Operating System

Bachelors in Computer Engineering

Chapter four

Kernel

OUTLINES:

1. Introduction, Architecture of the Kernel, Types of Kernels
2. Context Switching (Kernel and User mode), Kernel implementation processes

Introduction and Architecture of Kernel

- The kernel serves as the core component of an operating system, acting as a bridge between applications and hardware-level data processing.
- When an operating system is loaded into memory, the kernel loads first and remains in memory until the operating system is shut down again.
- The kernel is responsible for handling low-level tasks, including disk management, task management, and memory management.
- It acts as an interface between the application/user interface and the CPU, memory, and other hardware I/O devices.
- The kernel provides and manages computer resources, allowing other programs to run and use these resources.
- The kernel also sets up memory address space for applications, loads files with application code into memory, sets up the execution stack for programs.
- Manages processes for application execution, ensuring efficient utilization of system resources.

- The kernel is responsible for:
  - Process management for application execution
  - Memory management, allocation and I/O
  - Device management through the use of device drivers
  - System call control, which is essential for the execution of kernel services

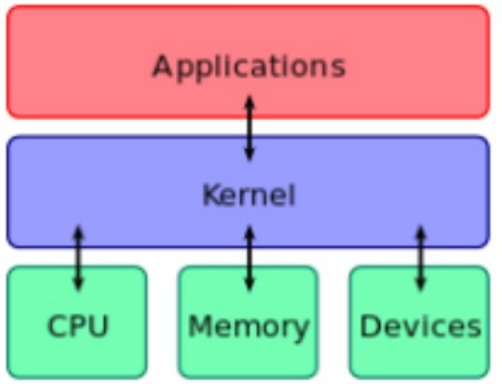


Fig: Architecture of Kernel

Types of kernel

Kernels can be broadly classified mainly in two categories:

- Monolithic Kernel
- Micro Kernel.

Monolithic Kernels

- Earlier in this type of kernel architecture, all the basic system services like a 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:
  - The Size of the kernel, which was huge.
  - Poor maintainability, which means bug fixing or addition of new features resulted in recompilation of the whole kernel which could consume hours.
- In a modern day approach to monolithic architecture, the kernel consists of different modules which can be dynamically loaded and unloaded. 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 the smallest bit of change. Also, stripping of a kernel for various platforms (say for embedded devices etc) became very easy as we can easily unload the module that we do not want. Linux follows the monolithic modular approach.

Microkernels

- 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.
- 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.

Other types of kernel include:

Hybrid Kernel:

This is a mix between the monolithic and microkernel architectures. The kernel is larger than hybrid but smaller than monolithic. What you normally get is a stripped down monolithic kernel that has the majority of device drivers removed but still all of the system services within the kernel space. The device drivers will be attached to the kernel as required when starting up or running. These kernels are typically found on desktops, your Windows, Mac and Linux OS flavors.

Nano kernel:

- This kernel type only offers hardware abstraction, there are no services and the kernel space is at a minimum. A Nano kernel forms the basis of a hypervisor upon which you may emulate multiple systems via virtualization. Nano kernels are also very good for embedded projects.
- Nano Kernel is a type of operating system kernel that has a minimal and streamlined design, with a focus on minimalism and efficiency. The goal of a nano kernel is to provide only the essential functions required for the operation of a system while delegating other functions to user-space processes.

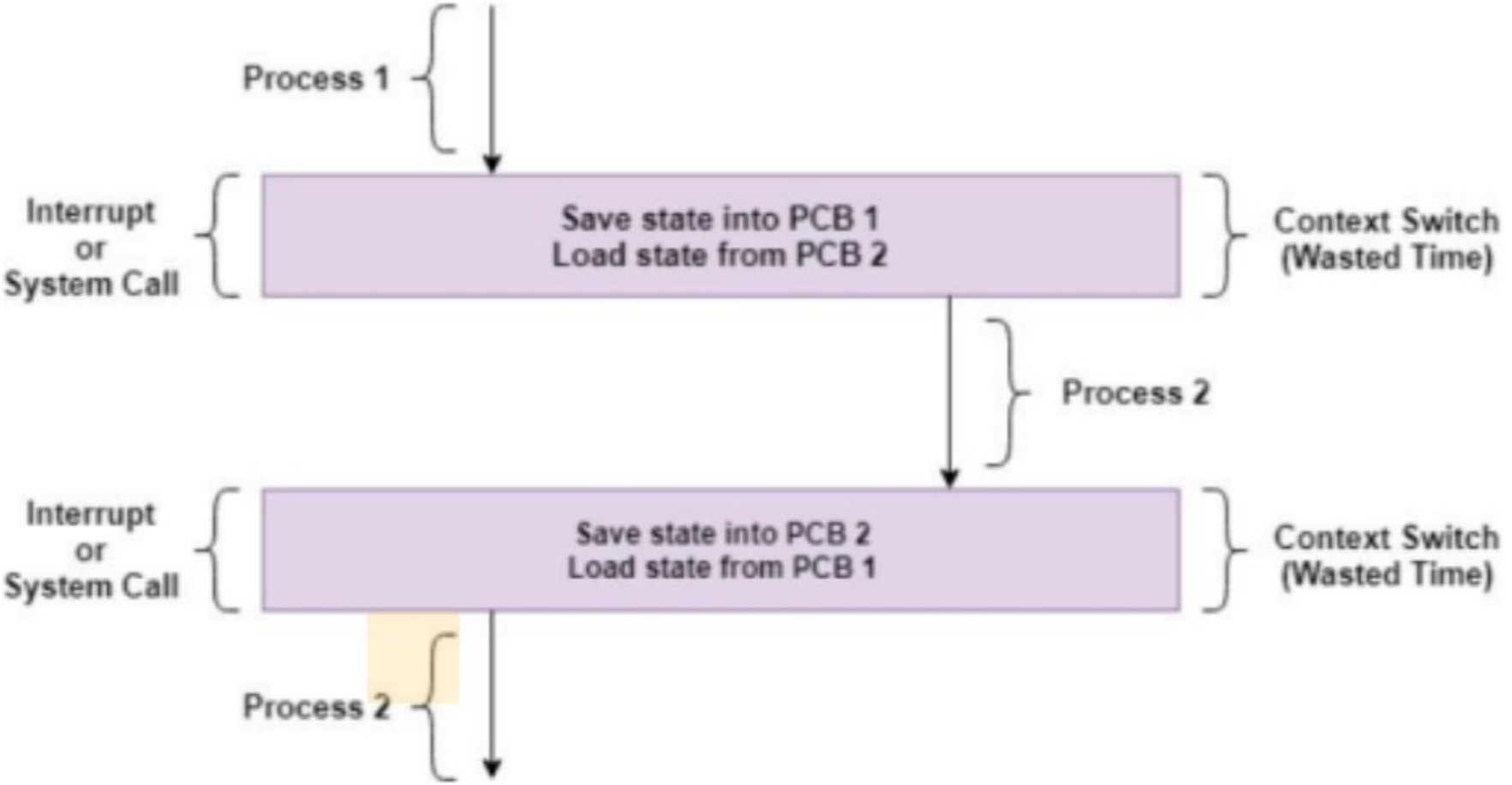
Exo-Kernel:

