Linux Kernel Programming: From Basics to Advanced Optimization

Part 1: Introduction to Linux Kernel

1. **Understanding the Linux Kernel**
    - What is an Operating System Kernel?
    - Monolithic vs. Microkernel Architecture
    - Linux Kernel Components & Design Principles
    - Kernel Versioning and Development Process
2. **Setting Up a Kernel Development Environment**
    - Choosing the Right Kernel Version
    - Installing Required Development Tools
    - Downloading and Compiling the Kernel from Source
    - Configuring the Kernel for Development
3. **Kernel Programming Essentials**
    - Kernel Space vs. User Space
    - Memory Management in the Kernel
    - Kernel Mode vs. User Mode Execution
    - System Calls and Their Role in Linux

Part 2: Kernel Modules and Device Drivers

4. **Introduction to Loadable Kernel Modules (LKM)**
    - What are LKMs?
    - Writing a Simple "Hello Kernel" Module
    - Loading and Unloading Kernel Modules
    - Debugging Kernel Modules
5. **Character Device Drivers**
    - Understanding Character Devices
    - Implementing File Operations (`open`, `read`, `write`)
    - Registering a Character Device in the Kernel
    - Using `ioctl` for Device Communication
6. **Block Device Drivers**
    - How Block Devices Work
    - Implementing Block Read and Write Operations
    - Buffer Management for Block Devices
7. **Network Device Drivers**

- Understanding the Linux Networking Stack
- Writing a Basic Network Driver
- Handling Packets in the Kernel

Part 3: Kernel Subsystems and Internals

8. **Linux Kernel Process Management**

    - Process Descriptor (`task_struct`)
    - Process Creation and Termination
    - Context Switching in the Kernel
    - Kernel Threads vs. User Processes

9. **Linux Kernel Memory Management**

    - Physical and Virtual Memory
    - Paging and Swapping
    - Allocating Memory in the Kernel (`kmalloc`, `vmalloc`)
    - Memory Pools and Slab Allocator

10. **The Filesystem Layer**

- Understanding Virtual File System (VFS)
- File Operations and Filesystem Implementation
- Writing a Simple Filesystem

11. **The Scheduler and CPU Scheduling**

- Linux Process Scheduler Basics
- Scheduling Policies (CFS, FIFO, RR)
- Kernel Preemption and Task Priorities

Part 4: Kernel Synchronization and Concurrency

12. **Kernel Synchronization Primitives**

- Spinlocks vs. Mutexes
- Read-Copy-Update (RCU)
- Semaphore, Atomic Operations, and Barriers

13. **Interrupts and Deferred Execution**

- Handling Interrupts in the Kernel
- Writing an Interrupt Handler
- Workqueues, Tasklets, and SoftIRQs

Part 5: Kernel Debugging and Profiling

14. **Debugging the Linux Kernel**

- Kernel Logs and `dmesg`
- Using `gdb` with the Kernel (`kgdb`)

- Debugging with Ftrace and Kprobes
- Live Patching and Kernel Crash Analysis

15. **Profiling and Performance Tuning**

- Using `perf` for Performance Analysis
- BPF and eBPF for Tracing and Profiling
- Optimizing Kernel Performance for Embedded Systems

Part 6: Advanced Topics and Optimization

16. **Linux Boot Process and Optimization**

- Understanding the Boot Sequence
- Analyzing Boot Time with `systemd-analyze`
- Reducing Boot Time for Embedded Systems

17. **Writing Real-Time Linux Applications**

- Introduction to Real-Time Linux
- RT Preemption and Scheduling
- Tuning Linux for Low-Latency Applications

18. **Security in the Linux Kernel**

- Kernel Security Mechanisms
- Securing Kernel Modules
- Implementing Secure Boot and Signed Modules

Part 7: Kernel Development and Contribution

19. **Contributing to the Linux Kernel**

- Understanding the Kernel Development Workflow
- Writing Patches and Submitting to LKML
- Best Practices for Kernel Coding

20. **Final Thoughts and Further Learning**

- Recommended Books and Resources
- Future of Linux Kernel Development

Chapter 1: Understanding the Linux Kernel

1.1 What is an Operating System Kernel?

The **kernel** is the **core component** of an operating system, acting as a bridge between hardware and software. It is responsible for managing system resources such as CPU, memory, devices, and processes.

Key Responsibilities of a Kernel

Function	Description
Process Management	Schedules tasks, manages process execution
Memory Management	Allocates and frees memory, handles virtual memory
Device Management	Provides an interface to hardware via drivers
File System Management	Handles file read/write operations
Networking	Manages communication between devices and applications

How the Kernel Interacts with System Components

```
+-------------------------------------------------------+
|                 User Applications                     |
+-------------------------------------------------------+
|  System Calls (API for user space to talk to kernel) |
+-------------------------------------------------------+
|              Kernel (OS Core)                         |
|   - Process Scheduler                                 |
|   - Memory Manager                                    |
|   - File System                                       |
|   - Device Drivers                                    |
|   - Network Stack                                     |
+-------------------------------------------------------+
|               Hardware (CPU, RAM, Disk, etc.)         |
+-------------------------------------------------------+
```

User applications interact with the kernel through **system calls**, and the kernel interacts with hardware through **device drivers**.

1.2 Monolithic vs. Microkernel Architecture

There are two major types of kernel designs:

Architecture	Description	Examples

Architecture	Description	Examples
Monolithic Kernel	All OS components run in kernel space, providing better performance but higher complexity.	Linux, Windows NT, Unix
Microkernel	Minimal core functionality; services like drivers run in user space, increasing stability but adding overhead.	Minix, QNX, L4

1.2.1 Monolithic Kernel (Linux's Design)

- The entire OS runs in **kernel space**.
- Fast performance due to direct function calls.
- Any **bug in a module** can crash the system.
- Example: **Linux, Windows NT, BSD**.

Diagram of a Monolithic Kernel:

```
+-------------------------------------------------------+
|                User Space                             |
|  - Applications                                       |
+-------------------------------------------------------+
|                Kernel Space                           |
|  - Process Scheduler, Memory Manager, FS, Drivers     |
|  - Network Stack, System Calls Interface              |
+-------------------------------------------------------+
|                Hardware                               |
+-------------------------------------------------------+
```

1.2.2 Microkernel Design

- Only essential parts (CPU scheduling, memory management) are in **kernel space**.
- Drivers and services run in **user space**, reducing kernel crashes.
- Example: **Minix, QNX, L4, macOS XNU (Hybrid)**.

Diagram of a Microkernel:

```
+-------------------------------------------------------+
|                User Space                             |
|  - Applications                                       |
|  - Device Drivers, File System, Network Stack         |
+-------------------------------------------------------+
|                Kernel Space                           |
|  - Process Scheduler, Memory Manager, IPC             |
+-------------------------------------------------------+
|                Hardware                               |
+-------------------------------------------------------+
```

◈ **Why Linux Uses a Monolithic Kernel?**

- Faster execution (no need for IPC between components).
- Modern kernels like Linux support **Loadable Kernel Modules (LKM)**, making monolithic kernels **extensible like microkernels**.

1.3 Linux Kernel Components & Design Principles

The Linux kernel is modular and comprises the following key components:

Component	Description
Process Scheduler	Allocates CPU time to processes.
Memory Manager	Manages physical and virtual memory.
File System Interface	Provides access to various file systems (ext4, XFS, Btrfs).
Device Drivers	Interacts with hardware devices.
Networking Stack	Handles TCP/IP, sockets, and communication protocols.
System Call Interface	Provides APIs for user applications to interact with the kernel.

1.3.1 Process Management

The kernel uses a **scheduler** to manage process execution. Example:

```
#include <linux/sched.h>

struct task_struct *task;
for_each_process(task) {
    printk(KERN_INFO "Process: %s (PID %d)\n", task->comm, task->pid);
}
```

This iterates over all running processes and prints their names and PIDs.

1.3.2 Memory Management

The kernel handles physical and virtual memory using paging. Example of allocating kernel memory:

```
#include <linux/slab.h>

void *ptr = kmalloc(1024, GFP_KERNEL); // Allocate 1KB
if (!ptr)
    printk(KERN_ERR "Memory allocation failed\n");

kfree(ptr); // Free allocated memory
```

1.4 Kernel Versioning and Development Process

The Linux kernel follows a structured versioning and development process.

1.4.1 Kernel Versioning Scheme

A Linux kernel version looks like this:

```
6.8.12
 │ │ │
 │ │ └── Minor bug fixes/security updates
 │ └── New features and hardware support
 └── Major version bump for significant changes
```

Example:

- **Linux 6.8** → Major release with new features.
- **Linux 6.8.12** → Bug fixes and security updates.

1.4.2 The Linux Kernel Development Workflow

Linux kernel development follows a structured process:

1️⃣ New Feature Development

- Developers submit patches to the **Linux Kernel Mailing List (LKML)**.
- Code is reviewed and tested.

2️⃣ Kernel Merge Window

- Maintainers like **Linus Torvalds** merge approved patches.
- Happens every **8-10 weeks** for major versions.

3️⃣ Release Candidate (RC) Testing

- Several **release candidates (rc1, rc2, ...)** are tested before final release.

4️⃣ Stable Kernel Release

- A new kernel version is released after extensive testing.

1.4.3 Where to Get the Linux Kernel?

- Official repository: **https://www.kernel.org/**
- Clone latest kernel using Git:

  ```
  git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
  ```

1.4.4 Contributing to the Linux Kernel

To contribute code, follow these steps:

1. **Clone the Linux kernel repository:**

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
cd linux
```

2. **Make code changes and commit:**

```
git commit -s -m "Added feature X"
```

3. **Submit a patch via email:**

```
git format-patch -1 --stdout | mail -s "PATCH: Feature X" linux-kernel@vger.kernel.org
```

Conclusion

In this chapter, we covered: ✅ The role and **responsibilities of the Linux kernel** ✅ Differences between **monolithic vs. microkernel architectures** ✅ **Key components** of the Linux kernel, including process management, memory management, and device drivers ✅ The **Linux kernel development process** and how to contribute

Chapter 2: Setting Up a Kernel Development Environment

Before diving into Linux kernel programming, you need a properly configured development environment. This chapter will guide you through choosing the right kernel version, installing necessary tools, compiling the kernel from source, and configuring it for development.

2.1 Choosing the Right Kernel Version

The **Linux kernel** is constantly evolving, with new features and bug fixes in every release. You need to decide which version to use based on your development needs.

2.1.1 Kernel Version Types

Kernel Type	Description	Best Use Case
Long-Term Support (LTS)	Stable, maintained for 2+ years	Production, embedded systems
Mainline (Latest Release)	Bleeding-edge, latest features	Kernel development, testing
Distribution-Specific	Modified by distros (Ubuntu, Fedora)	General use, stability
Custom (Patched)	Manually patched with features	Specialized development

2.1.2 Finding the Latest Kernel Version

Visit kernel.org to check the latest **stable**, **LTS**, and **mainline** releases.

To check your current kernel version:

```
uname -r
```

Example output:

```
6.5.8-arch1-1
```

This indicates:

- **6.5** → Major version
- **8** → Minor update
- **arch1-1** → Distribution-specific modifications

2.2 Installing Required Development Tools

To compile and work with the Linux kernel, install the following tools.

2.2.1 Installing Essential Packages

On Debian/Ubuntu

```
sudo apt update
sudo apt install build-essential libncurses-dev bison flex libssl-dev bc
```

On Fedora

```
sudo dnf install make gcc ncurses-devel bison flex elfutils-libelf-devel openssl-devel bc
```

On Arch Linux

```
sudo pacman -Syu base-devel ncurses bison flex bc openssl
```

2.2.2 Verifying Installed Tools

Check if **GCC** and **make** are installed:

```
gcc --version
make --version
```

2.2.3 Installing Git (for Kernel Source Management)

```
sudo apt install git   # Debian-based systems
```

Verify installation:

```
git --version
```

2.3 Downloading and Compiling the Kernel from Source

2.3.1 Cloning the Linux Kernel Source Code

To get the latest kernel source:

```
git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
cd linux
```

If you want a specific kernel version:

```
git checkout v6.5
```

To download a **compressed tarball** instead:

```
wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.5.tar.xz
tar -xvf linux-6.5.tar.xz
cd linux-6.5
```

2.3.2 Configuring the Kernel for Compilation

Linux provides several ways to configure the kernel:

Method 1: Use the Current System's Configuration

```
cp /boot/config-$(uname -r) .config
make oldconfig
```

This keeps your current system's kernel configuration.

Method 2: Manual Configuration

```
make menuconfig
```

This opens a **text-based UI** for configuring kernel features.

Method 3: Default Configuration

