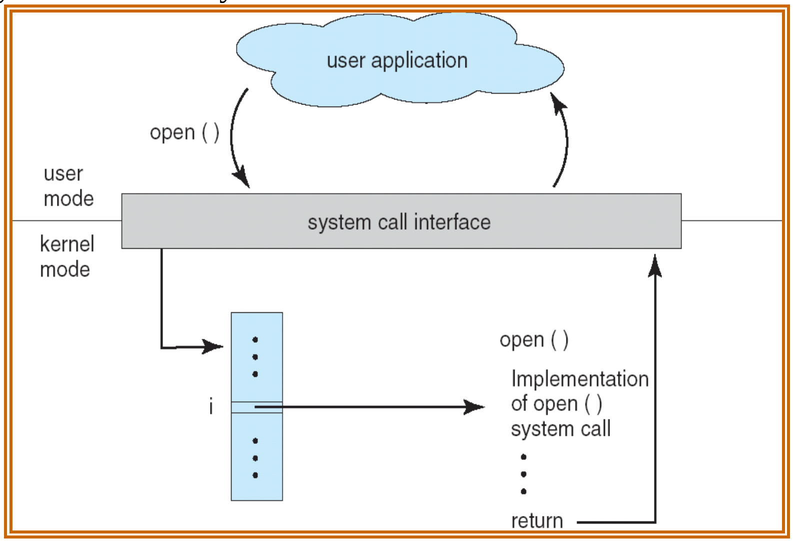
5.1 Unix System Call: The System Call Model

System Calls are requests by programs for services to be provided by the Operating System. The system call represents the interface of a program to the kernel of the Operating System

- The system call API (Application Program Interface) is given in C
- The system call is actually a wrapper routine typically consisting of a short piece of assembly language code.
- The system call generates a hardware trap, and passes the user's arguments and an index into the kernel's system call table for the service requested.
- The kernel processes the request by looking-up the index passed to see the service to perform, and carries out the request.
- Upon completion, the kernel returns a value which represents either successful completion of the request or an error. If an error occurs, the kernel sets a global variable, errno, indicating the reason for the error.
- The process is delivered the return value and continues its execution.

5.2 System Call Implementation

The system call interface intercepts function calls in the API and invokes the necessary system call within the operating system. Typically, a number associated with each system call when it is invoked by the system. And according to this number System-call interface maintains a table indexed which according to these numbers. The system call interface then invokes the intended system call in the Operating system kernel and returns the status of the system call and any return values.



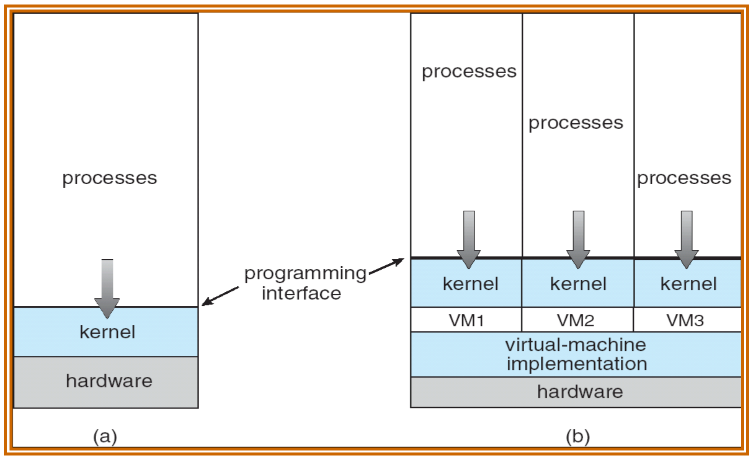
API – System Call – OS Relationship

5.3 Types of system call

- **Process control:** These types of system calls are used to control the processes. Some examples are end, abort, load, execute, create process, terminate process etc.
- **File management:** These types of system calls are used to manage files. Some examples are Create file, delete file, open, close, read, write etc.
- **Device management:** These types of system calls are used to manage devices. Some examples are Request device, release device, read, write, get device attributes etc.
- **Information maintenance:** These types of system calls are used to set system data and get process information. Some examples are time, OS parameters, id, time used etc.
- **Communications:** These types of system calls are used to establish a connection. Some examples are, send message, received messages, terminate etc.