- This kernel is the smallest kernel. It offers process protection and resource handling and nothing else. The programmer using this kernel is responsible for correctly accessing the device they wish to use.
- Exo-kernel are an attempt to separate security from abstraction, making non-overrideable parts of the OS 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 abstraction are best suited to their task without having to layer them on the top of other abstraction which may impose limits or unnecessary overhead.
- This is done by moving abstractions into untrusted user-space libraries called ‘’Library operating systems” which are linked to applications and call the operating system on their behalf.

Context Switching (Kernel mode and User mode)

- A context switch is a procedure that a computer's CPU (central processing unit) follows to change from one task (or process) to another while ensuring that the tasks do not conflict.
- Context Switching involves storing the context or state of a process so that it can be reloaded when required and execution can be resumed from the same point as earlier.
- This is a feature of a multitasking operating system and allows a single CPU to be shared by multiple processes.

A diagram that demonstrates context switching is as follows:



- In the above diagram, initially Process 1 is running. Process 1 is switched out and Process 2 is switched in because of an interrupt or a system call. Context switching involves saving the state of Process 1 into PCB1 and loading the state of process 2 from PCB2. After some time again a context switch occurs and Process 2 is switched out and Process 1 is switched in again. This involves saving the state of Process 2 into PCB2 and loading the state of process 1 from PCB1.

When does context switching take place?

- Multitasking: In a multitasking environment, a process is switched out of the CPU so another process can be run. The state of the old process is saved and the state of the new process is loaded. On a pre-emptive system, processes may be switched out by the scheduler.
- Interrupt Handling: The hardware switches a part of the context when an interrupt occurs. This happens automatically. Only some of the context is changed to minimize the time required to handle the interrupt.
- User and Kernel Mode Switching: A context switch may take place when a transition between the user mode and kernel mode is required in the operating system.

Context Switching Steps:

The steps involved in context switching are as follows:

- Save the context of the process that is currently running on the CPU. Update the process control block and other important fields.
- Move the process control block of the above process into the relevant queue such as the ready queue, I/O queue etc.
- Select a new process for execution.
- Update the process control block of the selected process. This includes updating the process state to running.
- Update the memory management data structures as required.
- Restore the context of the process that was previously running when it is loaded again on the processor.
    - This is done by loading the previous values of the process control block and registers.

Kernel Mode

- When CPU is in kernel mode, the code being executed can access any memory address and any hardware resource.
- It is a very privileged and powerful mode.
- If a program crashes in kernel mode, the entire system will be halted.
- In Kernel mode, the executing code has complete and unrestricted access to the underlying hardware.
- core operating system components run in kernel mode.

User Mode

- When CPU is in user mode, the programs don‘t have access to memory and hardware resources.
- In user mode, if any program crashes, only that particular program is halted. That means the system will be in a safe state even if a program in user mode crashes.
- Most programs in an OS run in user mode.
- In user mode, the executing code has restricted access to the underlying hardware.
- Applications run in user mode

Kernel Implementation of processes

- Most of the operating system executes within the environment of a user process.
- We need two modes: User mode and Kernel mode.
  - Some portion of the operating system operate as a separate process known as kernel process.
  - Process creation in Unix is made by means of kernel system call fork ().
- When a process issues a fork request, the operating system perform the following function:

  i. It allocates a slot in process table for a new process.

  ii. It assigns a unique process ID to a child process.

  iii. It assigns the child process to a ready to run state.

  iv. It returns the ID number of the child process to parent process and zero value to a child process.

- All this work is accomplished in kernel mode in the parent process. When a kernel has completed these functions, it can do one of the following routine

  i. Stay in parent process: Control return to the user mode at the point of the fork call of the parent.

  ii. Transfer control to the child process: The child process begins executing at the same point in the code as the parent, at the return from fork call.

  iii. Transfer control to another process: Both the child and the parent process are left in ready to run state.

Example:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        // make two process which run same
        // program after this instruction
        fork();
        printf("I am Nabraj!");
        return 0;
}
```

Output:

I am Nabraj!

I am Nabraj!

Number of times I am Nabraj! printed is equal to number of process created.

Total Number of Processes = 2^n where n is number of fork system calls.

So here n = 1, 2^1 =2

Thank You