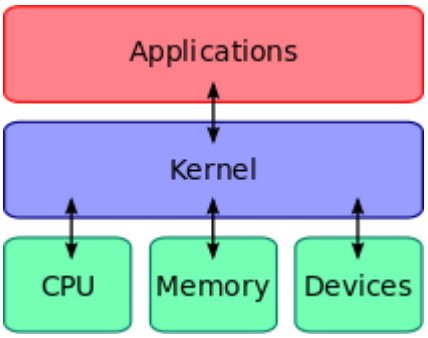
Kernels

1. **Introduction to architecture of a kernel**
2. **Types of kernels**
3. **Context switching (kernel mode and user mode)**
4. **First Level Interrupt handling**
5. **Kernel Implementation of processes**

1. Introduction and Architecture of a Kernel



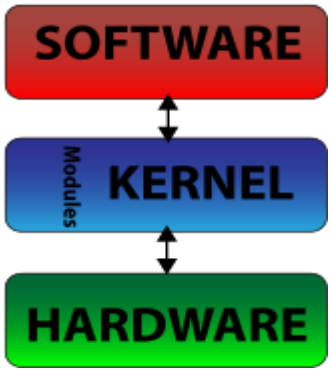
It is the core of the operating system and has complete control over everything that occurs in the system. It is also known as a heart of operating system. The kernel is the core of an operating system that manages input/output requests from software and translates them into data processing instructions. There are many programs, but resources are limited, the kernel has to decide when and how long a program should run. This is called scheduling.

Accessing the hardware directly can be very complex, since there are many different hardware designs for the same type of component. Kernels implement some level of hardware abstraction to hide the underlying complexity from applications and provide a clean and uniform interface. This helps application programmers to develop programs without having to know how to program for specific devices. The kernel relies upon software drivers that translate the generic command into instructions specific to that device. The kernel's primary function is to manage the computer's hardware and resources and allow other programs to run and use these resources. It is the software responsible for running programs and providing secure access to the machine's hardware.

A kernel is a central component of an operating system. It acts as an interface between the user applications and the hardware. The main aim of the kernel is to manage the communication between the software (user level applications) and the hardware (CPU, disk memory etc) i.e. the kernel's primary function is to manage the computer's hardware and resources and allow other programs to run and use these resources. Kernels also usually provide methods for synchronization and communication between processes called inter-process communication (IPC). The main tasks of the kernel are:

- Process management
- Device management
- Memory management
- Interrupt handling
- I/O communication
- File system, etc.

2. Types of Kernels



Kernels may be classified mainly in two categories

- Monolithic
- Micro Kernel

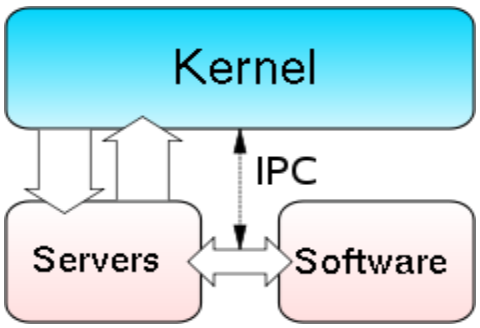
I. Monolithic Kernel

It provides rich and powerful abstractions of the underlying hardware. A Monolithic kernel executes all the operating system instructions in the same address space to improve the performance of the system. In this type of kernel architecture, all the basic system services like process and memory management, interrupt handling etc. were packaged into a single module in kernel space. This type of architecture led to some serious drawbacks like;

- Kernels often become very large and difficult to maintain.
- Monolithic kernels are not portable
- Since the modules run in the same address space, a bug can bring down the entire system.

Monolithic kernels contain all the operating system core functions and the device drivers (such as disk drives, video cards and printers). A monolithic kernel is one single program that contains all of the code necessary to perform every kernel related task. Linux follows the monolithic modular approach

II. Micro Kernel



A micro kernel runs most of the operating system's background process in user space to make the operating system more modular and, therefore, it is easier to maintain. It provides a small set of simple hardware abstractions and use applications called servers to provide more functionality. This architecture allows some basic services like device driver management, protocol stack, file system etc to run in user space. This reduces the kernel code size and also increases the security and stability of operating system. So, if suppose a basic service like network service crashes due to buffer overflow, then only the networking service's memory would be corrupted, leaving the rest of the system still functional.