```
make defconfig
```

This sets up a default configuration optimized for your system.

2.3.3 Compiling the Kernel

Once the configuration is set, compile the kernel using:

```
make -j$(nproc)
```

The `-j$(nproc)` flag speeds up compilation by using all available CPU cores.

Compilation Time Estimates

CPU Cores	Approximate Time
2 Cores	1-2 Hours
4 Cores	30-60 Minutes
8+ Cores	15-30 Minutes

2.3.4 Installing the Compiled Kernel

Once compiled, install modules and the kernel:

```
sudo make modules_install
sudo make install
```

Then update **GRUB** (for bootloader configuration):

```
sudo update-grub
```

Or for EFI-based systems:

```
sudo grub-mkconfig -o /boot/grub/grub.cfg
```

Reboot and select the new kernel in the GRUB menu:

```
reboot
```

Check the running kernel after reboot:

```
uname -r
```

2.4 Configuring the Kernel for Development

2.4.1 Enabling Debugging Options

For kernel development, enable **debugging options** to make debugging easier.

Run:

```
make menuconfig
```

Navigate to:

```
Kernel Hacking  --->
    [*] Kernel debugging
    [*] Compile the kernel with debug info
```

Save and exit.

2.4.2 Enabling Loadable Kernel Module (LKM) Support

To develop **kernel modules**, ensure **module loading** is enabled:

```
Device Drivers  --->
    [*] Enable loadable module support
```

2.4.3 Setting Up Kernel Logging

View kernel messages using:

```
dmesg | tail -n 20
```

Or enable **real-time logging**:

```
sudo journalctl -kf
```

2.4.4 Debugging with GDB and QEMU

To run the kernel inside a **virtual machine** for debugging:

```
qemu-system-x86_64 -kernel arch/x86/boot/bzImage -append "console=ttyS0" -
nographic
```

Then connect **GDB**:

```
gdb vmlinux
```

This allows step-by-step debugging of the kernel.

Conclusion

In this chapter, we covered: ✅ Choosing the **right kernel version** for development ✅ Installing **essential development tools** ✅ Downloading and **compiling the Linux kernel from source** ✅ Configuring the kernel for **debugging and module development**

Chapter 3: Kernel Programming Essentials

Before diving into Linux kernel programming, it is essential to understand the fundamental concepts that differentiate kernel development from user-space programming. This chapter covers **kernel space vs. user space**, **memory management in the kernel**, **kernel mode vs. user mode execution**, and **system calls**—all of which are critical for writing and debugging kernel code effectively.

3.1 Kernel Space vs. User Space

Linux, like other modern operating systems, separates execution into two primary domains:

- **User Space** → Where normal applications run, restricted from directly accessing hardware or system resources.
- **Kernel Space** → Where the operating system core operates, with full access to system resources.

3.1.1 Key Differences

Feature	User Space	Kernel Space
Access Level	Restricted (uses system calls)	Full control over hardware
Memory Access	Can only access user-space memory	Can access both user and kernel memory
Crash Impact	Affected process crashes only	System-wide crash if an error occurs
Performance	Slower due to system call overhead	Faster due to direct hardware interaction
Example Code	Standard C programs	Kernel modules, device drivers

3.1.2 How User Space Interacts with Kernel Space

User-space applications interact with the kernel via **system calls**:

1. **Application requests a system service (e.g., file I/O, memory allocation).**
2. **System call transfers control to kernel space.**
3. **Kernel processes the request and returns the result to user space.**

📌 Example: **Reading a file from user space**

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

int main() {
    int fd = open("example.txt", O_RDONLY);  // System call
    if (fd < 0) {
        perror("File open failed");
        return 1;
    }
```

```
    char buffer[100];
    read(fd, buffer, sizeof(buffer));  // System call
    printf("File Content: %s\n", buffer);

    close(fd);  // System call
    return 0;
}
```

Explanation:

- The `open()`, `read()`, and `close()` functions are **system calls** that interact with the kernel to access a file.

3.2 Memory Management in the Kernel

Unlike user-space programs that use `malloc()` and `free()`, kernel code must use specialized memory management functions.

3.2.1 Kernel Memory Allocation

The kernel provides the following functions for memory allocation:

Function	Purpose
`kmalloc(size, flags)`	Allocates contiguous memory (like `malloc()`)
`kfree(ptr)`	Frees memory allocated by `kmalloc()`
`vmalloc(size)`	Allocates large, non-contiguous memory blocks
`vfree(ptr)`	Frees memory allocated by `vmalloc()`

📌 **Example: Allocating Memory in Kernel Space**

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/slab.h>  // kmalloc, kfree

static int __init my_init(void) {
    char *ptr = kmalloc(128, GFP_KERNEL);  // Allocate 128 bytes
    if (!ptr) {
        printk(KERN_ERR "Memory allocation failed\n");
        return -ENOMEM;
    }

    strcpy(ptr, "Hello from Kernel!");
    printk(KERN_INFO "%s\n", ptr);

    kfree(ptr);  // Free memory
    return 0;
```

```
}

static void __exit my_exit(void) {
    printk(KERN_INFO "Module Unloaded\n");
}

module_init(my_init);
module_exit(my_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Kernel Memory Allocation Example");
```

Explanation:

- `kmalloc(128, GFP_KERNEL)` allocates 128 bytes in kernel space.
- `kfree(ptr)` frees the allocated memory.

3.3 Kernel Mode vs. User Mode Execution

The CPU operates in different **privilege levels** depending on whether it executes user-space or kernel-space code.

3.3.1 CPU Privilege Levels

Most modern CPUs use a **ring-based** privilege model:

```
Ring 0 (Kernel Mode)   --> Full hardware access
Ring 3 (User Mode)      --> Restricted access, must use system calls
```

- **User Mode (Ring 3)**: Applications run here. If they try to access hardware directly, a **segmentation fault** occurs.
- **Kernel Mode (Ring 0)**: The OS kernel runs with unrestricted access to hardware.

3.3.2 Transitioning Between User and Kernel Mode

A process transitions from **user mode to kernel mode** when:

- A system call is made (`read()`, `write()`, etc.).
- A hardware interrupt occurs.
- A software exception occurs (e.g., dividing by zero).

📌 Example: Checking Execution Mode in Kernel

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h>

static int __init check_mode(void) {
```

```
    if (current->mm)  // If mm is NULL, it's in kernel mode
        printk(KERN_INFO "Running in User Mode\n");
    else
        printk(KERN_INFO "Running in Kernel Mode\n");

    return 0;
}

static void __exit cleanup(void) {
    printk(KERN_INFO "Module removed\n");
}

module_init(check_mode);
module_exit(cleanup);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Check Kernel/User Mode");
```

Explanation:

- `current->mm != NULL` → User mode.
- `current->mm == NULL` → Kernel mode.

3.4 System Calls and Their Role in Linux

A **system call** is the interface between user-space applications and the kernel.

3.4.1 How System Calls Work

1. User program calls a **wrapper function** (e.g., `read()`).
2. The function executes a special **CPU instruction** (`syscall` on x86-64).
3. The processor switches to **kernel mode** and executes the requested operation.
4. The kernel returns the result back to user space.

📌 **Example: Writing a Custom System Call** Let's add a new system call that prints "Hello from Kernel!"

Step 1: Define the System Call

Modify `kernel/sys.c` and add:

```
SYSCALL_DEFINE0(hello_syscall) {
    printk(KERN_INFO "Hello from Kernel!\n");
    return 0;
}
```

Step 2: Add System Call Entry

Edit `arch/x86/entry/syscalls/syscall_64.tbl`:

```
335   common   hello_syscall    sys_hello_syscall
```

Step 3: Compile and Load the New Kernel

Recompile the kernel:

```
make -j$(nproc)
sudo make modules_install
sudo make install
sudo reboot
```

Step 4: Call the System Call from User Space

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

#define __NR_hello_syscall 335

int main() {
    syscall(__NR_hello_syscall);
    return 0;
}
```

Compile and run:

```
gcc test_syscall.c -o test_syscall
./test_syscall
```

Conclusion

In this chapter, we covered: ✅ **User space vs. kernel space** and how they interact ✅ **Kernel memory management** and safe allocation techniques ✅ **Kernel mode vs. user mode** and CPU privilege levels ✅ **System calls** and how to write a custom one

Chapter 4: Introduction to Loadable Kernel Modules (LKM)

4.1 What are Loadable Kernel Modules (LKMs)?

A **Loadable Kernel Module (LKM)** is a piece of code that can be loaded and unloaded into the Linux kernel **dynamically** without recompiling or rebooting the system. LKMs allow adding functionalities such as **device drivers, file systems, and system calls** at runtime.

4.1.1 Why Use LKMs?

- ✅ **Modular Design** – No need to modify the core kernel.
- ✅ **Efficient Development** – Can be tested independently.
- ✅ **Resource Management** – Load when needed, unload when not.
- ✅ **Custom Features** – Extend Linux functionality without rebuilding the kernel.

4.1.2 LKM vs. Built-in Kernel Code

Feature	LKM	Built-in Kernel Code
Flexibility	Can be loaded/unloaded dynamically	Requires kernel recompilation
Development Time	Fast testing and debugging	Slower due to rebuilds
Memory Usage	Uses memory only when loaded	Always occupies memory

---

4.2 Writing a Simple "Hello Kernel" Module

Let's write a basic LKM that prints `"Hello, Kernel!"` when loaded and `"Goodbye, Kernel!"` when unloaded.

4.2.1 Structure of an LKM

A minimal LKM consists of:

- **Initialization function** (`module_init`) – Runs when the module is loaded.
- **Cleanup function** (`module_exit`) – Runs when the module is unloaded.
- **Module metadata** – Provides information like author and license.

4.2.2 Creating the Source File

📌 **hello_kernel.c**

```c
#include <linux/module.h>   // Required for all LKMs
#include <linux/kernel.h>   // For printk()
#include <linux/init.h>     // For __init and __exit macros

// Initialization function (Runs when the module is loaded)
```

```c
static int __init hello_init(void) {
    printk(KERN_INFO "Hello, Kernel!\n");
    return 0;  // Return 0 indicates success
}

// Cleanup function (Runs when the module is unloaded)
static void __exit hello_exit(void) {
    printk(KERN_INFO "Goodbye, Kernel!\n");
}

// Register init and exit functions
module_init(hello_init);
module_exit(hello_exit);

// Module metadata
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Hello Kernel module");
MODULE_VERSION("1.0");
```

4.2.3 Writing the Makefile

To compile an LKM, we need a **Makefile**.

📌 **Makefile**

```
obj-m += hello_kernel.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Explanation:

- `obj-m += hello_kernel.o` → Defines the module object file.
- `make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules` → Compiles the module against the current kernel.

4.3 Loading and Unloading Kernel Modules

4.3.1 Compiling the Module

Run the following command to compile:

```
make
```

This generates a **hello_kernel.ko** file (Kernel Object file).

4.3.2 Loading the Module

Use `insmod` to load the module into the kernel:

```
sudo insmod hello_kernel.ko
```

Check if it's loaded:

```
lsmod | grep hello_kernel
```

View kernel logs:

```
dmesg | tail -n 10
```

Expected output:

```
[ 1234.5678 ] Hello, Kernel!
```

4.3.3 Unloading the Module

To remove the module:

```
sudo rmmod hello_kernel
```

Check logs again:

```
dmesg | tail -n 10
```

Expected output:

```
[ 5678.1234 ] Goodbye, Kernel!
```

4.3.4 Listing Loaded Modules

To see all loaded modules:

```
lsmod
```

4.3.5 Viewing Module Information

To get module details:

```
modinfo hello_kernel.ko
```

Output:

```
filename:       /home/user/hello_kernel.ko
license:        GPL
author:         Your Name
description:    A simple Hello Kernel module
version:        1.0
```

4.4 Debugging Kernel Modules

Since print statements (`printf`) don't work in the kernel, we use `printk()`.

4.4.1 Using printk for Debugging

Modify the `hello_kernel.c` module:

```
printk(KERN_DEBUG "Debug message: Module loaded\n");
```

Logging Levels in printk

Level	Macro	Purpose
0	`KERN_EMERG`	System is unusable
1	`KERN_ALERT`	Immediate action needed
2	`KERN_CRIT`	Critical condition
3	`KERN_ERR`	Error condition
4	`KERN_WARNING`	Warning condition
5	`KERN_NOTICE`	Normal but significant
6	`KERN_INFO`	Informational messages
7	`KERN_DEBUG`	Debugging messages

View debug logs:

```
dmesg | grep "Debug message"
```

4.4.2 Using /proc for Debugging

The `/proc` filesystem provides runtime kernel information.

