KERNEL - AN INTRODUCTION

Introduction:

The central module of an operating system. It is the part of the operating system that loads first, and it remains in main memory. Because it stays in memory, it is important for the kernel to be as small as possible while still providing all the essential services required by other parts of the operating system and applications. Typically, the kernel is responsible for memory management, process and task management, and disk management.

Context Switch

A **context switch** is the computing process of storing and restoring the 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. Context switches are usually computationally intensive and much of the design of operating systems is to optimize the use of context switches. A context switch can mean a register context switch, a task context switch, a thread context switch, or a process context switch. What constitutes the context is determined by the processor and the operating system. 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.

process can run in two modes:
1.User Mode.
2.Kernel Mode.
1.User Mode:
=>A mode of the CPU when running a program.
=>In this mode ,the user process has no access to the memory locations used by the kernel.When a program is running in User Mode, it cannot directly access the kernel data structures or the kernel programs.

2.Kernel Mode:
=>A mode of the CPU when running a program. =>In this mode, it is the kernel that is running on behalf of the user process and directly access the kernel data structures or the kernel programs.Once the system call returns,the CPU switches back to user mode.

When you execute a C program,the CPU runs in user mode till the system call is invoked. In this mode,the user process has access to a limited section of the computer's memory and can execute a restricted set of machine instructions. however,when the process invokes a system call,the CPU switches from user mode to a more privileged mode the kernel. In this mode ,it is the kernel that runs on behalf of the user process,but it has access to any memory location and can execute any machine instruction. After the system call has returned,the CPU switches back to user mode.

Types Of Kernels

1 Monolithic Kernels

Earlier 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 1) Size of kernel, which was huge. 2)Poor maintainability, which means bug fixing or addition of new features resulted in recompilation of the whole kernel which could consume hours .

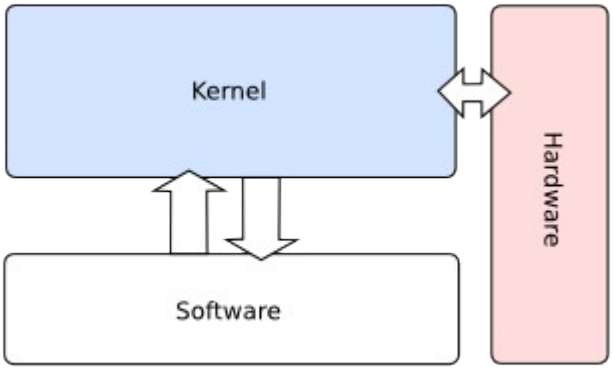

Fig: Monolithic Kernel

In a modern day approach to monolithic architecture, the kernel consists of different modules which can be dynamically loaded and un-loaded. This modular approach allows easy extension of OS's capabilities. With this approach, maintainability of kernel became very easy as only the concerned module needs to be loaded and unloaded every time there is a change or bug fix in a particular module. So, there is no need to bring down and recompile the whole kernel for a smallest bit of change. Also, stripping of kernel for various platforms (say for embedded devices etc) became very easy as we can easily unload the module that we do not want.

Examples:

- Linux
- Windows 9x (95, 98, Me)
- Mac OS <= 8.6
- BSD

2 Microkernels

A Microkernel tries to run most services - like networking, filesystem, etc. - as daemons / servers in user space. All that's left to do for the kernel are basic services, like memory allocation (however, the actual memory manager is implemented in userspace), scheduling, and messaging (Inter Process Communication).

This architecture majorly caters to the problem of ever growing size of kernel code which we could not control in the monolithic approach. 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 OS as we have the bare minimum code running in kernel. 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.

