Linux System Internals and Programming

1.1. Linux System Overview

1.1.1. Purpose of Understanding Linux System

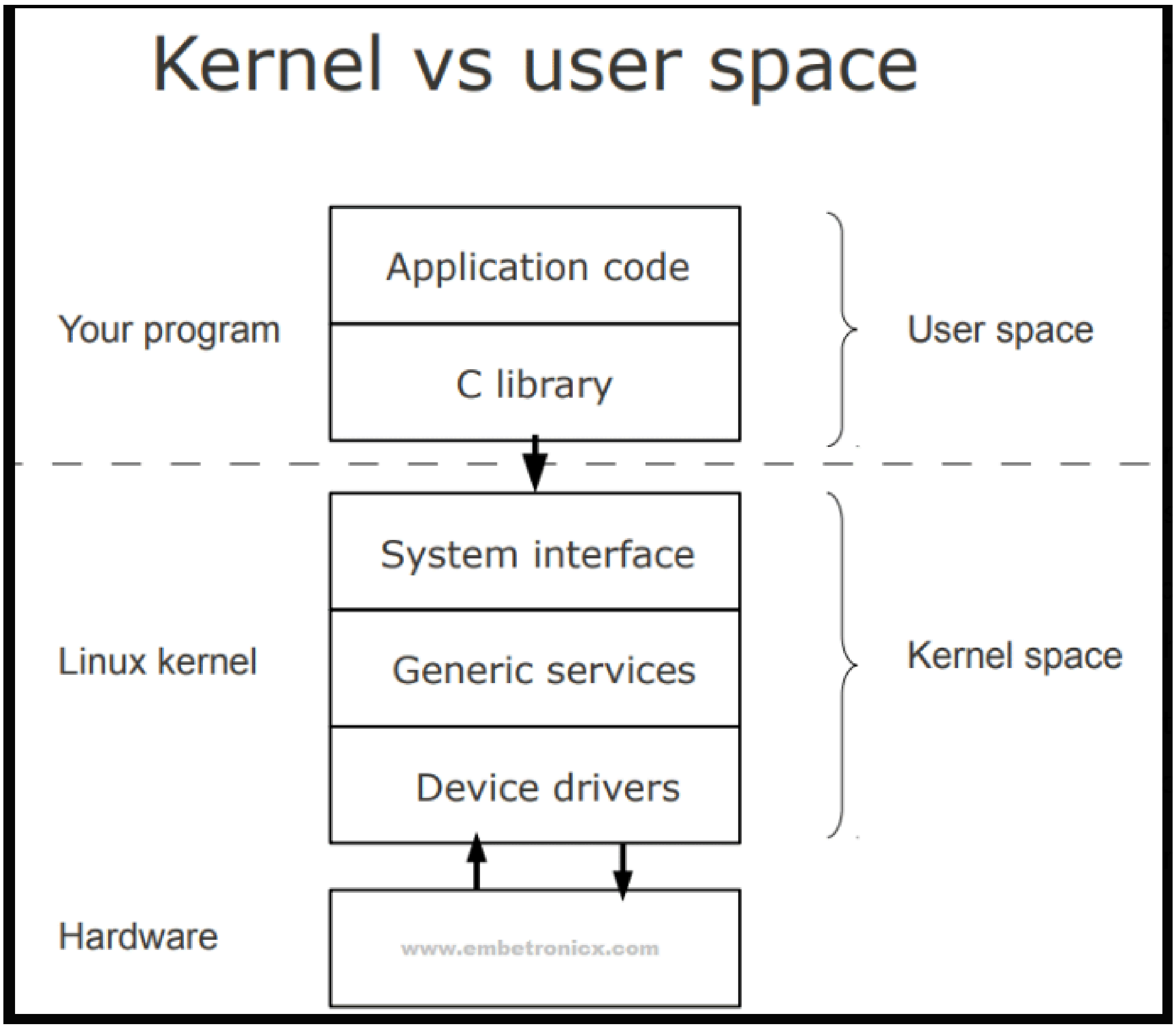
Understanding Linux system internals and programming is crucial for several reasons:

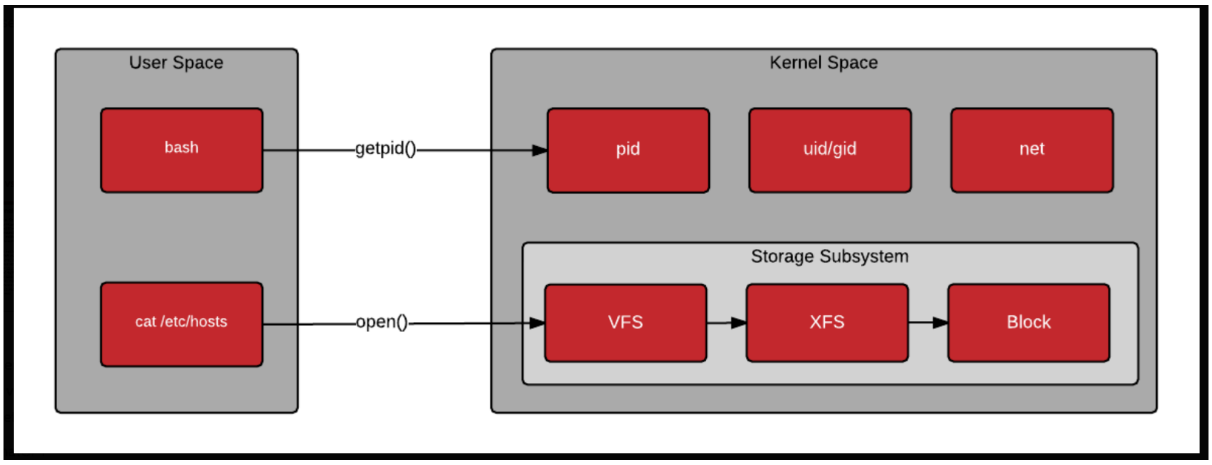
- **Efficient System Resource Management:** Developers can write optimized code by understanding how the OS manages resources like memory, CPU, and I/O.
- **Kernel Interactions:** Knowing how applications interact with the kernel helps in debugging performance issues and writing system-level programs.
- **Security & Access Control:** Deep knowledge of privilege levels, user permissions, and kernel security mechanisms enhances system security.
- **Customization & Optimization:** System administrators and embedded developers can fine-tune Linux for specific applications.
- **Effective Troubleshooting:** Debugging system crashes, performance bottlenecks, and security vulnerabilities requires an understanding of Linux internals.