Check running kernel modules:

```
cat /proc/modules
```

4.4.3 Enabling Dynamic Debugging

To enable **dynamic debugging**, use:

```
echo 'module hello_kernel +p' > /sys/kernel/debug/dynamic_debug/control
```

This enables **printk debug messages** for our module.

4.4.4 Debugging with GDB

For deeper debugging, use GDB with QEMU:

```
qemu-system-x86_64 -kernel bzImage -initrd initramfs.img -append "console=ttyS0" -
s -S
```

Then connect with GDB:

```
gdb vmlinux
target remote :1234
```

Conclusion

In this chapter, we covered: ✅ What **Loadable Kernel Modules (LKMs)** are and their advantages. ✅ Writing a **simple Hello Kernel module**. ✅ **Compiling, loading, and unloading** an LKM. ✅ Debugging **kernel modules** using `printk`, `/proc`, and **dynamic debugging**.

Chapter 5: Character Device Drivers

5.1 Understanding Character Devices

A **character device** is a type of device that transmits data **one character at a time**, similar to how a file is read or written sequentially. Examples include **serial ports, keyboards, and sound cards**.

5.1.1 Character Devices vs. Block Devices

Feature	Character Device	Block Device
Data Handling	Byte-by-byte (sequential)	Block-by-block
Buffering	No internal buffering	Uses internal buffers
Examples	Keyboards, Mice, Serial Ports	Hard Disks, SSDs
Access	Can be accessed at any time	Must wait for blocks to be read/written

5.1.2 Device Files in `/dev` Directory

Character devices appear in the `/dev/` directory. Run the following command to list character devices:

```
ls -l /dev | grep '^c'
```

Example output:

```
crw-rw---- 1 root audio 14,  4 Feb 19 10:00 /dev/audio
crw-rw-rw- 1 root tty   5,  0 Feb 19 10:00 /dev/tty
```

- The **'c'** at the beginning indicates a **character device**.
- The **major number** (e.g., `5` for `/dev/tty`) identifies the driver.
- The **minor number** (`0`) differentiates devices handled by the same driver.

5.2 Implementing File Operations (`open`, `read`, `write`)

A **character device driver** provides implementations for **file operations** like `open()`, `read()`, `write()`, and `close()`.

5.2.1 File Operations Structure

A character device driver defines a `file_operations` structure that links system calls to driver functions.

```
static struct file_operations fops = {
    .owner = THIS_MODULE,
```

```
    .open = device_open,
    .read = device_read,
    .write = device_write,
    .release = device_release,
};
```

- `.open` → Runs when the device is opened (`open("/dev/mydevice")`).
- `.read` → Runs when `read()` is called.
- `.write` → Runs when `write()` is called.
- `.release` → Runs when the device is closed.

5.2.2 Writing the Device Operations

📌 **Simple Character Device Driver**

```c
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/uaccess.h>  // Copy data between user and kernel space

#define DEVICE_NAME "mychardev"

static int major;
static char message[256] = {0};

static int device_open(struct inode *inode, struct file *file) {
    printk(KERN_INFO "Device opened\n");
    return 0;
}

static ssize_t device_read(struct file *file, char __user *buffer, size_t len,
loff_t *offset) {
    int bytes_read = simple_read_from_buffer(buffer, len, offset, message,
strlen(message));
    printk(KERN_INFO "Device read: %s\n", message);
    return bytes_read;
}

static ssize_t device_write(struct file *file, const char __user *buffer, size_t
len, loff_t *offset) {
    if (copy_from_user(message, buffer, len)) {
        return -EFAULT;
    }
    message[len] = '\0';  // Ensure null termination
    printk(KERN_INFO "Device write: %s\n", message);
    return len;
}

static int device_release(struct inode *inode, struct file *file) {
    printk(KERN_INFO "Device closed\n");
    return 0;
}
```

```
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = device_open,
    .read = device_read,
    .write = device_write,
    .release = device_release,
};

static int __init char_init(void) {
    major = register_chrdev(0, DEVICE_NAME, &fops);
    if (major < 0) {
        printk(KERN_ERR "Failed to register character device\n");
        return major;
    }
    printk(KERN_INFO "Registered char device with major number %d\n", major);
    return 0;
}

static void __exit char_exit(void) {
    unregister_chrdev(major, DEVICE_NAME);
    printk(KERN_INFO "Unregistered char device\n");
}

module_init(char_init);
module_exit(char_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple character device driver");
```

5.3 Registering a Character Device in the Kernel

5.3.1 Compiling the Module

Create a **Makefile** to build the module: 📌 **Makefile**

```
obj-m += mychardev.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Compile the module:

```
make
```

This generates **mychardev.ko**.

5.3.2 Loading the Module

```
sudo insmod mychardev.ko
dmesg | tail -n 5
```

Expected output:

```
Registered char device with major number 240
```

5.3.3 Creating a Device File

Find the major number:

```
cat /proc/devices | grep mychardev
```

Create the device file:

```
sudo mknod /dev/mychardev c 240 0
sudo chmod 666 /dev/mychardev
```

5.3.4 Testing the Character Device

Write to the device:

```
echo "Hello, kernel!" > /dev/mychardev
```

Read from the device:

```
cat /dev/mychardev
```

Expected output:

```
Hello, kernel!
```

Unload the module:

```
sudo rmmod mychardev
```

5.4 Using `ioctl` for Device Communication

5.4.1 What is `ioctl`?

The `ioctl` (Input/Output Control) system call allows **sending custom commands** to a device driver.

5.4.2 Defining `ioctl` Commands

Modify the character device driver to support `ioctl`.

📌 **Adding `ioctl` Support**

```c
#define IOCTL_SET_MSG _IOR('k', 1, char *)

static long device_ioctl(struct file *file, unsigned int cmd, unsigned long arg) {
    switch (cmd) {
        case IOCTL_SET_MSG:
            if (copy_from_user(message, (char __user *)arg, sizeof(message))) {
                return -EFAULT;
            }
            printk(KERN_INFO "ioctl received: %s\n", message);
            break;
        default:
            return -EINVAL;
    }
    return 0;
}

static struct file_operations fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = device_ioctl,
};
```

5.4.3 Writing a User-Space Program to Call `ioctl`

📌 **test_ioctl.c**

```c
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <string.h>

#define IOCTL_SET_MSG _IOR('k', 1, char *)

int main() {
```

```
    int fd = open("/dev/mychardev", O_WRONLY);
    if (fd < 0) {
        perror("open");
        return 1;
    }

    char msg[] = "Message via ioctl";
    ioctl(fd, IOCTL_SET_MSG, msg);

    close(fd);
    return 0;
}
```

Compile and run:

```
gcc test_ioctl.c -o test_ioctl
./test_ioctl
```

Check logs:

```
dmesg | tail -n 5
```

Expected output:

```
ioctl received: Message via ioctl
```

Conclusion

In this chapter, we covered: ✅ **What character devices are** and how they differ from block devices. ✅ **Implementing file operations (`open`, `read`, `write`).** ✅ **Registering a character device and testing it.** ✅ **Using `ioctl` for device communication.**

Chapter 6: Block Device Drivers

6.1 How Block Devices Work

A **block device** is a type of device that transfers data in **fixed-size blocks** rather than character-by-character. These devices are mainly used for **storage**, such as **hard drives, SSDs, USB drives, and SD cards**.

6.1.1 Key Characteristics of Block Devices

✅ **Buffered I/O** – Data is stored in kernel buffers before being written to disk. ✅ **Random Access** – Supports reading/writing from any position. ✅ **File System Support** – Block devices can be formatted with **ext4, xfs, etc.**

6.1.2 Block Devices vs. Character Devices

Feature	Block Device	Character Device
Data Handling	Blocks (512 bytes or more)	Bytes (1 byte at a time)
Buffering	Uses kernel buffers	No internal buffering
Access Pattern	Random or sequential	Always sequential
Examples	HDD, SSD, SD card	Keyboard, Mouse, Serial Port

6.1.3 Viewing Block Devices in Linux

List all block devices:

```
lsblk
```

Show details about a specific block device:

```
cat /sys/class/block/sda/size
```

6.2 Implementing Block Read and Write Operations

A **block device driver** must register with the **block layer** and implement **read/write** operations.

6.2.1 Registering a Block Device

📌 Basic Structure of a Block Device Driver

```
#include <linux/module.h>
#include <linux/fs.h>
```

```
#include <linux/genhd.h>
#include <linux/blkdev.h>

#define DEVICE_NAME "myblockdev"
#define SECTOR_SIZE 512
#define NUM_SECTORS 1024  // 512 KB storage

static struct request_queue *queue;
static struct gendisk *my_disk;
static char *device_data;

// Request handling function
static void myblock_request(struct request_queue *q) {
    struct request *req;
    while ((req = blk_fetch_request(q)) != NULL) {
        struct bio_vec bvec;
        struct req_iterator iter;
        sector_t sector = blk_rq_pos(req);
        size_t len = blk_rq_bytes(req);

        rq_for_each_segment(bvec, req, iter) {
            void *buffer = page_address(bvec.bv_page) + bvec.bv_offset;
            size_t size = bvec.bv_len;

            if (rq_data_dir(req) == WRITE) {
                memcpy(device_data + (sector * SECTOR_SIZE), buffer, size);
            } else {
                memcpy(buffer, device_data + (sector * SECTOR_SIZE), size);
            }
        }
        __blk_end_request_all(req, 0);
    }
}

// Block device operations
static struct block_device_operations myblock_ops = {
    .owner = THIS_MODULE,
};

// Module initialization
static int __init myblock_init(void) {
    device_data = vmalloc(NUM_SECTORS * SECTOR_SIZE);
    queue = blk_init_queue(myblock_request, NULL);

    my_disk = alloc_disk(1);
    strcpy(my_disk->disk_name, DEVICE_NAME);
    my_disk->major = register_blkdev(0, DEVICE_NAME);
    my_disk->first_minor = 0;
    my_disk->fops = &myblock_ops;
    my_disk->queue = queue;
    set_capacity(my_disk, NUM_SECTORS);

    add_disk(my_disk);
    printk(KERN_INFO "Block device registered\n");
```

```
    return 0;
}

// Module cleanup
static void __exit myblock_exit(void) {
    del_gendisk(my_disk);
    put_disk(my_disk);
    unregister_blkdev(my_disk->major, DEVICE_NAME);
    blk_cleanup_queue(queue);
    vfree(device_data);
    printk(KERN_INFO "Block device unregistered\n");
}

module_init(myblock_init);
module_exit(myblock_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple block device driver");
```

6.3 Buffer Management for Block Devices

6.3.1 Using Kernel Buffers

The **bio (Block I/O)** structure is used for block device buffering. Kernel provides **bio_vec** to manage data buffers.

Example of reading a block from the buffer:

```
struct bio_vec bvec;
struct req_iterator iter;
rq_for_each_segment(bvec, req, iter) {
    void *buffer = page_address(bvec.bv_page) + bvec.bv_offset;
    memcpy(buffer, device_data + (sector * SECTOR_SIZE), bvec.bv_len);
}
```

6.3.2 Using the Page Cache

The **page cache** improves performance by caching recently accessed data. To read/write using the page cache:

```
struct page *page = alloc_page(GFP_KERNEL);
void *page_addr = kmap(page);
memcpy(page_addr, device_data, PAGE_SIZE);
kunmap(page);
```

6.3.3 Using `blk_queue_make_request()` for Advanced Queues

For optimized buffering, the block layer allows configuring request queues:

```
blk_queue_make_request(queue, myblock_request);
blk_queue_logical_block_size(queue, SECTOR_SIZE);
```

6.4 Testing the Block Device Driver

6.4.1 Compiling and Loading the Module

Create a **Makefile**:

```
obj-m += myblockdev.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Compile the driver:

```
make
```

Load the module:

```
sudo insmod myblockdev.ko
dmesg | tail -n 5
```

Check if the device is registered:

```
lsblk
```

6.4.2 Creating a Device File

Find the **major number**:

```
cat /proc/devices | grep myblockdev
```

Create the block device:

```
sudo mknod /dev/myblockdev b 250 0
sudo chmod 666 /dev/myblockdev
```

6.4.3 Formatting the Block Device

Format it with **ext4**:

```
sudo mkfs.ext4 /dev/myblockdev
```

Mount the device:

```
sudo mkdir /mnt/myblock
sudo mount /dev/myblockdev /mnt/myblock
```

Write a test file:

```
echo "Hello, block device!" > /mnt/myblock/test.txt
```

Unmount and unload:

```
sudo umount /mnt/myblock
sudo rmmod myblockdev
```

Conclusion

In this chapter, we covered: ✅ **How block devices work** and their differences from character devices. ✅ **Implementing read and write operations** for block devices. ✅ **Buffer management** techniques like bio_vec and page cache. ✅ **Registering a block device and testing it.**

Chapter 7: Network Device Drivers

7.1 Understanding the Linux Networking Stack

A **network device driver** manages a network interface and handles packet transmission and reception. The **Linux networking stack** follows the **OSI model** and supports various protocols (TCP/IP, UDP, etc.).