The microkernel approach consists of defining a simple abstraction over the hardware, with a set of primitives or system calls to implement minimal operating system services such as memory management, multitasking, and inter-process communication. Other services, including those normally provided by the kernel, such as networking, are implemented in user-space programs, and referred to as **servers**. Micro-kernels are easier to maintain than monolithic kernels, but the large number of system calls and context switches might slow down the system because they typically generate more overhead than plain function calls.

3. Context Switching (Kernel Mode and User Mode)

A **context switch** is the computing process of storing and restoring state (context) of a CPU so that execution can be resumed from the same point at a later time. This enables multiple processes to share a single CPU. The context switch is an essential feature of a multitasking operating system. A multitasking

operating system is one in which multiple processes execute on a single CPU simultaneously and without interfering with each other. This illusion of *concurrency* is achieved by means of context switches that are occurring in rapid succession (tens or hundreds of times per second). These context switches occur as a result of processes voluntarily relinquishing their time in the CPU or as a result of the *scheduler* making the switch when a process has used up its CPU *time slice*. Switching from one process to another requires a certain amount of time for doing the administration - saving and loading registers and memory maps, updating various tables and list etc. It is also known as *process switch* or a *task switch.*

Context switching can be described in the kernel performing the following activities with regard to processes (including threads) on the CPU:

- Suspending the progression of one process and storing the CPU's *state* (i.e., the context) for that process somewhere in memory,
- Retrieving the context of the next process from memory and restoring it in the CPU's registers and
- Returning to the location indicated by the program counter in order to resume the process.

There are three situations where a context switch needs to occur. They are:

- Multitasking
- Interrupt handling
- User and kernel level switching

I. Multitasking

In multitasking environment one process needs to be switched out of the CPU so another process can run. Within a preemptive multitasking operating system, the scheduler allows every task to run for some certain amount of time. It is called its time slice. If a process does not voluntarily give up the CPU (for example, by performing an I/O operation), a timer interrupt fires, and the operating system schedules another process for execution instead.

II. Interrupt handling

Modern architectures are interrupt-driven. If any interrupt signal is generated then it is handled by a program called an interrupt handler which is already installed, and it is the interrupt handler that handles the interrupt from the disk.

The kernel services the interrupts in the context of the interrupted process even though it may not have caused the interrupt. The interrupted process may have been executing in user mode or in kernel mode. The kernel saves enough information so that it can later resume execution of the interrupted process and services the interrupt in kernel mode. The kernel does not spawn or schedule a special process to handle interrupts.

III. User and kernel level switching

Kernel space is strictly reserved for running privileged kernel, kernel extensions, and most device drivers. In contrast, ***user space*** is the memory area where application software and some drivers execute.

Context switches can occur only in kernel mode. Kernel mode is a privileged mode of the CPU in which only the kernel runs and which provides access to all memory locations and all other system resources. Other programs, including applications, initially operate in user mode, but they can run portions of the kernel code via system calls. Kernel is the heart of operating system which manages the core features of an operating system while if some useful applications and utilities are added over the

kernel, then the complete package becomes an operating system. So, it can easily be said that an operating system consists of a kernel space and a user space. There are two execution modes for the CPU for the execution of task and they are:

- User mode and
- Kernel mode

a) User Mode:

It is a *non-privileged* mode for user programs in which each process starts out. It is forbidden for processes in this mode to access those portions of memory (i.e., RAM) that have been allocated to the kernel or to other programs. The kernel is not a process, but rather a controller of processes, and it alone has access to all resources on the system.