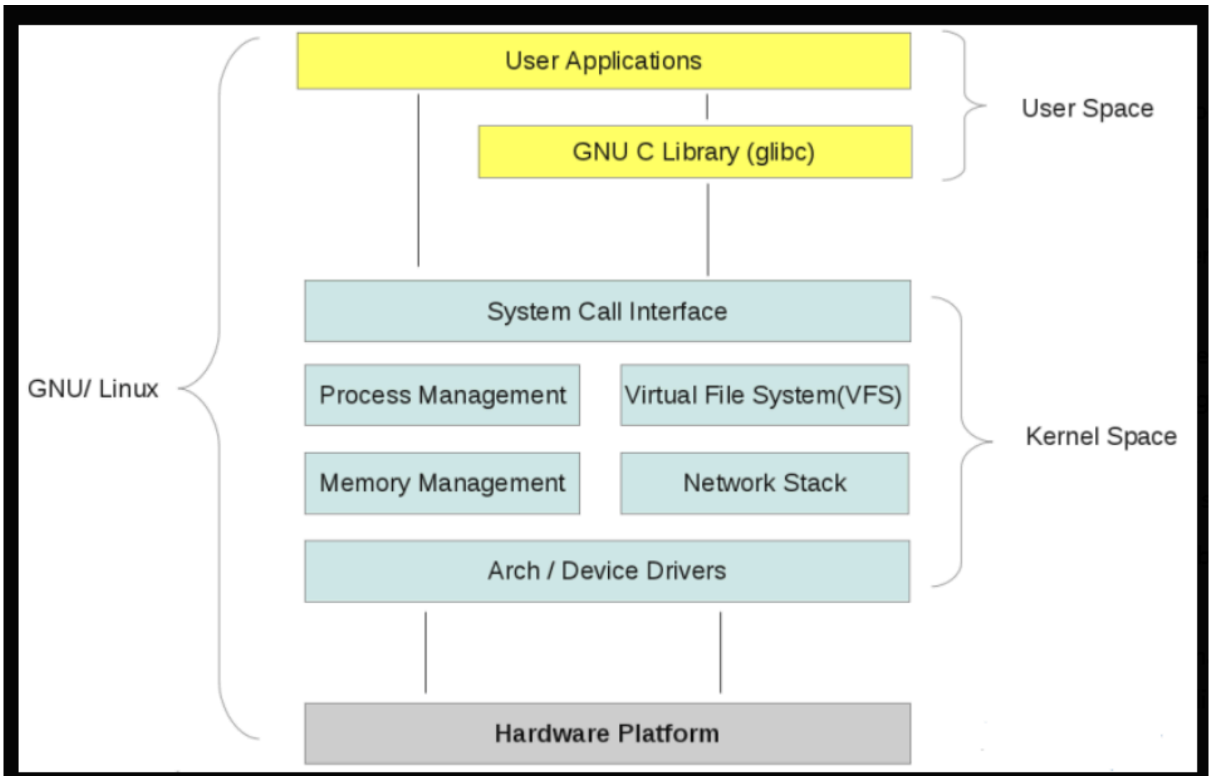
1.1.2. Significance of Kernel vs. User Space

Linux is divided into two major spaces:

1. **Kernel Space**
    - Runs with **full privileges** (Ring 0).
    - Manages system hardware, process scheduling, memory, file systems, and security.
    - Kernel modules, device drivers, and core OS functionalities reside here.
2. **User Space**
    - Runs with **limited privileges** (Ring 3).
    - Contains user applications, system utilities, and libraries.
    - Applications interact with the kernel through **system calls** (e.g., `read()`, `write()`).


Kernel vs user space
Your program
Application code
C library
User space
Linux kernel
System interface
Generic services
Device drivers
Kernel space
Hardware
www.embetronicx.com


User Space
bash
cat /etc/hosts
getpid()
open()
Kernel Space
pid
uid/gid
net
Storage Subsystem
VFS
XFS
Block


User Applications
GNU C Library (glibc)
User Space
System Call Interface
Process Management
Virtual File System(VFS)
Memory Management
Network Stack
Arch / Device Drivers
Kernel Space
GNU/ Linux
Hardware Platform

Key Differences:

Feature	Kernel Space	User Space
Access Level	Ring 0 (Full Privileges)	Ring 3 (Restricted)
Direct Hardware Access	Yes	No (Relies on System Calls)
Memory Protection	No (Direct Access)	Yes (Protected)
Execution Speed	Faster (No Context Switching)	Slower (Context Switching Required)

1.2. Privilege Levels and Rings

Understanding CPU Privilege Rings in Linux

1. What Are Rings in a CPU?

Rings in a CPU are **privilege levels** designed to control access to system resources. They ensure that only trusted parts of an operating system can execute critical operations while restricting user applications from directly manipulating hardware.

Most modern processors, including **x86 architecture**, support **four privilege levels (Ring 0 to Ring 3)**, but Linux primarily uses only **Ring 0 (Kernel Mode)** and **Ring 3 (User Mode)**.

---

2. How CPU Rings Work

CPU rings implement **layered security**, where lower-numbered rings have **more privileges** and can control higher-numbered rings.