7.1.1 The Linux Networking Stack Overview

The Linux network stack consists of multiple layers:

1. **User Space Applications** (e.g., `ping`, `curl`, `netcat`)
2. **Socket Layer** (e.g., BSD sockets, `send()`, `recv()`)
3. **Transport Layer** (TCP, UDP)
4. **Network Layer** (IP)
5. **Link Layer** (Ethernet, Wi-Fi)
6. **Hardware Drivers**

📌 **Packet Flow in the Linux Kernel:**

```
User Process → Sockets → Transport Layer (TCP/UDP) → Network Layer (IP) → Device
Driver → Network Card
```

7.2 Writing a Basic Network Driver

7.2.1 Registering a Network Device

A **network driver** registers a `net_device` structure and implements essential functions.

📌 **Basic Network Driver Skeleton**

```
#include <linux/module.h>
#include <linux/netdevice.h>

#define DRIVER_NAME "mynet"

static struct net_device *my_netdev;

// Open function (called when interface is activated)
static int my_open(struct net_device *dev) {
    netif_start_queue(dev);
    printk(KERN_INFO "Network device opened\n");
    return 0;
}

// Stop function (called when interface is deactivated)
```

```c
static int my_stop(struct net_device *dev) {
    netif_stop_queue(dev);
    printk(KERN_INFO "Network device stopped\n");
    return 0;
}

// Transmit function (called when sending a packet)
static netdev_tx_t my_xmit(struct sk_buff *skb, struct net_device *dev) {
    printk(KERN_INFO "Packet transmitted\n");
    dev_kfree_skb(skb);  // Free memory after transmission
    return NETDEV_TX_OK;
}

// Network operations structure
static const struct net_device_ops my_netdev_ops = {
    .ndo_open = my_open,
    .ndo_stop = my_stop,
    .ndo_start_xmit = my_xmit,
};

// Module initialization
static int __init my_net_init(void) {
    my_netdev = alloc_netdev(0, DRIVER_NAME, NET_NAME_UNKNOWN, ether_setup);
    if (!my_netdev)
        return -ENOMEM;

    my_netdev->netdev_ops = &my_netdev_ops;
    if (register_netdev(my_netdev)) {
        printk(KERN_ERR "Failed to register network device\n");
        free_netdev(my_netdev);
        return -ENODEV;
    }

    printk(KERN_INFO "Network device registered\n");
    return 0;
}

// Module cleanup
static void __exit my_net_exit(void) {
    unregister_netdev(my_netdev);
    free_netdev(my_netdev);
    printk(KERN_INFO "Network device unregistered\n");
}

module_init(my_net_init);
module_exit(my_net_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Basic Network Device Driver");
```

7.3 Handling Packets in the Kernel

7.3.1 Receiving Packets

The kernel provides a **struct sk_buff** (socket buffer) to handle network packets.

📌 Packet Reception Example

```
static int my_receive(struct sk_buff *skb, struct net_device *dev, struct
packet_type *pt, struct net_device *orig_dev) {
    printk(KERN_INFO "Packet received: length=%d\n", skb->len);
    kfree_skb(skb);  // Free the packet after processing
    return 0;
}

// Register packet handler
static struct packet_type my_proto = {
    .type = htons(ETH_P_ALL),  // Capture all Ethernet packets
    .func = my_receive,
};

static int __init my_packet_init(void) {
    dev_add_pack(&my_proto);
    return 0;
}

static void __exit my_packet_exit(void) {
    dev_remove_pack(&my_proto);
}

module_init(my_packet_init);
module_exit(my_packet_exit);
```

7.3.2 Transmitting Packets

Network drivers must **fill the sk_buff structure** before sending data.

📌 Packet Transmission Example

```
static netdev_tx_t my_xmit(struct sk_buff *skb, struct net_device *dev) {
    printk(KERN_INFO "Transmitting packet of length: %d\n", skb->len);
    dev_kfree_skb(skb);
    return NETDEV_TX_OK;
}
```

7.4 Testing the Network Driver

7.4.1 Compiling and Loading the Module

📌 **Makefile**

```
obj-m += mynet.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Compile:

```
make
```

Load the module:

```
sudo insmod mynet.ko
dmesg | tail -n 5
```

Check if the device is registered:

```
ip link show mynet
```

7.4.2 Assigning an IP Address

```
sudo ip link set mynet up
sudo ip addr add 192.168.1.100/24 dev mynet
ping -I mynet 8.8.8.8
```

7.4.3 Capturing Packets

Use `tcpdump` to inspect network traffic:

```
sudo tcpdump -i mynet
```

Conclusion

In this chapter, we covered: ✅ **Linux networking stack and its interaction with network drivers** ✅ **Writing a basic network device driver** ✅ **Handling packet transmission and reception** ✅ **Testing the**

driver using `ip link` and `tcpdump`

Chapter 8: Linux Kernel Process Management

Process management is one of the core functionalities of the **Linux kernel**, responsible for handling **process creation, scheduling, and termination**. This chapter explores **how the kernel represents processes**, **how they are created and terminated**, and **the difference between user processes and kernel threads**.

8.1 Process Descriptor (`task_struct`)

The **Linux kernel** represents each process with a **process descriptor**, which is stored in a structure called `task_struct`.

8.1.1 The `task_struct` Structure

The `task_struct` structure holds **essential information about a process**, such as its **state, process ID, parent process, memory information, scheduling policies, and more**.

📌 **Key Fields in `task_struct`**

Field	Description
`pid`	Process ID
`state`	Process state (running, sleeping, stopped, etc.)
`comm`	Process name
`parent`	Pointer to parent process
`mm`	Memory descriptor (maps user space memory)
`files`	Open file descriptors
`prio`	Process priority
`policy`	Scheduling policy (e.g., real-time, normal)

8.1.2 Viewing Process Information

The kernel provides **procfs** to expose process details. You can view a process's `task_struct` using:

```
cat /proc/<PID>/status
```

Example:

```
cat /proc/1/status  # Check process 1 (init/systemd)
```

8.2 Process Creation and Termination

8.2.1 Process Creation: `fork()`, `vfork()`, and `clone()`

In Linux, a **new process** is created using **system calls** such as `fork()`, `vfork()`, and `clone()`.

📌 How `fork()` Works

- **The parent process calls `fork()`**, which creates a **child process**.
- The child gets a **copy of the parent's memory space**.
- Both parent and child continue execution.

📌 Key Differences Between `fork()`, `vfork()`, and `clone()`

System Call	Copy Address Space?	Execution Behavior
`fork()`	Yes (copy-on-write)	Parent and child execute separately
`vfork()`	No	Child runs first, parent waits
`clone()`	Selective (shared memory, file descriptors)	Used for threads and namespaces

📌 Example of `fork()` in User Space

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

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child process: PID=%d\n", getpid());
    } else {
        printf("Parent process: PID=%d, Child PID=%d\n", getpid(), pid);
    }
    return 0;
}
```

Compile and run:

```
gcc fork_example.c -o fork_example
./fork_example
```

8.2.2 Process Creation Inside the Kernel (`do_fork()`)

In the kernel, `fork()` is implemented as:

```c
long do_fork(unsigned long clone_flags, unsigned long stack_start,
             unsigned long stack_size, int __user *parent_tidptr,
```

```
            int __user *child_tidptr)
```

- The kernel uses `copy_process()` to duplicate the process descriptor.
- The new process is added to the **process list**.
- The child starts execution after **context switch**.

8.2.3 Process Termination

A process **terminates** when it calls `exit()`. In the kernel, it's handled by:

```c
void do_exit(long code);
```

- **Zombie processes** occur if the parent does not read the exit status of the child.

📌 Example of `waitpid()` to Handle Child Process Termination

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child process exiting...\n");
        _exit(0);
    } else {
        waitpid(pid, NULL, 0);  // Parent waits for child
        printf("Child terminated, parent continues.\n");
    }
    return 0;
}
```

8.3 Context Switching in the Kernel

8.3.1 What is Context Switching?

A **context switch** occurs when the CPU switches from **one process/thread to another**.

8.3.2 Steps in Context Switching

1. **Save the current process state (`task_struct`)**
2. **Load the new process state**
3. **Update the CPU registers**

📌 Key Kernel Functions for Context Switching

- `schedule()` → Calls the **scheduler** to decide the next process.

- `switch_to(prev, next, last)` → Switches the CPU state.
- `context_switch()` → Saves and restores CPU registers.

8.3.3 Checking Context Switches in Linux

View the number of context switches:

```
cat /proc/<PID>/status | grep ctxt
```

Example:

```
cat /proc/1/status | grep ctxt  # Check init/systemd process context switches
```

8.4 Kernel Threads vs. User Processes

8.4.1 Differences Between Kernel Threads and User Processes

Feature	Kernel Threads	User Processes
Runs in	Kernel space	User space
Uses `task_struct`?	Yes	Yes
Memory	No separate memory space	Own memory space
Example	`kworker`, `kswapd`	`bash`, `firefox`

8.4.2 Creating a Kernel Thread

A **kernel thread** is created using `kthread_create()`.

📌 **Example: Creating a Simple Kernel Thread**

```
#include <linux/module.h>
#include <linux/kthread.h>

static struct task_struct *my_thread;

static int my_thread_fn(void *data) {
    while (!kthread_should_stop()) {
        printk(KERN_INFO "Kernel thread running...\n");
        ssleep(2);
    }
    return 0;
}

static int __init my_init(void) {
```

```
    my_thread = kthread_run(my_thread_fn, NULL, "my_kthread");
    return 0;
}

static void __exit my_exit(void) {
    kthread_stop(my_thread);
    printk(KERN_INFO "Kernel thread stopped\n");
}

module_init(my_init);
module_exit(my_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Simple Kernel Thread Example");
```

Compile and load:

```
make
sudo insmod my_kthread.ko
dmesg | tail -n 5
```

Stop and remove:

```
sudo rmmod my_kthread
```

Conclusion

In this chapter, we covered: ✅ **How Linux represents processes using `task_struct`** ✅ **How processes are created using `fork()`, `vfork()`, and `clone()`** ✅ **How context switching works in the kernel** ✅ **Differences between user processes and kernel threads** ✅ **Writing a simple kernel thread**

Chapter 9: Linux Kernel Memory Management

Memory management is one of the most crucial components of the **Linux kernel**, responsible for handling **physical and virtual memory, paging, memory allocation, and optimizations**. In this chapter, we explore **how the kernel manages memory efficiently** and how developers can **allocate and free memory safely in kernel space**.

9.1 Physical and Virtual Memory

9.1.1 Physical vs. Virtual Memory

The Linux kernel manages two types of memory:

Type	Description
Physical Memory	The actual RAM installed in the system.
Virtual Memory	An abstraction layer that allows each process to see its own memory space, independent of physical RAM.

9.1.2 How Virtual Memory Works in Linux

- Each **process has its own virtual address space**, which is mapped to physical memory.
- The **Memory Management Unit (MMU)** translates **virtual addresses** to **physical addresses** using **page tables**.
- If a process accesses memory that is **not mapped**, a **page fault** occurs.

📌 **Checking Memory Usage in Linux**

```
free -h  # Show total, used, and available memory
cat /proc/meminfo  # Detailed memory statistics
```

9.2 Paging and Swapping

9.2.1 What is Paging?

Paging is a technique that divides memory into **fixed-size pages** (usually **4 KB**).

- The **kernel and user processes** operate on virtual addresses.
- The **page table** maps **virtual pages** to **physical pages**.
- If a required page is **not in memory**, a **page fault** occurs, and the kernel **loads it from disk (swap space)**.

📌 **Checking Page Table Entries (PTE)**

```
cat /proc/pid/maps  # View process memory mapping
```

9.2.2 Swapping in Linux

Swapping moves **inactive memory pages** from RAM to disk (**swap space**) to **free up RAM**.

📌 **Check Swap Usage**

```
swapon --summary  # Show swap usage
cat /proc/swaps   # Detailed swap information
```

📌 **Enable a Swap File**

```
sudo fallocate -l 1G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
```

9.3 Allocating Memory in the Kernel (`kmalloc`, `vmalloc`)

9.3.1 `kmalloc()` – Allocating Small Memory Blocks

The `kmalloc()` function is used to allocate **physically contiguous memory** for small objects (<128 KB).

📌 **Example: Allocating and Freeing Memory Using `kmalloc()`**