When a *user mode process* (i.e., a process currently in user mode) wants to use a service that is provided by the kernel, it must switch temporarily into kernel mode, which has *root* (i.e., administrative) privileges, including *root access permissions* (i.e., permission to access any memory space or other resources on the system). When the kernel has satisfied the process's request, it restores the process to user mode. This change in mode is termed a ***mode switch***.

b) Kernel Mode:

Kernel mode is also referred to as *system mode*. When the CPU is in kernel mode, it is assumed to be executing *trusted* software, and thus it can execute any instructions and reference any memory addresses. Kernel is *trusted* software, but all other programs are considered *untrusted* software. Thus, all user mode software must request use of the kernel by means of a system call in order to perform privileged instructions, such as process creation or *input/output* operations.

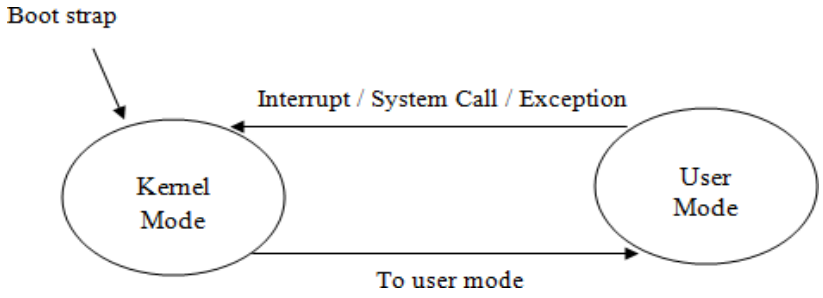


Fig: User Mode – Kernel Model

4. Interrupt Handling

Interrupt handler is a section of a computer program or of the operating system that takes control when an interrupt is received and performs the operations required to service the interrupt. It is also known as Interrupt Service Routine (ISR). These handlers are initiated by either hardware interrupts or software interrupts. It is used for servicing hardware devices and transitions between protected modes of operation such as system calls.

Interrupt handlers are divided into two parts:

- First-Level Interrupt Handler (FLIH) and
- Second-Level Interrupt Handlers (SLIH)

I. First-Level Interrupt Handler (FLIH)

FLIH is also known as *hard interrupt handlers* or *fast interrupt handlers*. It is a software or hardware routine that is activated by interrupt signals sent by peripheral devices and decides, based on the relative importance of the interrupts, how they should be handled. The job of a FLIH is to quickly service the interrupt and schedule the execution of a Second Level Interrupt Handler for further long-lived interrupt handling. It performs following task:

- save registers of current process in PCB
- Determine the source of interrupt
- Initiate service of interrupt

II. Second-Level Interrupt Handlers (SLIH)

SLIH is also known as *slow/soft interrupt handlers*. A SLIH completes long interrupt processing tasks similarly to a process. SLIH either have a dedicated kernel thread for each handler, or are executed by a pool of kernel worker threads. These threads sit on a run queue in the operating system until processor time is available for them to perform processing for the interrupt. SLIHs may have a long-lived execution time, and thus are typically scheduled similarly to threads and processes.

5. Kernel Implementation of Processes

A kernel process is a process that is created in the kernel protection domain and always executes in the kernel protection domain. Kernel processes can be used in subsystems, by complex device drivers, and by system calls. They can also be used by interrupt handlers to perform asynchronous processing not available in the interrupt environment. Kernel processes can also be used as device managers where asynchronous input/output (I/O) and device management is required.

Introduction to Kernel Processes

A kernel process (kproc) exists only in the kernel protection domain and differs from a user process in the following ways:

- It is created using the **creatp** and **initp** kernel services.
- It executes only within the kernel protection domain and has all security privileges.
- It can call a restricted set of system calls and all applicable kernel services.
- It has access to the global kernel address space (including the kernel pinned and pageable heaps), kernel code, and static data areas.
- It must poll for signals and can choose to ignore any signal delivered, including a **kill** signal.
- Its text and data areas come from the global kernel heap.
- It cannot use application libraries.
- It has a process-private region containing only the **u-block** (user block) structure and possibly the kernel stack.
- Its *parent process* is the process that issued the **creatp** kernel service to create the process.
- It can change its parent process to the **init** process and can use interrupt disable functions for serialization.
- It can use locking to serialize process-time access to critical data structures.
- It can only be a 32–bit process in the 32–bit kernel.
- It can only be a 64–bit process in the 64–bit kernel.