Privilege Levels in x86 Architecture

```
+-------------------------------------+
| Ring 3 (User Mode)                 |  << Least Privileged
| - User Applications                 |
| - Libraries (glibc, etc.)            |
+-------------------------------------+
| Ring 2 (Not Used in Linux)          |
+-------------------------------------+
| Ring 1 (Not Used in Linux)          |
+-------------------------------------+
| Ring 0 (Kernel Mode)                |  << Most Privileged
| - Kernel (Linux, Windows, etc.)      |
| - Device Drivers, Memory Manager     |
| - Process Scheduler, Filesystem      |
+-------------------------------------+
```

- **Ring 0 (Kernel Mode)**
    - Has **full access** to hardware and system resources.
    - Executes the **Linux kernel**, device drivers, and core system functions.
    - Can directly access **CPU registers, memory, and I/O ports**.
- **Ring 1 & Ring 2 (Unused in Linux)**
    - These were meant for **device drivers and privileged applications**, but modern OSes do not use them.
- **Ring 3 (User Mode)**
    - Runs **user applications** (e.g., web browsers, games, text editors).
    - Cannot access hardware directly—must request services from the kernel via **system calls**.
    - Prevents user programs from crashing the entire OS.

---

3. How Rings Protect the System

CPU rings protect the system using **privilege separation** and **memory protection**.

- **If a user program (Ring 3) tries to access hardware directly, the CPU blocks it.**
- **Only the kernel (Ring 0) has full system control.**

Example of Privilege Enforcement:

Imagine a **text editor (like Vim) running in Ring 3**:

1. **Text Editor (User Mode - Ring 3) → Wants to Save a File → Needs Disk Access.**
2. **Cannot access the disk directly (restricted in Ring 3).**
3. **Sends a System Call (`write()`) to the Kernel (Ring 0).**
4. **Kernel (Ring 0) Interacts with the Filesystem and Writes Data to Disk.**
5. **Kernel Returns Control to the Text Editor in Ring 3.**

💡 **Key Concept: User programs must ask the kernel for privileged operations via system calls.**

---

4. System Calls: The Bridge Between Rings

Since **Ring 3 (User Mode) cannot access hardware**, it must request services from **Ring 0 (Kernel Mode)** using **System Calls**.

Example: Reading a File (C Program)

When a program wants to read a file, it uses a **system call** like `read()`:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    int fd = open("example.txt", O_RDONLY); // Opens a file (system call)
    if (fd < 0) {
        perror("Open failed");
        return 1;
    }

    char buffer[100];
    read(fd, buffer, 100); // Reads file (system call)
    close(fd); // Closes file (system call)