```
#include <linux/module.h>
#include <linux/slab.h>  // For kmalloc and kfree

static int __init my_init(void) {
    char *buffer = kmalloc(128, GFP_KERNEL);
    if (!buffer) {
        printk(KERN_ERR "Failed to allocate memory\n");
        return -ENOMEM;
    }
    printk(KERN_INFO "Memory allocated at %p\n", buffer);

    kfree(buffer);
    printk(KERN_INFO "Memory freed\n");
    return 0;
}

static void __exit my_exit(void) {}

module_init(my_init);
```

```c
module_exit(my_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Example of kmalloc()");
```

📌 **Key Flags for `kmalloc()`**

Flag	Description
`GFP_KERNEL`	Normal allocation (can sleep)
`GFP_ATOMIC`	Allocation in interrupt context (cannot sleep)

9.3.2 `vmalloc()` – Allocating Large Memory Blocks

For **large allocations** (>128 KB), the kernel provides `vmalloc()`, which allocates **virtually contiguous memory** but not **physically contiguous**.

📌 **Example: Using `vmalloc()`**

```c
#include <linux/module.h>
#include <linux/vmalloc.h>

static int __init my_init(void) {
    char *vbuffer = vmalloc(1 * 1024 * 1024);  // Allocate 1MB
    if (!vbuffer) {
        printk(KERN_ERR "Failed to allocate memory\n");
        return -ENOMEM;
    }
    printk(KERN_INFO "Memory allocated at %p\n", vbuffer);

    vfree(vbuffer);
    printk(KERN_INFO "Memory freed\n");
    return 0;
}

static void __exit my_exit(void) {}

module_init(my_init);
module_exit(my_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Example of vmalloc()");
```

📌 **Difference Between `kmalloc()` and `vmalloc()`**

Feature	`kmalloc()`	`vmalloc()`

Feature	`kmalloc()`	`vmalloc()`
Memory Type	Physically contiguous	Virtually contiguous
Performance	Faster	Slower
Usage	Small allocations	Large allocations

9.4 Memory Pools and Slab Allocator

9.4.1 What is a Slab Allocator?

The **slab allocator** is optimized for allocating **frequently used objects** (e.g., file descriptors, inodes). It reduces **fragmentation** and improves performance.

📌 Key Functions in the Slab Allocator

Function	Description
`kmem_cache_create()`	Create a memory cache
`kmem_cache_alloc()`	Allocate an object from the cache
`kmem_cache_free()`	Free an object back to the cache
`kmem_cache_destroy()`	Destroy a memory cache

9.4.2 Example: Creating a Slab Cache

```c
#include <linux/module.h>
#include <linux/slab.h>

static struct kmem_cache *my_cache;

static int __init my_init(void) {
    my_cache = kmem_cache_create("my_cache", 128, 0, SLAB_HWCACHE_ALIGN, NULL);
    if (!my_cache) {
        printk(KERN_ERR "Failed to create cache\n");
        return -ENOMEM;
    }
    void *obj = kmem_cache_alloc(my_cache, GFP_KERNEL);
    if (!obj) {
        printk(KERN_ERR "Failed to allocate object\n");
        return -ENOMEM;
    }

    printk(KERN_INFO "Object allocated at %p\n", obj);
    kmem_cache_free(my_cache, obj);
    return 0;
}

static void __exit my_exit(void) {
```

```
    kmem_cache_destroy(my_cache);
}

module_init(my_init);
module_exit(my_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Example of Slab Allocator");
```

📌 Checking Slab Allocator Statistics

```
cat /proc/slabinfo  # Show slab cache usage
```

Conclusion

In this chapter, we covered: ✅ **How the kernel manages physical and virtual memory** ✅ **Paging and swapping mechanisms** ✅ **Memory allocation using `kmalloc()` and `vmalloc()`** ✅ **Using the slab allocator for efficient object allocation**

Chapter 10: The Filesystem Layer

The **filesystem layer** is an essential part of the Linux kernel, responsible for managing file access, storage devices, and organizing data efficiently. The Linux **Virtual File System (VFS)** provides an abstraction that allows multiple filesystems (EXT4, XFS, Btrfs, etc.) to coexist while sharing a common API.

In this chapter, we explore: ✅ How the **Virtual File System (VFS)** works ✅ File operations and how the **kernel interacts with filesystems** ✅ Writing a **simple filesystem module**

10.1 Understanding the Virtual File System (VFS)

10.1.1 What is the VFS?

The **Virtual File System (VFS)** is an **abstraction layer** that provides a common API for different filesystems (EXT4, FAT, NFS, etc.).

- Applications interact with the **VFS API** without needing to know the specifics of the underlying filesystem.
- The VFS translates file operations to the **corresponding filesystem implementation**.

📌 **Linux Filesystem Structure**

```
/        → Root filesystem
/bin     → Essential binaries
/etc     → System configuration files
/home    → User home directories
/dev     → Device files (managed via VFS)
```

📌 **Filesystem Hierarchy in the Kernel**

Layer	Description
User Space	Applications use system calls (`open()`, `read()`, `write()`, etc.).
VFS Layer	Translates system calls into filesystem-specific operations.
Filesystem Drivers	Implements specific filesystem operations (EXT4, XFS, FAT).
Block Layer	Manages disk access and caching.

📌 **Checking Mounted Filesystems**

```
mount | column -t  # Show mounted filesystems
cat /proc/mounts   # Alternative way to check
```

10.2 File Operations and Filesystem Implementation

10.2.1 Key Structures in VFS

Structure	Purpose
`struct file_operations`	Defines file-related system calls (read, write, open, close).
`struct inode`	Represents a file in the filesystem.
`struct dentry`	Represents a directory entry (filename-to-inode mapping).
`struct super_block`	Represents a mounted filesystem.

10.2.2 `struct file_operations` – Defining File Operations

The `file_operations` **structure** defines how a file behaves when accessed.

📌 Example: Implementing Basic File Operations

```c
#include <linux/fs.h>

static ssize_t myfs_read(struct file *file, char __user *buf, size_t len, loff_t
*offset) {
    printk(KERN_INFO "Reading from my filesystem\n");
    return 0;  // No actual data read
}

static ssize_t myfs_write(struct file *file, const char __user *buf, size_t len,
loff_t *offset) {
    printk(KERN_INFO "Writing to my filesystem\n");
    return len;  // Pretend to accept all data
}

static struct file_operations myfs_fops = {
    .read = myfs_read,
    .write = myfs_write,
};
```

📌 Key File Operations

Function	Purpose
`read`	Reads data from a file.
`write`	Writes data to a file.
`open`	Opens a file.
`release`	Closes a file.
`llseek`	Moves the file offset.

10.3 Writing a Simple Filesystem

In this section, we implement a **basic Linux filesystem module** that demonstrates how filesystems work at the kernel level.

10.3.1 Registering a Filesystem

A **filesystem module** must be registered with the kernel using `register_filesystem()`.

📌 **Example: Basic Filesystem Registration**

```c
#include <linux/module.h>
#include <linux/fs.h>

static struct file_system_type myfs_type = {
    .owner = THIS_MODULE,
    .name = "myfs",
    .mount = simple_mount,
    .kill_sb = kill_block_super,
};

static int __init myfs_init(void) {
    int ret = register_filesystem(&myfs_type);
    if (ret != 0) {
        printk(KERN_ERR "Failed to register filesystem\n");
        return ret;
    }
    printk(KERN_INFO "Filesystem registered successfully\n");
    return 0;
}

static void __exit myfs_exit(void) {
    unregister_filesystem(&myfs_type);
    printk(KERN_INFO "Filesystem unregistered\n");
}

module_init(myfs_init);
module_exit(myfs_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Basic Linux Filesystem");
```

10.3.2 Mounting the Filesystem

Once loaded, the new filesystem can be **mounted** as follows:

```
sudo mount -t myfs none /mnt
```

To unmount:

```
sudo umount /mnt
```

Conclusion

In this chapter, we covered: ✅ **How the Linux Virtual File System (VFS) works** ✅ **Key structures and file operations (`file_operations`, `inode`, `super_block`)** ✅ **How to create a simple filesystem in the kernel**

Chapter 11: The Scheduler and CPU Scheduling

The **Linux process scheduler** is responsible for managing how CPU time is allocated among multiple processes. Efficient scheduling ensures smooth system performance and responsiveness.

In this chapter, we will explore: ✅ How the **Linux process scheduler** works ✅ Different **scheduling policies** (CFS, FIFO, RR) ✅ **Kernel preemption** and how task priorities are handled

---

11.1 Linux Process Scheduler Basics

11.1.1 What is CPU Scheduling?

- The **CPU scheduler** determines **which process** gets CPU time.
- When multiple processes are ready, the scheduler **chooses one** to execute.
- The scheduler follows a **policy** to decide **when** and **how long** a process runs.

📌 **Types of Scheduling in Linux**

Type	Description
Preemptive Scheduling	A process can be interrupted and replaced by another process.
Non-Preemptive Scheduling	Once a process starts, it runs until it voluntarily yields the CPU.

11.1.2 The Linux Scheduling System

- The Linux scheduler is based on the **Completely Fair Scheduler (CFS)**.
- It maintains a **run queue** of processes waiting for CPU time.
- The scheduler picks the **process with the lowest "virtual runtime"** (how long it has run).

📌 **Checking Current Scheduler**

```
cat /sys/kernel/debug/sched_features
```

11.2 Scheduling Policies (CFS, FIFO, RR)

Linux supports multiple scheduling policies, each designed for different use cases.

11.2.1 Completely Fair Scheduler (CFS)

The **CFS scheduler** is the default for general-purpose Linux systems.

- It aims to **distribute CPU time fairly** among processes.
- It uses a **red-black tree** to keep track of runnable processes.
- Processes with **less CPU time** get higher priority.

📌 **Key CFS Features**

Feature	Description
Dynamic Priorities	Adjusts process priority based on recent CPU usage.
Task Weighting	CPU-intensive tasks get lower priority.
Granularity Control	Ensures smooth multitasking.

📌 **Checking CFS Statistics**

```
cat /proc/sched_debug | grep -A 10 "cfs_rq"
```

11.2.2 Real-Time Scheduling (FIFO and RR)

Linux provides two **real-time scheduling policies**:

- **FIFO (First-In, First-Out)**: The highest-priority task runs **until it blocks or is preempted**.
- **RR (Round-Robin)**: Similar to FIFO, but each task gets a **fixed time slice** before the next task runs.

📌 **Changing Scheduling Policy**

```
chrt -f -p 99 <pid>   # Set FIFO priority 99 (highest)
chrt -r -p 50 <pid>   # Set RR priority 50
```

📌 **Example: Running a Process with Real-Time Priority**

```
sudo chrt -f 90 ./my_program  # Run with FIFO priority 90
```

11.3 Kernel Preemption and Task Priorities

11.3.1 What is Kernel Preemption?

Kernel preemption allows **higher-priority tasks** to interrupt lower-priority ones.

- Without preemption, a process **must voluntarily yield** control.
- Preemptive scheduling improves **responsiveness** for real-time applications.

📌 **Checking Kernel Preemption Mode**

```
zgrep CONFIG_PREEMPT /proc/config.gz
```

📌 **Kernel Preemption Options**

Mode	Description
CONFIG_PREEMPT_NONE	No preemption (good for servers).
CONFIG_PREEMPT_VOLUNTARY	Allows preemption in long-running kernel tasks.
CONFIG_PREEMPT	Full kernel preemption (good for desktops).

11.3.2 Task Priorities in Linux

Each process has a **priority value** that determines how often it runs.

📌 Checking Process Priorities

```
ps -eo pid,pri,ni,comm | sort -k2 -n  # List processes by priority
```

Value	Description
Static Priority	Used for real-time tasks (1-99).
Nice Value (-20 to +19)	Affects scheduling of non-real-time tasks.

📌 Changing Process Priority (Nice Value)

```
nice -n -10 ./my_program  # Increase priority
renice -n 5 -p <pid>  # Decrease priority
```

Conclusion

In this chapter, we covered: ✅ **How the Linux process scheduler works** ✅ **Different scheduling policies (CFS, FIFO, RR)** ✅ **Kernel preemption and task priorities**

Chapter 12: Kernel Synchronization Primitives

The Linux kernel is a **multitasking environment**, meaning multiple processes or threads can execute concurrently. **Kernel synchronization primitives** are essential to prevent **race conditions, deadlocks, and inconsistent data states** when accessing shared resources.

In this chapter, we will explore: ✅ **Spinlocks vs. Mutexes** – Choosing the right lock for synchronization ✅ **Read-Copy-Update (RCU)** – Optimizing read-heavy workloads ✅ **Semaphores, Atomic Operations, and Memory Barriers**

12.1 Spinlocks vs. Mutexes

12.1.1 Understanding Spinlocks

A **spinlock** is a lightweight lock where a thread **"spins" in a loop** until the lock is available. It is useful when:

- The critical section is **very short** (e.g., a few CPU cycles).
- The cost of putting a thread to sleep (context switch) is **higher than waiting**.

📌 **Spinlock Example**