6. Virtual machine

A **virtual machine** (**VM**) is a "completely isolated operating system installation within your normal operating system". These days this is implemented by either software emulation or hardware virtualization. A **system virtual machine** provides a complete system platform which supports the execution of a complete operating system (OS). In contrast, a **process virtual machine** is designed to run a single program, which means that it supports a single process. An essential characteristic of a virtual machine is that the software running inside is limited to the resources and abstractions provided by the virtual machine—it cannot break out of its virtual world.



6.1 System virtual machine

System virtual machines (sometimes called **hardware virtual machines**) allow the sharing of the underlying physical machine resources between different virtual machines, each running its own operating system. The software layer providing the virtualization is called a **virtual machine monitor** or hypervisor

The main advantages of VMs are:

- multiple OS environments can co-exist on the same computer, in strong isolation from each other
- the virtual machine can provide an instruction set architecture (ISA) that is somewhat different from that of the real machine
- application provisioning, maintenance, high availability and disaster recovery

The main disadvantages of VMs are:

- a virtual machine is less efficient than a real machine when it accesses the hardware indirectly
- when multiple VMs are concurrently running on the same physical host, each VM may exhibit a varying and unstable performance (Speed of Execution, and not results), which highly depends on the workload imposed on the system by other VMs, unless proper techniques are used for temporal isolation among virtual machines.

6.2 Process virtual machines

A process VM, sometimes called an *application virtual machine*, runs as a normal application inside a host OS and supports a single process. It is created when that process is started and destroyed when it exits. Its purpose is to provide a platform-independent programming environment that abstracts away details of the underlying hardware or operating system, and allows a program to execute in the same way on any platform.

A process VM provides a high-level abstraction — that of a high-level programming language (compared to the low-level ISA abstraction of the system VM). Process VMs are implemented using an interpreter; performance comparable to compiled programming languages is achieved by the use of just-in-time compilation.

This type of VM has become popular with the Java programming language, which is implemented using the Java virtual machine. Other examples include the Parrot virtual machine, which serves as an abstraction layer for several interpreted languages, and the .NET Framework, which runs on a VM called the Common Language Runtime.

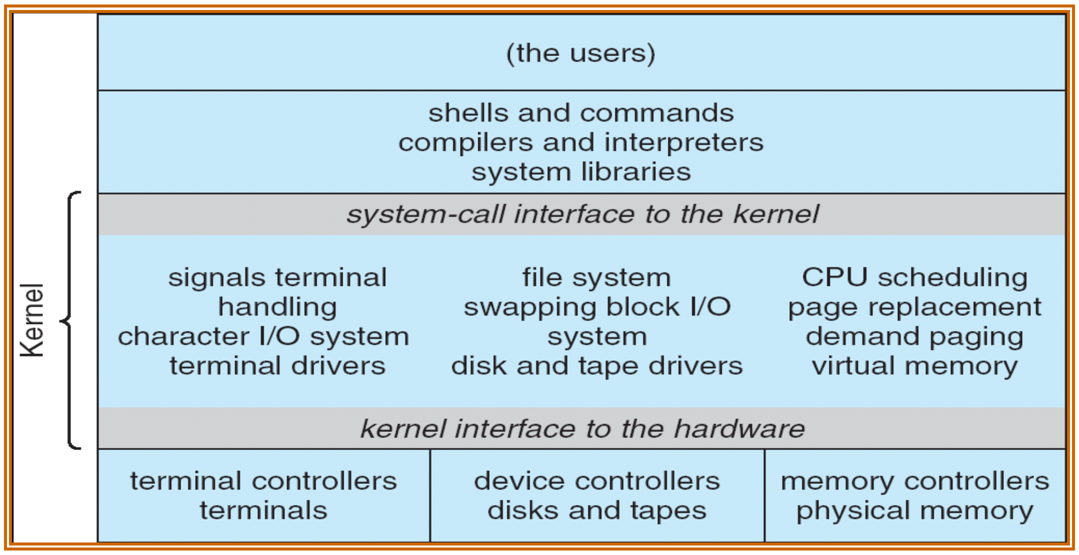
7. Layered Approach

The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface. With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers

7.1 UNIX system

UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts

- Systems programs
- The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

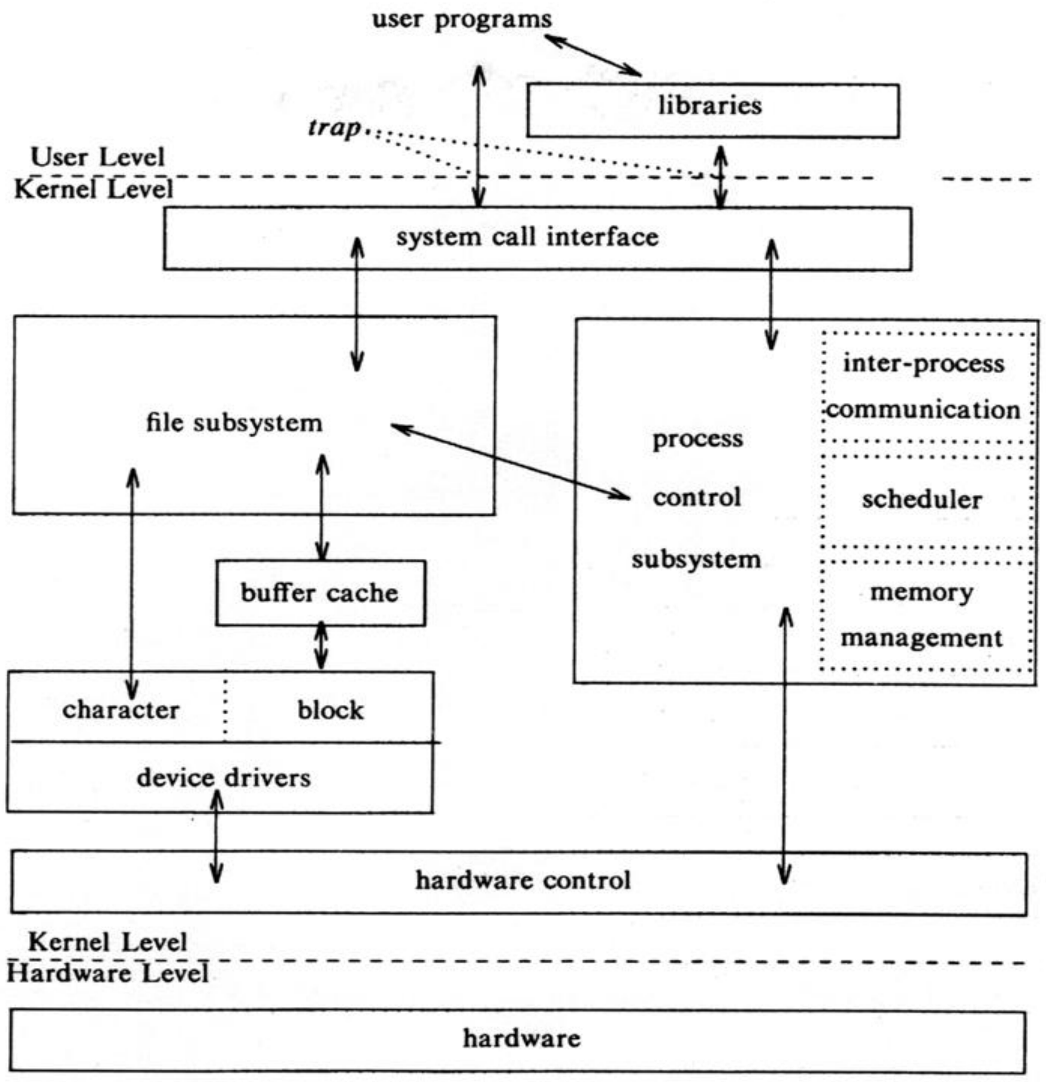


UNIX System Structure

7.2 UNIX kernel system

At the center of the UNIX onion is a program called the kernel. The kernel provides the essential services that make up the heart of UNIX systems; it allocates memory, keeps track of the physical location of files on the computer's hard disks, loads and executes binary programs such as shells, and schedules the task swapping without which UNIX systems would be incapable of doing more than one thing at a time. The kernel accomplishes all these tasks by providing an interface between the other programs running under its control and the physical hardware of the computer; this interface, the system call interface, effectively insulates the other programs on the UNIX system from the complexities of the computer. For example, when a running program needs access to a file, it cannot simply open the file; instead it issues a system call which asks the kernel to open the file. The kernel takes over and handles the request, then notifies the program whether the request succeeded or failed. To read data in from the file takes another system call; the kernel determines whether or not the request is valid, and if it is, the kernel reads the required block of data and passes it back to the program. Unlike DOS (and some other operating systems), UNIX system programs do *not* have access to the