    return 0;
}
```

- `open()` calls the kernel to access the file system.
- `read()` calls the kernel to fetch data from the disk.
- `close()` tells the kernel to release the file descriptor.

💡 **System Calls = Controlled Entry into Ring 0 from Ring 3.**

---

5. Why Are Rings Important?

✅ **Security** – Prevents user applications from damaging the system.
✅ **Stability** – If a user program crashes, it does not crash the entire OS.
✅ **Performance** – Reduces the risk of malicious or buggy programs affecting system integrity.
✅ **Multi-User Support** – Different users can run applications safely without interfering with each other.

---

6. Real-World Example of Rings in Action

Case: A Web Browser Running in Ring 3

- **A browser like Chrome runs in User Mode (Ring 3).**
- **It cannot directly access the network card or disk.**
- **When it needs to fetch a web page, it sends a request to the Kernel (Ring 0) via system calls (like `send()` and `recv()`).**
- **Kernel handles network communication and returns data to the browser.**

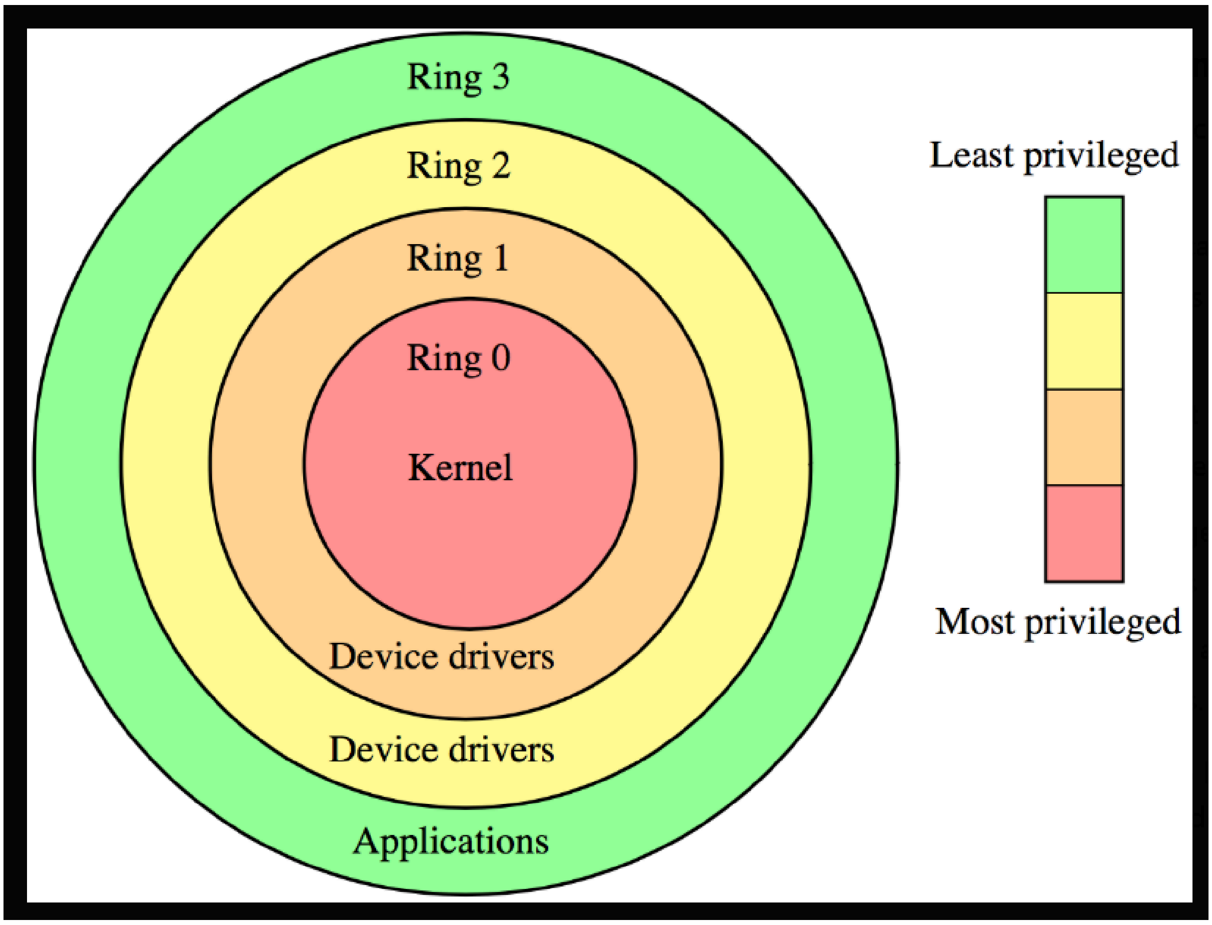
- **This prevents the browser from damaging the OS if it crashes or contains malware.**

---

7. Summary

Ring	Used By	Access Level	Used in Linux?
Ring 0	Kernel, Device Drivers	Full Access to Hardware	✅ Yes
Ring 1	Reserved (Unused)	Some Privileges	❌ No
Ring 2	Reserved (Unused)	Some Privileges	❌ No
Ring 3	User Applications	Restricted Access	✅ Yes

🔹 **Linux only uses Ring 0 (Kernel Mode) and Ring 3 (User Mode).**
🔹 **System Calls allow User Mode to request privileged operations from the Kernel.**



1.2.1. Concept of Privilege Levels (Rings)

Modern processors implement **privilege levels (rings)** to enforce security and isolation between different execution contexts.

- **Ring 0:** Full system access (Kernel mode).
- **Ring 1 & Ring 2:** Intermediate levels (not commonly used in Linux).
- **Ring 3:** Restricted access (User mode).

Why Rings Exist?

- Prevents user applications from directly accessing system hardware.
- Ensures that only trusted code (kernel) has full control over system operations.
- Protects against security vulnerabilities like buffer overflows and privilege escalation.

1.2.2. Ring 0: The Kernel Mode

- **Executes critical system operations** (e.g., managing memory, scheduling processes, handling interrupts).
- Directly interacts with **hardware (CPU, RAM, disks, network interfaces, etc.)**.
- Runs **kernel threads, device drivers, and system calls**.
- Can **bypass memory protection** and directly manipulate system resources.

1.2.3. Ring 3: The User Mode

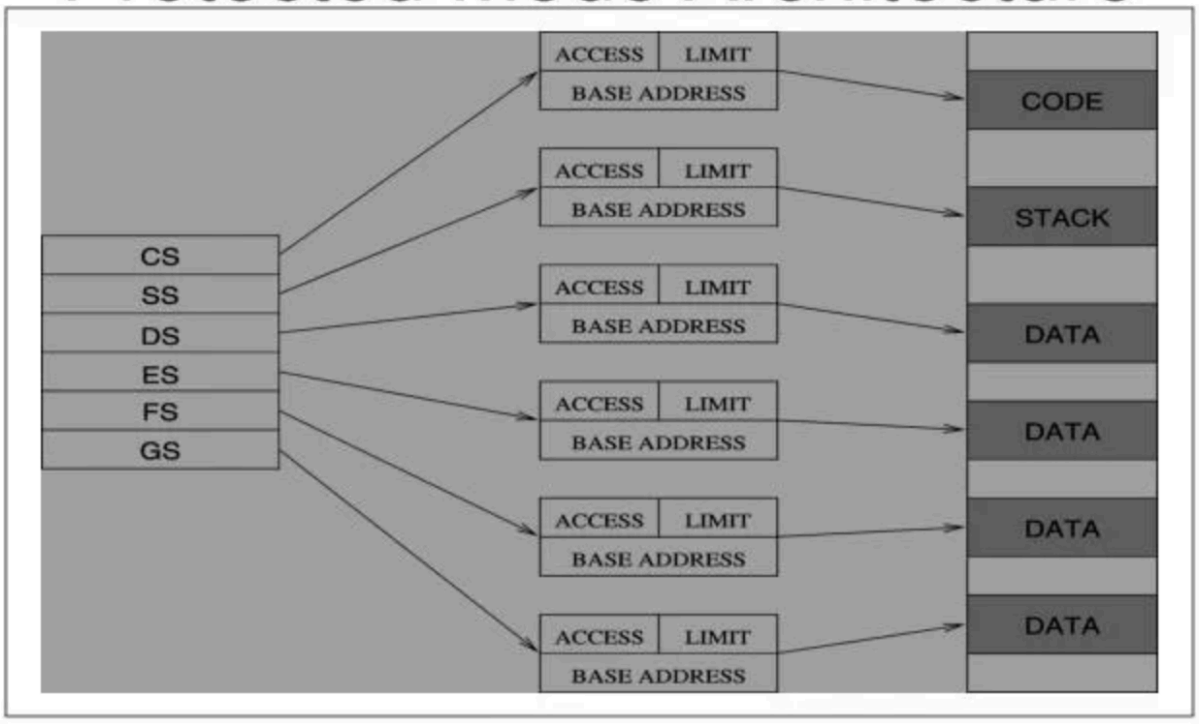
- Runs **applications, libraries, and shell commands**.
- Programs operate in **protected memory space**, preventing direct access to hardware.
- Uses **system calls** to request kernel services (e.g., file operations, memory allocation).