```
#include <linux/spinlock.h>

static spinlock_t my_lock;

void critical_section(void) {
    spin_lock(&my_lock);  // Acquire lock
    // Critical section: modify shared data
    spin_unlock(&my_lock);  // Release lock
}

static int __init my_module_init(void) {
    spin_lock_init(&my_lock);  // Initialize spinlock
    return 0;
}

module_init(my_module_init);
MODULE_LICENSE("GPL");
```

📌 **Key Features of Spinlocks**

Feature	Description
Busy-waiting	Spins in a loop until the lock is free.
No sleeping	Cannot be used in code that might sleep (e.g., calling `schedule()`).
Efficient for short locks	Best when lock contention is low.

12.1.2 Understanding Mutexes

A **mutex (Mutual Exclusion)** is a **blocking lock** that **suspends** a process if the lock is already held.

- Used when the **critical section takes longer** to execute.
- The **kernel puts the waiting thread to sleep**, avoiding CPU wastage.

📌 **Mutex Example**

```
#include <linux/mutex.h>

static DEFINE_MUTEX(my_mutex);

void critical_section(void) {
    mutex_lock(&my_mutex);  // Acquire lock
    // Critical section
    mutex_unlock(&my_mutex);  // Release lock
}
```

📌 **Key Features of Mutexes**

Feature	Description
Blocking	If locked, the thread **sleeps** instead of busy-waiting.
Only for process context	Cannot be used in **interrupt handlers**.
Prevents starvation	FIFO-based scheduling of lock requests.

📌 **Choosing Between Spinlock and Mutex**

Condition	Use Spinlock	Use Mutex
Short critical section	✅	❌
Code can sleep	❌	✅
Interrupt handlers	✅	❌
High contention	❌	✅

12.2 Read-Copy-Update (RCU)

RCU is an **advanced synchronization mechanism** optimized for **read-heavy workloads**.

- **Readers do not block** writers or other readers.
- **Writers create a new copy** of the data structure, update it, and replace the old version.

📌 **Example: Using RCU**

```c
#include <linux/rcupdate.h>

struct my_data {
    int value;
    struct rcu_head rcu;
};

static struct my_data *global_ptr;

void reader_function(void) {
    struct my_data *ptr;
    rcu_read_lock();
    ptr = rcu_dereference(global_ptr);
    printk(KERN_INFO "Read value: %d\n", ptr->value);
    rcu_read_unlock();
}

void writer_function(void) {
    struct my_data *new_data = kmalloc(sizeof(*new_data), GFP_KERNEL);
    new_data->value = 42;
    rcu_assign_pointer(global_ptr, new_data);
    synchronize_rcu();  // Ensure previous readers are done
}
```

📌 RCU Advantages

Feature	Description
Fast reads	Readers do not use locks.
Efficient for multi-core systems	Minimizes contention.
Deferred updates	Writers operate on a new copy.

📌 When to Use RCU?

Condition	Use RCU
Many readers, few writers	✅
Need non-blocking reads	✅
Data structure updates are infrequent	✅

12.3 Semaphore, Atomic Operations, and Memory Barriers

12.3.1 Semaphores

A **semaphore** is used when multiple threads/processes need controlled access to a resource.

- **Mutexes** are a special case of semaphores (binary semaphore).
- **Semaphores** can allow **multiple** threads to access a resource at the same time.

📌 Example: Using Semaphores

```
#include <linux/semaphore.h>

static struct semaphore my_semaphore;

static int __init my_module_init(void) {
    sema_init(&my_semaphore, 2);  // Allow 2 concurrent accesses
    return 0;
}

void access_resource(void) {
    down(&my_semaphore);  // Acquire
    // Critical section
    up(&my_semaphore);  // Release
}
```

📌 Key Features of Semaphores

Feature	Description
Allows multiple accesses	Unlike mutexes, it allows more than one thread to proceed.
Blocking	If the semaphore count is **0**, the thread **sleeps** until available.

12.3.2 Atomic Operations

Atomic operations ensure that **certain instructions execute as an indivisible unit**, preventing race conditions.

📌 Example: Atomic Increment

```
#include <linux/atomic.h>

static atomic_t counter = ATOMIC_INIT(0);

void update_counter(void) {
    atomic_inc(&counter);  // Thread-safe increment
}
```

📌 Common Atomic Functions

Function	Description
`atomic_inc(&var)`	Increments `var` atomically.
`atomic_dec(&var)`	Decrements `var` atomically.
`atomic_add(n, &var)`	Adds `n` to `var`.

12.3.3 Memory Barriers

Memory barriers ensure that **memory operations occur in the correct order**.

- Required when working with **multi-core processors** where out-of-order execution happens.

📌 **Example: Using `smp_mb()` for Memory Barrier**

```c
#include <linux/smp.h>

int shared_var = 0;

void writer(void) {
    shared_var = 42;
    smp_mb();  // Ensure update is visible before proceeding
}

void reader(void) {
    smp_mb();  // Ensure memory consistency
    printk(KERN_INFO "Shared var: %d\n", shared_var);
}
```

📌 **Types of Memory Barriers**

Barrier	Description
`smp_mb()`	Full memory barrier (ensures all memory operations complete before proceeding).
`smp_rmb()`	Read memory barrier (ensures reads complete before proceeding).
`smp_wmb()`	Write memory barrier (ensures writes complete before proceeding).

Conclusion

In this chapter, we covered: ✅ **Spinlocks vs. Mutexes – Choosing the right lock** ✅ **Read-Copy-Update (RCU) – Optimized for multi-reader workloads** ✅ **Semaphores for resource control** ✅ **Atomic operations and memory barriers**

Chapter 13: Interrupts and Deferred Execution

Interrupts are essential for handling asynchronous events in the Linux kernel, such as hardware signals from devices (keyboards, network cards, timers, etc.). Since interrupt handlers must execute quickly, Linux provides **deferred execution mechanisms** like **SoftIRQs, Tasklets, and Workqueues** to process time-consuming tasks outside of interrupt context.

In this chapter, we will explore: ✅ **How the Linux kernel handles interrupts** ✅ **Writing and registering an interrupt handler** ✅ **Deferred execution mechanisms: Workqueues, Tasklets, and SoftIRQs**

13.1 Handling Interrupts in the Kernel

13.1.1 What is an Interrupt?

An **interrupt** is a signal sent by hardware or software to notify the CPU of an event.

📌 **Types of Interrupts**

Type	Description
Hardware Interrupts	Generated by devices (e.g., keyboard, network card).
Software Interrupts	Triggered by software using system calls (e.g., `kill -SIGINT`).

📌 **Interrupt Handling Process**

1. CPU receives an **interrupt signal**.
2. The **Interrupt Controller** determines the interrupt source.
3. The CPU **saves the current execution state** and jumps to the **Interrupt Service Routine (ISR)**.
4. The ISR **handles the interrupt** quickly and **returns control** to the previous task.

13.2 Writing an Interrupt Handler

13.2.1 Registering an Interrupt Handler

In Linux, an **interrupt handler** (ISR) is a kernel function that responds to an interrupt request (IRQ).

📌 **Steps to Register an Interrupt Handler**

1. Find the IRQ number for the device.
2. Register an ISR using `request_irq()`.
3. Implement the ISR function.
4. Unregister the ISR when the module is removed.

📌 **Example: Writing an Interrupt Handler**

```c
#include <linux/module.h>
#include <linux/interrupt.h>

#define IRQ_NUM 1  // Keyboard interrupt (example)

static irqreturn_t my_interrupt_handler(int irq, void *dev_id) {
    printk(KERN_INFO "Interrupt received: %d\n", irq);
    return IRQ_HANDLED;
}

static int __init my_module_init(void) {
    return request_irq(IRQ_NUM, my_interrupt_handler, IRQF_SHARED,
"my_irq_handler", (void *)my_interrupt_handler);
}

static void __exit my_module_exit(void) {
    free_irq(IRQ_NUM, (void *)my_interrupt_handler);
}

module_init(my_module_init);
module_exit(my_module_exit);
MODULE_LICENSE("GPL");
```

📌 **Key Functions**

Function	Description
`request_irq(irq, handler, flags, name, dev_id)`	Registers an interrupt handler.
`free_irq(irq, dev_id)`	Unregisters the interrupt handler.

📌 **Interrupt Flags**

Flag	Description
`IRQF_SHARED`	Allows sharing IRQs between multiple devices.
`IRQF_TRIGGER_RISING`	Triggers on a rising edge.
`IRQF_TRIGGER_FALLING`	Triggers on a falling edge.

13.3 Deferred Execution: Workqueues, Tasklets, and SoftIRQs

Since **interrupt handlers must execute quickly**, time-consuming tasks are **deferred** to be executed later in a safer context.

13.3.1 SoftIRQs

SoftIRQs are **lightweight kernel threads** used for handling high-priority work outside of interrupt context.

📌 **SoftIRQ Use Cases** ✅ Network packet processing ✅ Block device I/O scheduling

📌 **SoftIRQ Example**

```
#include <linux/interrupt.h>

static void my_softirq_function(struct softirq_action *action) {
    printk(KERN_INFO "SoftIRQ executed!\n");
}

static int __init my_module_init(void) {
    open_softirq(1, my_softirq_function);
    return 0;
}

module_init(my_module_init);
MODULE_LICENSE("GPL");
```

13.3.2 Tasklets

Tasklets are **simpler than SoftIRQs** and execute **in a single-threaded manner**.

📌 **Tasklet Features** ✅ Run at **lower priority** than SoftIRQs ✅ Only one instance of a tasklet runs at a time

📌 **Tasklet Example**

```
#include <linux/interrupt.h>

static void my_tasklet_function(unsigned long data);

DECLARE_TASKLET(my_tasklet, my_tasklet_function, 0);

static void my_tasklet_function(unsigned long data) {
    printk(KERN_INFO "Tasklet executed!\n");
}

static int __init my_module_init(void) {
    tasklet_schedule(&my_tasklet);  // Schedule the tasklet
    return 0;
}

module_init(my_module_init);
MODULE_LICENSE("GPL");
```

📌 **Tasklet API**

Function	Description
`DECLARE_TASKLET(name, function, data)`	Declare a tasklet.

Function	Description
`tasklet_schedule(&tasklet)`	Schedule a tasklet for execution.

13.3.3 Workqueues

Workqueues allow **deferred work** to be executed in **process context**. Unlike tasklets and SoftIRQs, **workqueues can sleep**, making them useful for tasks that require blocking operations (e.g., file I/O).

📌 Workqueue Example

```c
#include <linux/workqueue.h>

static struct workqueue_struct *my_wq;
static void my_work_function(struct work_struct *work);

static DECLARE_WORK(my_work, my_work_function);

static void my_work_function(struct work_struct *work) {
    printk(KERN_INFO "Workqueue task executed!\n");
}

static int __init my_module_init(void) {
    my_wq = create_singlethread_workqueue("my_queue");
    queue_work(my_wq, &my_work);  // Schedule work
    return 0;
}

static void __exit my_module_exit(void) {
    flush_workqueue(my_wq);
    destroy_workqueue(my_wq);
}

module_init(my_module_init);
module_exit(my_module_exit);
MODULE_LICENSE("GPL");
```

📌 Workqueue API

Function	Description
`queue_work(workqueue, &work)`	Adds a work item to a workqueue.
`flush_workqueue(workqueue)`	Waits for all work to finish.
`destroy_workqueue(workqueue)`	Destroys a workqueue.

📌 Choosing Between SoftIRQs, Tasklets, and Workqueues

Feature	SoftIRQ	Tasklet	Workqueue

Feature	SoftIRQ	Tasklet	Workqueue
Runs in Interrupt Context?	✅ Yes	✅ Yes	❌ No
Can Sleep?	❌ No	❌ No	✅ Yes
Priority	High	Medium	Low
Use Case	Network stack, block I/O	Lightweight tasks	Long-running tasks

Conclusion

In this chapter, we covered: ✅ **How Linux handles interrupts** ✅ **Writing an interrupt handler** ✅ **Deferred execution: SoftIRQs, Tasklets, and Workqueues**

Chapter 14: Debugging the Linux Kernel

Debugging the Linux kernel is **challenging** because traditional debugging tools like `printf` and GDB are often **not directly usable** in the kernel space. However, Linux provides powerful debugging tools such as **`dmesg`, kgdb, Ftrace, and Kprobes** to diagnose issues efficiently.

In this chapter, we will explore: ✅ **Kernel Logs and `dmesg`** – Checking logs for debugging information. ✅ **Using `gdb` with the Kernel (`kgdb`)** – Debugging the kernel step by step. ✅ **Ftrace and Kprobes** – Advanced tracing and function hooking. ✅ **Live Patching and Kernel Crash Analysis** – Handling kernel crashes without rebooting.

14.1 Kernel Logs and `dmesg`

14.1.1 Understanding Kernel Logs

The Linux kernel continuously logs important system messages, such as **boot messages, driver errors, and crash reports**. These logs can be accessed using `dmesg` or `journalctl`.

📌 **Checking Kernel Logs with `dmesg`**