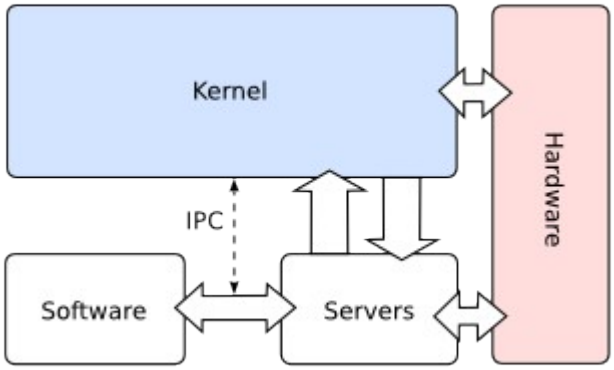

Fig: Microkernel

In this architecture, all the basic OS services which are made part of user space are made to run as servers which are used by other programs in the system through inter process communication (IPC). eg: we have servers for device drivers, network protocol stacks, file systems, graphics, etc. Microkernel servers are essentially daemon programs like any others, except that the kernel grants some of them privileges to interact with parts of physical memory that are otherwise off limits to most programs. This allows some servers, particularly device drivers, to interact directly with hardware. These servers are started at the system start-up.

So, what the bare minimum that microKernel architecture recommends in kernel space?

* Managing memory protection
* Process scheduling
* Inter Process communication (IPC)

Apart from the above, all other basic services can be made part of user space and can be run in the form of servers.

Examples:

- QNX
- L4
- AmigaOS
- Minix

QNX follows the Microkernel approach

3. Exo-kernel:

Exokernels are an attempt to separate security from abstraction, making non-overrideable parts of the operating system do next to nothing but securely multiplex the hardware. The goal is to avoid forcing any particular abstraction upon applications, instead allowing them to use or implement whatever abstractions are best suited to their task without having to layer them on top of other abstractions which may impose limits or unnecessary overhead. This is done by moving abstractions into untrusted user-space libraries called "library operating systems" (libOSes), which are linked to applications and call the operating system on their behalf.

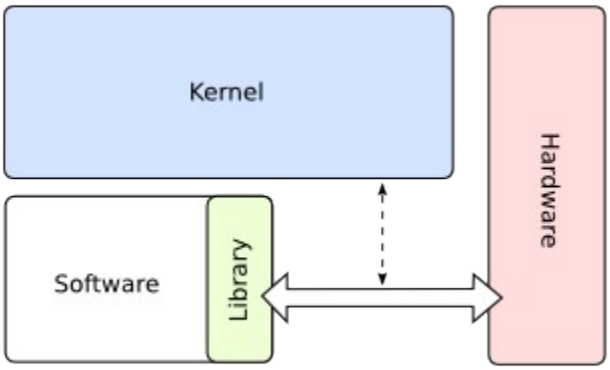

Fig: Exokernel

Interrupt Handler:

An interrupt handler, also known as an interrupt service routine (ISR), is a callback subroutine in operating system or device driver whose execution is triggered by the reception of an interrupt. Interrupt handlers have a multitude of functions, which vary based on the reason the interrupt was generated and the speed at which the interrupt handler completes its task.

First Level Interrupt Handler (FLIH)

- save registers of current process in PCB
- Determine the source of interrupt
- Initiate service of interrupt - calls interrupt handler
  Hardware dependent - implemented in assembler

In several operating systems - Linux, Unix, Mac OS X, Microsoft Windows, and some other operating systems in the past, interrupt handlers are divided into two parts: the First-Level Interrupt Handler (FLIH) and the Second-Level Interrupt Handlers (SLIH). FLIHs are also known as hard interrupt handlers or fast interrupt handlers, and SLIHs are also known as slow/soft interrupt handlers, Deferred Procedure Call.

A FLIH implements at minimum platform-specific interrupt handling similarly to interrupt routines. In response to an interrupt, there is a context switch, and the code for the interrupt is loaded and executed. The job of a FLIH is to quickly service the interrupt, or to record platform-specific critical information which is only available at the time of the interrupt, and schedule the execution of a SLIH for further long-lived interrupt handling.

Source : http://dayaramb.files.wordpress.com/2012/02/operating-system-pu.pdf