1.2.4. Intermediate Rings (Ring 1 and Ring 2)

- Rarely used in Linux.
- Could be used for **virtualization, hypervisors, or device drivers** in some architectures.
- Linux mostly uses **only Ring 0 (Kernel Mode) and Ring 3 (User Mode)**.

1.3. Protected Mode in the x86 Architecture

Protected Mode Architecture



Default Segments

- Pentium uses default segments depending on the purpose of the memory reference
  - Instruction fetch
    - CS register (Code Section)
  - Stack operations
    - 16-bit mode: SP (Stack Pointer)
    - 32-bit mode: ESP (Extended Stack Pointer)
  - Accessing data
    - DS register (Data Section)
    - Offset depends on the addressing mode

[illegible]

1.3.1. Introduction to Protected Mode

Protected mode is an operational mode of x86 CPUs that allows:

- **Memory protection** (each process gets its own memory space).
- **Multitasking** (multiple processes can run simultaneously).
- **Virtual memory support** (using paging and segmentation).

It was introduced in **Intel 80286 processors** and fully utilized in **80386 and later CPUs**.

1.3.2. Transition from Real Mode to Protected Mode

Real Mode (Legacy Mode)

- Default startup mode of x86 processors.
- **No memory protection, no multitasking, and direct hardware access.**
- Can only access **1MB of memory** (uses **segmentation**).

Transition to Protected Mode

1. **Disable interrupts** to avoid execution conflicts.
2. **Load Global Descriptor Table (GDT)** to define memory segments.
3. **Set the Protection Enable (PE) bit** in the Control Register (CR0).
4. **Jump to a protected-mode code segment** to execute secure operations.
5. **Enable memory paging and multitasking**.

1.3.3. Benefits of Protected Mode

Feature	Real Mode	Protected Mode
Memory Access	1MB (Segmented)	Up to 4GB (Paged and Segmented)
Multitasking	No	Yes
Memory Protection	No	Yes
Direct Hardware Access	Yes	No (Controlled via System Calls)

Advantages:

- **Prevents applications from corrupting each other's memory** (memory isolation).
- **Supports multitasking** (multiple processes can run without interference).
- **Enhances security** (applications cannot execute privileged operations).

- **Allows use of modern OS features** (virtual memory, paging, kernel mode, and user mode).

1.3.4. Memory Protection in Protected Mode

Protected mode enforces **memory protection** using:

1. **Segmentation:**
    - Divides memory into segments (Code, Data, Stack).
    - Uses a **Global Descriptor Table (GDT)** to manage access permissions.
2. **Paging:**
    - Divides memory into fixed-size pages (e.g., 4KB).
    - Uses a **Page Table** to map virtual addresses to physical addresses.
    - Implements **Demand Paging** to load only necessary memory blocks.
3. **Ring-based Protection:**
    - User applications run in **Ring 3**, preventing direct hardware access.