```
dmesg | less
```

🔹 **Filters logs for specific messages**

```
dmesg | grep "error"
```

🔹 **Continuous kernel log monitoring**

```
dmesg -w
```

14.1.2 Logging from the Kernel

📌 **Using `printk()` for Debugging in the Kernel**

```c
#include <linux/kernel.h>
#include <linux/module.h>

static int __init my_module_init(void) {
    printk(KERN_INFO "Hello, Kernel! Module loaded.\n");
    return 0;
}
```

```
static void __exit my_module_exit(void) {
    printk(KERN_INFO "Goodbye, Kernel! Module removed.\n");
}

module_init(my_module_init);
module_exit(my_module_exit);
MODULE_LICENSE("GPL");
```

📌 **Kernel Log Levels**

Log Level	Macro	Description
0	`KERN_EMERG`	System is unusable
1	`KERN_ALERT`	Immediate action needed
2	`KERN_CRIT`	Critical conditions
3	`KERN_ERR`	Error conditions
4	`KERN_WARNING`	Warning conditions
5	`KERN_NOTICE`	Normal but significant
6	`KERN_INFO`	Informational messages
7	`KERN_DEBUG`	Debugging messages

14.2 Using `gdb` with the Kernel (`kgdb`)

14.2.1 What is `kgdb`?

`kgdb` is a **remote kernel debugger** that allows you to debug the kernel using `gdb`. You can: ✅ Set breakpoints in the kernel ✅ Step through kernel code ✅ Examine memory and variables

14.2.2 Setting Up `kgdb`

📌 **1. Enable `kgdb` in the Kernel Configuration**

- Enable **CONFIG_KGDB** and **CONFIG_KGDB_SERIAL_CONSOLE** in your kernel `.config`:

```
make menuconfig
```

Navigate to: `Kernel hacking → Kernel debugging → KGDB: kernel debugger`

📌 **2. Set Kernel Boot Parameters** Add these boot parameters to the kernel command line:

```
kgdboc=ttyS0,115200 nokaslr
```

📌 **3. Start Debugging with `gdb`** On the host machine, run:

```
gdb vmlinux
target remote /dev/ttyS0
```

📌 **Basic `gdb` Commands for Kernel Debugging**

Command	Description
`b function_name`	Set a breakpoint at `function_name`
`c`	Continue execution
`s`	Step into a function
`n`	Step over a function
`p var`	Print the value of `var`
`bt`	Print stack trace

14.3 Debugging with Ftrace and Kprobes

14.3.1 Using Ftrace for Function Tracing

Ftrace is a **powerful built-in tracing tool** for monitoring function execution.

📌 **Enable Ftrace in the Kernel** Make sure `CONFIG_FTRACE` is enabled:

```
cat /boot/config-$(uname -r) | grep FTRACE
```

📌 **Start Tracing Function Calls**

```
echo function > /sys/kernel/debug/tracing/current_tracer
echo 1 > /sys/kernel/debug/tracing/tracing_on
cat /sys/kernel/debug/tracing/trace
```

📌 **Filter a Specific Function**

```
echo do_sys_open > /sys/kernel/debug/tracing/set_ftrace_filter
```

14.3.2 Using Kprobes for Kernel Function Hooking

Kprobes allow you to **dynamically insert breakpoints** into running kernel code.

📌 **Kprobes Example: Hooking `sys_open`**

```
#include <linux/kprobes.h>

static struct kprobe kp;

static int handler_pre(struct kprobe *p, struct pt_regs *regs) {
    printk(KERN_INFO "sys_open called: %s\n", (char *)regs->si);
    return 0;
}

static int __init my_init(void) {
    kp.symbol_name = "sys_open";
    register_kprobe(&kp);
    return 0;
}

static void __exit my_exit(void) {
    unregister_kprobe(&kp);
}

module_init(my_init);
module_exit(my_exit);
MODULE_LICENSE("GPL");
```

14.4 Live Patching and Kernel Crash Analysis

14.4.1 Live Patching with `kpatch`

Linux supports **live kernel patching** to fix security issues **without rebooting**.

📌 **Installing `kpatch`**

```
sudo apt install kpatch
```

📌 **Apply a Live Patch**

```
kpatch load my_patch.ko
```

14.4.2 Analyzing Kernel Crashes with `crash`

When a kernel **panic** occurs, use the `crash` tool to analyze the core dump.

📌 **Install and Run `crash`**

```
sudo apt install crash linux-crashdump
crash /usr/lib/debug/boot/vmlinux /var/crash/core
```

📌 Basic Crash Commands

Command	Description
`bt`	Show stack trace
`ps`	Show process list
`vm`	Show virtual memory info

Conclusion

In this chapter, we covered: ✅ **Kernel Logs and `dmesg` for debugging** ✅ **Using `kgdb` for step-by-step kernel debugging** ✅ **Advanced tracing with Ftrace and Kprobes** ✅ **Live kernel patching and crash analysis**

Chapter 15: Profiling and Performance Tuning

Performance optimization is critical in the Linux kernel, especially for **high-performance computing, embedded systems, and real-time applications**. This chapter will cover **profiling tools and techniques** to analyze and optimize kernel performance.

In this chapter, we will explore: ✅ **Using `perf` for Performance Analysis** – Profiling CPU, memory, and function execution. ✅ **BPF and eBPF for Tracing and Profiling** – Efficient, low-overhead kernel tracing. ✅ **Optimizing Kernel Performance for Embedded Systems** – Reducing latency and improving efficiency.

15.1 Using `perf` for Performance Analysis

15.1.1 Introduction to `perf`

`perf` is a **powerful Linux performance analysis tool** that provides insights into: ✅ **CPU usage** (which functions consume the most CPU time) ✅ **Memory access patterns** (cache misses, page faults) ✅ **Kernel bottlenecks** (lock contention, scheduler delays)

📌 **Install `perf` (if not installed)**

```
sudo apt install linux-perf
```

15.1.2 Profiling Kernel Performance

📌 **Measure CPU Usage for the Kernel**

```
sudo perf top
```

🔹 This continuously shows the hottest kernel functions in real time.

📌 **Record CPU Events and Analyze**

```
sudo perf record -g -a sleep 10
sudo perf report
```

🔹 This captures **call graphs** to analyze function execution time.

📌 **Trace Kernel Function Execution**

```
sudo perf probe -a do_sys_open
sudo perf record -e probe:do_sys_open -a
sudo perf report
```

◈ This traces how many times `do_sys_open` is called and how long it takes.

📌 **Measure Context Switches**

```
sudo perf sched record
sudo perf sched latency
```

◈ Helps detect **high scheduling latency** issues.

15.2 BPF and eBPF for Tracing and Profiling

15.2.1 What is eBPF?

Extended Berkeley Packet Filter (eBPF) is a **high-performance in-kernel virtual machine** that allows: ✅ **Efficient event tracing** (without modifying kernel code) ✅ **Low-overhead monitoring** (ideal for production systems) ✅ **Running safe, verified programs inside the kernel**

📌 **Check if eBPF is enabled**

```
ls /sys/fs/bpf
```

15.2.2 Using `bpftrace` for Kernel Tracing

📌 **Install `bpftrace`**

```
sudo apt install bpftrace
```

📌 **List Available Kernel Functions**

```
bpftrace -l 'kprobe:*'
```

📌 **Trace a Kernel Function (`do_sys_open`)**

```
sudo bpftrace -e 'kprobe:do_sys_open { printf("File opened: %s\n", str(arg1)); }'
```

📌 **Monitor System Calls (`execve`)**

```
sudo bpftrace -e 'tracepoint:syscalls:sys_enter_execve { printf("%s executed
%s\n", comm, str(args->filename)); }'
```

15.2.3 Writing a Simple eBPF Program

📌 **eBPF Program to Count `do_sys_open` Calls**

```
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>

BPF_HASH(counter);

int count_open(struct pt_regs *ctx) {
    u64 pid = bpf_get_current_pid_tgid();
    counter.increment(pid);
    return 0;
}
```

🔷 This eBPF program **counts how many times each process calls** `open()`.

15.3 Optimizing Kernel Performance for Embedded Systems

15.3.1 Reducing Kernel Size

📌 **Disable Unused Features in** `menuconfig`

```
make menuconfig
```

Disable: ✅ **Unused device drivers** ✅ **Debugging options (e.g.,** `CONFIG_DEBUG_INFO`**)** ✅ **Legacy compatibility options**

📌 **Compile a Minimal Kernel**

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bzImage
```

15.3.2 Reducing Boot Time

📌 **Enable Fast Boot Options**

- Enable `CONFIG_INITRAMFS_COMPRESSION_NONE`
- Disable `CONFIG_PRINTK` for faster boot logs

📌 **Measure Boot Performance**

```
dmesg | grep -i boot
systemd-analyze blame
```

15.3.3 Optimizing Scheduler for Real-Time Performance

📌 **Use `SCHED_FIFO` or `SCHED_RR` for Real-Time Tasks**

```
#include <sched.h>

struct sched_param param;
param.sched_priority = 50;
sched_setscheduler(0, SCHED_FIFO, &param);
```

📌 **Check Kernel Preemption Settings**

```
zgrep CONFIG_PREEMPT /proc/config.gz
```

◈ **Enable `CONFIG_PREEMPT_RT` for real-time systems.**

Conclusion

In this chapter, we covered: ✅ **Using `perf` for CPU profiling** ✅ **eBPF for advanced kernel tracing** ✅ **Optimizing the kernel for embedded systems**

Chapter 16: Linux Boot Process and Optimization

The **Linux boot process** is critical for system startup, and **optimizing boot time** is crucial for embedded systems, real-time applications, and performance-critical environments. This chapter will cover:

✅ **Understanding the Boot Sequence** – How the Linux kernel initializes. ✅ **Analyzing Boot Time with `systemd-analyze`** – Measuring and identifying slow boot components. ✅ **Reducing Boot Time for Embedded Systems** – Optimizing the kernel and userspace startup.

16.1 Understanding the Boot Sequence

The Linux boot process involves multiple stages, each responsible for initializing different components of the system.

16.1.1 Stages of the Linux Boot Process

📌 **1. Firmware Initialization (BIOS/UEFI)**

- The **BIOS or UEFI firmware** initializes hardware (CPU, memory, storage).
- It locates the **bootloader** and transfers control to it.

📌 **2. Bootloader (GRUB, U-Boot, systemd-boot)**

- The **bootloader loads the Linux kernel** into memory.
- It may also load an **initial RAM disk (initramfs)** for early userspace tasks.

📌 **3. Kernel Initialization**

- The kernel: ✅ Sets up memory management. ✅ Detects and initializes hardware. ✅ Mounts the root filesystem.

📌 **4. Init System (`systemd`, `SysVinit`, or `OpenRC`)**

- `systemd` (on most modern Linux systems) initializes **user-space services** and targets.

📌 **5. Userspace Startup**

- System services, daemons, and the login prompt/graphical environment start.

16.2 Analyzing Boot Time with `systemd-analyze`

📌 **Check Total Boot Time**

```
systemd-analyze
```

🔷 Example Output:

```
Startup finished in 3.456s (kernel) + 7.892s (userspace) = 11.348s
```

- **Kernel time**: Time taken to initialize the kernel.
- **Userspace time**: Time taken for system services and login screen.

📌 **Identify Slowest Services**

```
systemd-analyze blame
```

🔹 Example Output:

```
5.890s NetworkManager.service
3.012s docker.service
2.458s bluetooth.service
```

🔹 This helps identify **slow-starting services** that can be optimized.

📌 **Visualizing Boot Order**

```
systemd-analyze critical-chain
```

🔹 This command shows dependencies that **delay the boot process**.