physical hardware of the computer. All they see are the kernel services, provided by the system call interface.
we can say that the kernel handles the following operations :
1. It is responsible for scheduling running of user and other processes.
2. It is responsible for allocating memory.
3. It is responsible for managing the swapping between memory and disk.
4. It is responsible for moving data to and from the peripherals.
5. it receives service requests from the processes and honors them.
All these services are provided by the kernel through a call to a system utility. As a result, kernel by itself is rather a small program that just maintains enough data structures to pass arguments, receive the results from a call and then pass them on to the calling process.
Kernel also aids in carrying out system generation which ensures that Unix is aware of all the peripherals and resources in its environment. For instance, when a new disk is attached, right from its formatting to mounting it within the file system is a part of system generation.

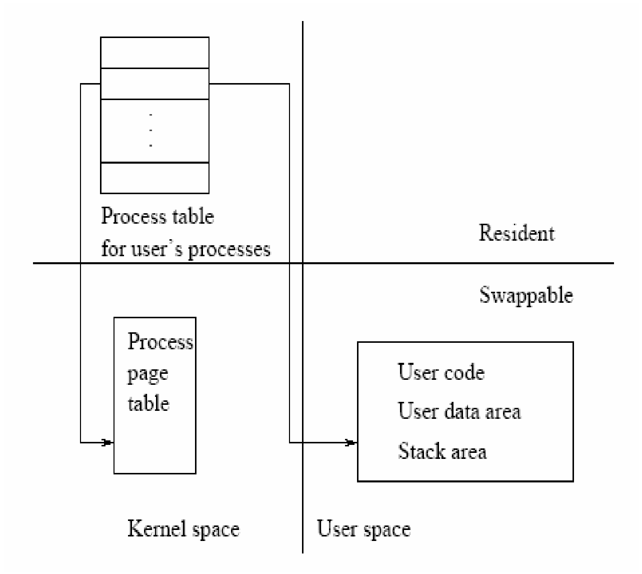


Block Diagram of the System Kernel

7.3 Kernel Operations

The Unix kernel is a main memory resident \process". It has an entry in the process table and has its own data area in the primary memory. Kernel, like any other process, can also use devices on the systems and run processes. Kernel differs from a user process in three major ways.

- The first major key difference between the kernel process and other processes lies in the fact that kernel also maintains the needed data-structures on behalf of UNIX. Kernel maintains most of this data-structure in the main memory itself. The OS based paging or segmentation cannot swap these data structures in or out of the main memory.
- Another way the kernel differs from the user processes is that it can access the scheduler. Kernel also maintains a disk cache, basically a buffer, which is synchronized ever so often (usually every 30 seconds) to maintain disk file consistency. During this period all the other processes except kernel are suspended.
- Finally, kernel can also issue signals to kill any process (like a process parent can send a signal to child to abort). Also, no other process can abort kernel.



User and kernel space

8. UNIX Shell

A **Unix shell** is a command-line interpreter or shell that provides a traditional user interface for the UNIX operating system and for Unix-like systems. Users direct the operation of the computer by entering commands as text for a command line interpreter to execute or by creating text scripts of one or more such commands.

The most influential UNIX shells have been the Bourne shell and the C shell. When the user logs in to the system the shell program is automatically executed. Many types of shells have been developed for this purpose. The program is called a "shell" because it hides the details of the underlying operating system behind the shell's interface. The shell manages the technical details of the operating system kernel interface, which is the lowest-level, or 'inner-most' component of an operating system. The UNIX shell was unusual when it was first created. Since it is both an interactive command language as well as a scripting programming language it is used by UNIX as the facility to control the execution of the system.