    - Kernel code runs in **Ring 0**, managing system resources securely.

Example of Protected Memory Access:

- A user program (Ring 3) **cannot access kernel memory** (Ring 0) directly.
- If an application tries to access restricted memory, a **Segmentation Fault (SIGSEGV)** occurs.

Memory Protection in Protected Mode (x86 Architecture)

1. What is Protected Mode?

Protected Mode is an operational mode of **x86 processors** that provides **advanced memory management** and **process protection**. It enables features such as:
✅ **Memory Protection**
✅ **Virtual Memory**
✅ **Multitasking Support**
✅ **Segmentation and Paging**

🔹 **Before Protected Mode:**
In **Real Mode**, all programs had **direct access** to the entire memory, which meant one faulty or malicious program could **overwrite another program's memory** or even crash the entire system.

🔹 **With Protected Mode:**
The **operating system (OS)** controls memory access and ensures that each process runs in **its own isolated memory space**.

---

2. What is Memory Protection?

Memory protection in **Protected Mode** ensures that:

- A program **cannot access another program's memory**.
- User applications **cannot access kernel memory**.
- The CPU enforces **segmentation and paging** to prevent unauthorized access.

Why Is Memory Protection Important?

✅ **Prevents crashes** – One faulty program does not crash the whole system.
✅ **Enhances security** – Malware cannot modify kernel memory.
✅ **Supports multitasking** – Each process runs in its own protected space.

---

3. How Memory Protection Works in Protected Mode

Memory protection is achieved using **two key mechanisms**:

1. **Segmentation** – Divides memory into logical sections (segments).
2. **Paging** – Divides memory into fixed-size blocks (pages) for fine-grained protection.

---

4. Segmentation in Protected Mode

What is Segmentation?

Segmentation divides memory into **logical blocks called segments**. Each segment has:

- A **Segment Descriptor** that defines its **base address**, **size**, and **access rights**.
- A **Segment Selector** that points to the descriptor in the **Global Descriptor Table (GDT)**.

Segment Registers

Each segment is accessed using special CPU registers:

- **CS** (Code Segment) → Stores executable code
- **DS** (Data Segment) → Stores global variables
- **SS** (Stack Segment) → Stores stack memory
- **ES, FS, GS** → Additional segments

Example: Segmentation in Protected Mode

💡 Let's assume a program has the following memory segments:

Segment	Base Address	Limit	Access Rights
Code	`0x1000`	`64 KB`	Read-Execute
Data	`0x5000`	`32 KB`	Read-Write
Stack	`0x8000`	`16 KB`	Read-Write

If a program tries to **execute code from the Data Segment**, the CPU **throws a segmentation fault**.

C Example (Causing a Segmentation Fault)

```
#include <stdio.h>
#include <string.h>

int main() {
    char *ptr = "Hello, World!";  // This is stored in a read-only segment
    ptr[0] = 'h';  // Trying to modify a read-only string (causes segmentation fault)
    printf("%s\n", ptr);
    return 0;
}
```

🔹 **What happens?**

- The string `"Hello, World!"` is stored in the **read-only Code Segment**.
- Attempting to modify it (`ptr[0] = 'h'`) violates memory protection → **Segmentation Fault**.

---

5. Paging in Protected Mode

What is Paging?

Paging divides memory into **fixed-size blocks called pages (4KB, 2MB, 1GB, etc.)**.
The OS manages a **Page Table**, which maps **virtual addresses** (used by programs) to **physical addresses** (actual locations in RAM).

Paging vs. Segmentation

Feature	Segmentation	Paging
Memory Division	Logical Blocks (Segments)	Fixed Blocks (Pages)
Variable Size	Yes	No (Fixed)
Protection Level	Coarse-Grained	Fine-Grained

How Paging Provides Memory Protection

✅ **A process cannot access another process's page**.
✅ **Kernel pages are marked as "supervisor-only"** – User Mode (Ring 3) cannot access them.
✅ **Read-Only pages prevent accidental writes**.

Example: Page Table in Protected Mode

Assume **Process A** has the following page mappings:

Virtual Address	Physical Address	Permissions
`0x400000`	`0x1000`	Read-Write
`0x401000`	`0x2000`	Read-Only
`0x402000`	`0x3000`	Kernel-Only

If **Process A** tries to **write to `0x401000` or access `0x402000`**, the OS will generate a **page fault**.

Example: Page Fault in C (Accessing Invalid Memory)

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int *ptr = (int *)0x402000;  // Trying to access a restricted memory address
```

```
    printf("%d\n", *ptr);  // This will cause a segmentation fault
    return 0;
}
```

🔹 **What happens?**

- `0x402000` **is a protected page.**
- The OS detects unauthorized access and **triggers a page fault (segmentation fault).**

---

6. Combining Segmentation and Paging for Security

How Linux Uses Both Mechanisms

🔹 **Segmentation:**

- Used mainly for separating User Mode and Kernel Mode memory.
- **Kernel memory is restricted** to **Ring 0**.

🔹 **Paging:**

- Provides **fine-grained** memory protection.
- Enforces **virtual memory** (swap, memory isolation, etc.).
- **Prevents buffer overflows** and **ensures security**.

---

7. Summary

Feature	Segmentation	Paging
Purpose	Divides memory into logical sections	Divides memory into fixed-size blocks
Size	Variable	Fixed (usually 4 KB)
Access Control	Segment-based	Page-based
Protection	Coarse-grained	Fine-grained
Used in Linux?	Yes (but limited)	Yes (extensively)

✅ **Memory Protection in Protected Mode ensures system security and stability**
✅ **Segmentation and Paging work together to isolate processes and prevent crashes**

Conclusion

- **Linux internals provide insights into system architecture, kernel operations, and security mechanisms.**
- **Understanding privilege levels (Rings) is crucial for writing secure applications and debugging system issues.**
- **Protected mode ensures efficient memory management and process isolation, making modern OS features possible.**