16.3 Reducing Boot Time for Embedded Systems

16.3.1 Optimizing Kernel Boot Time

📌 **1. Reduce Kernel Size**

- Disable unused drivers and features using:

```
make menuconfig
```

- Disable: ✅ Unused file systems ✅ Debugging symbols (`CONFIG_DEBUG_INFO`) ✅ Legacy hardware support

📌 **2. Use `initramfs` Minimally**

- Reduce `initramfs` size by removing unnecessary modules:

```
dracut --force --omit-drivers "unused_driver"
```

📌 **3. Use `quiet` and `loglevel=3` Boot Parameters**

- This **reduces boot log verbosity** and speeds up boot.
- Edit `/etc/default/grub`:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet loglevel=3"
```

- Update GRUB:

```
sudo update-grub
```

16.3.2 Optimizing Userspace Startup

📌 **1. Disable Unnecessary Services**

```
systemctl disable bluetooth.service
systemctl disable NetworkManager-wait-online.service
```

🔷 This removes services **not needed at boot**.

📌 **2. Use Parallel Service Startup**

- Enable `systemd` parallel startup:

```
systemctl set-default multi-user.target
```

- This **skips GUI startup** for faster boot on servers.

📌 **3. Optimize Systemd Units**

- Edit slow services to **start later (lazy loading)**. Example:

```
systemctl edit docker.service
```

- Add:

```
[Service]
ExecStartPre=/bin/sleep 10
```

🔷 This delays `docker.service` by 10 seconds.

Conclusion

In this chapter, we covered: ✅ **Linux boot stages and system initialization** ✅ **Using `systemd-analyze` to profile boot time** ✅ **Optimizing boot performance for embedded systems**

Chapter 17: Writing Real-Time Linux Applications

Real-time Linux applications require **predictable response times**, which is essential for **industrial automation, robotics, medical systems, and telecommunications**.

In this chapter, we will explore: ✅ **Introduction to Real-Time Linux** – Soft vs. Hard real-time requirements. ✅ **RT Preemption and Scheduling** – Understanding `PREEMPT_RT` and real-time schedulers. ✅ **Tuning Linux for Low-Latency Applications** – Kernel and system optimizations.

17.1 Introduction to Real-Time Linux

17.1.1 What is a Real-Time System?

A real-time system must respond to an **event within a specific deadline**.

📌 **Types of Real-Time Systems:**

- **Hard Real-Time: Missing a deadline causes system failure** (e.g., pacemakers, automotive ABS).
- **Soft Real-Time: Missing a deadline degrades performance** (e.g., video streaming, audio processing).

📌 **Standard Linux vs. Real-Time Linux:**

Feature	Standard Linux	Real-Time Linux
Scheduling	Best-effort (CFS)	Deterministic (FIFO, RR)
Latency	Variable	Bounded
Interrupt Handling	Deferred execution	Low-latency response

17.2 RT Preemption and Scheduling

17.2.1 The `PREEMPT_RT` Patch

🔶 The `PREEMPT_RT` **patch** transforms Linux into a **fully preemptible kernel** for **low-latency tasks**.

📌 **Check if the Kernel Supports RT Preemption:**

```
zgrep CONFIG_PREEMPT /proc/config.gz
```

🔶 If `CONFIG_PREEMPT_RT` is enabled, the system supports real-time preemption.

📌 **Installing an RT Kernel:**

```
sudo apt install linux-image-rt
```

🔷 This installs the **real-time (RT) patched kernel** for low-latency scheduling.

17.2.2 Real-Time Scheduling Policies

Linux provides **three real-time scheduling policies**:

Policy	Description	Use Case
SCHED_FIFO	First-In, First-Out (runs until blocked)	Hard real-time
SCHED_RR	Round-Robin with time slices	Soft real-time
SCHED_DEADLINE	Tasks execute within a deadline	High-precision workloads

📌 **Setting a Real-Time Process Priority (`SCHED_FIFO`)**

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

int main() {
    struct sched_param param;
    param.sched_priority = 50;  // Priority (1-99)

    if (sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
        perror("sched_setscheduler failed");
    } else {
        printf("Process running with SCHED_FIFO priority 50\n");
        while (1);  // Infinite loop to simulate real-time task
    }
    return 0;
}
```

🔷 This sets the process to **high-priority real-time execution**.

📌 **Check Running Real-Time Processes:**

```
ps -eo pid,comm,policy,rtprio
```

🔷 This shows the **real-time priority** (`rtprio`) of running tasks.

17.3 Tuning Linux for Low-Latency Applications

17.3.1 Reducing Kernel Latency

📌 **Disable Power-Saving Features**

```
echo 0 | sudo tee /sys/devices/system/cpu/cpu*/cpuidle/state*/disable
```

🔷 This prevents CPU frequency scaling, which introduces jitter.

📌 **Disable Timer Tick (Tickless Kernel Mode)** Edit `/etc/default/grub`:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet nohz_full=1-3"
```

🔷 This reduces CPU interruptions for dedicated real-time cores.

17.3.2 Prioritizing Real-Time Processes

📌 **Lock a Process in Memory (Avoid Swapping)**

```
sudo chrt -f 99 ./real_time_app
sudo ionice -c 1 -n 0 ./real_time_app
```

🔷 This prevents real-time tasks from being delayed by **disk I/O or page swapping**.

📌 **Isolate CPUs for Real-Time Tasks** Edit `/etc/default/grub`:

```
GRUB_CMDLINE_LINUX_DEFAULT="isolcpus=1,2 nohz_full=1,2 rcu_nocbs=1,2"
```

🔷 This reserves **CPUs 1 and 2** exclusively for real-time tasks.

Conclusion

In this chapter, we covered: ✅ **Real-time Linux concepts and scheduling policies** ✅ **The `PREEMPT_RT` patch for low-latency processing** ✅ **Optimizing Linux for real-time applications**

Chapter 18: Security in the Linux Kernel

Security is a critical aspect of **Linux kernel development**, as the kernel controls **process execution, memory access, and hardware interaction**.

In this chapter, we will explore: ✅ **Kernel Security Mechanisms** – Mandatory Access Control, seccomp, and namespaces. ✅ **Securing Kernel Modules** – Preventing unauthorized module loading and execution. ✅ **Implementing Secure Boot and Signed Modules** – Protecting the kernel from tampering.

18.1 Kernel Security Mechanisms

The Linux kernel has **built-in security features** that protect against exploits and unauthorized access.

18.1.1 Mandatory Access Control (MAC) - SELinux & AppArmor

📌 **Check if SELinux is enabled:**

```
getenforce
```

🔹 If the output is **"Enforcing"**, SELinux is active.

📌 **Set SELinux to Enforcing Mode:**

```
sudo setenforce 1
```

🔹 This forces strict access control for applications.

📌 **Using AppArmor for Security Policies**

```
sudo aa-status
```

🔹 This checks which processes are restricted by AppArmor profiles.

18.1.2 Seccomp - System Call Filtering

Seccomp (Secure Computing Mode) restricts system calls available to a process, reducing attack surface.

📌 **Enable Seccomp Filtering in C:**

```
#include <linux/seccomp.h>
#include <sys/prctl.h>
```

```
int main() {
    prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);
    return 0;
}
```

🔹 This **blocks system calls** except read(), write(), `_exit()`, and sigreturn().

18.1.3 Kernel Namespaces - Process Isolation

Namespaces isolate resources between processes, providing enhanced security.

📌 **Check Existing Namespaces:**

```
lsns
```

🔹 This lists active namespaces, including **PID, network, and user namespaces**.

📌 **Create an Isolated Namespace:**

```
unshare --user --map-root-user bash
```

🔹 This starts a shell with a **new user namespace**, preventing access to system-wide privileges.

18.2 Securing Kernel Modules

18.2.1 Restricting Kernel Module Loading

By default, users with root privileges can **load and unload kernel modules**, posing a security risk.

📌 **Prevent Unauthorized Module Loading:**

```
echo 1 > /proc/sys/kernel/modules_disabled
```

🔹 This **disables dynamic module loading** after boot.

📌 **Restrict Module Loading to Signed Modules Only:**

```
echo 1 > /proc/sys/kernel/module_sig_enforce
```

🔹 This ensures only **cryptographically signed modules** can be loaded.

18.2.2 Writing a Secure Kernel Module

A **malicious kernel module** can compromise system integrity. Secure module development practices include: ✅ **Minimal privileged operations** ✅ **Restricting module parameters** ✅ **Using memory-safe functions (`strncpy` instead of `strcpy`)**

📌 **Secure Kernel Module Example (Prevents Unauthorized Unload)**

```
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Secure Dev");
MODULE_DESCRIPTION("A Secure Kernel Module");
MODULE_VERSION("1.0");

static int __init secure_init(void) {
    printk(KERN_INFO "Secure module loaded\n");
    return 0;
}

static void __exit secure_exit(void) {
    printk(KERN_INFO "Secure module unloaded\n");
}

module_init(secure_init);
module_exit(secure_exit);

/* Prevent removal */
MODULE_INFO(intree, "Y");
```

🔷 The `MODULE_INFO(intree, "Y");` prevents **forced removal** using `rmmod`.

18.3 Implementing Secure Boot and Signed Modules

18.3.1 What is Secure Boot?

Secure Boot ensures that only **trusted, signed kernel images** are loaded at boot time.

📌 **Check Secure Boot Status:**

```
mokutil --sb-state
```

🔷 If enabled, the system **only allows signed kernels and bootloaders**.

18.3.2 Signing Kernel Modules

📌 **Generate a Signing Key:**

```
openssl req -new -x509 -keyout MOK.key -out MOK.crt -nodes -days 365
```

📌 **Sign a Kernel Module (`my_module.ko`)**

```
sudo /usr/src/linux-headers-$(uname -r)/scripts/sign-file sha256 MOK.key MOK.crt
my_module.ko
```

📌 **Verify the Module Signature:**

```
modinfo my_module.ko | grep sig
```

🔹 If signed, it will show `signature` details.

📌 **Load the Signed Module:**

```
sudo insmod my_module.ko
```

🔹 **Unsigned modules will be rejected** if Secure Boot is active.

Conclusion

In this chapter, we covered: ✅ **Kernel security mechanisms (SELinux, seccomp, namespaces)** ✅ **Restricting kernel module loading to prevent exploits** ✅ **Implementing Secure Boot and signed modules**

Chapter 19: Contributing to the Linux Kernel

Contributing to the Linux kernel is a great way to **improve system stability, enhance features, and gain expertise in low-level development**. However, the kernel community has strict **coding guidelines, patch submission protocols, and review processes** that contributors must follow.

This chapter will cover: ✅ **Understanding the Kernel Development Workflow** – How kernel changes are made and merged. ✅ **Writing Patches and Submitting to LKML** – How to format and send patches. ✅ **Best Practices for Kernel Coding** – Ensuring high-quality contributions.

19.1 Understanding the Kernel Development Workflow

The **Linux kernel is developed collaboratively** by thousands of developers worldwide. Contributions go through **review and testing** before being merged into the mainline kernel.

19.1.1 The Linux Kernel Development Model

📌 **Key Components in Kernel Development:**

- **Linus Torvalds** – Maintains the mainline kernel (`torvalds/linux`).
- **Subsystem Maintainers** – Handle different kernel components (e.g., memory management, networking).
- **Developers & Contributors** – Write patches and submit them for review.

19.1.2 Kernel Release Cycle

📌 **How Often is the Kernel Released?**

- **New kernel versions** are released every **8-10 weeks**.
- **Long-Term Support (LTS) versions** receive fixes for **several years**.

📌 **Check the Latest Kernel Version:**

```
wget -qO- https://www.kernel.org/ | grep "latest_stable"
```

19.1.3 Kernel Source Code Structure

📌 **Cloning the Kernel Source Code:**

```
git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
cd linux
```

📌 **Important Directories in the Kernel Source:**

Directory	Purpose
`arch/`	Architecture-specific code (x86, ARM, RISC-V)
`drivers/`	Device drivers (GPU, networking, storage)
`fs/`	Filesystem implementations (ext4, XFS, Btrfs)
`include/`	Kernel headers for internal APIs
`kernel/`	Core kernel functions (scheduling, locking)
`mm/`	Memory management subsystem
`net/`	Networking stack
`security/`	Security features (SELinux, AppArmor)

19.2 Writing Patches and Submitting to LKML

19.2.1 Writing a Kernel Patch

Linux kernel patches must follow a strict format.

📌 **Example: Fixing a Typo in a Kernel Comment**

```
nano kernel/sched/core.c
```

Change:

```
/* This is the schduler */
```

To:

```
/* This is the scheduler */
```

📌 **Generating a Patch:**

```
git add kernel/sched/core.c
git commit -s -m "sched: Fix typo in scheduler comment"
git format-patch -1
```

🔷 The `-s` flag **adds a Signed-off-by line**, required for all patches.

19.2.2 Submitting Patches to LKML

📌 **Find the Right Maintainer for Your Code:**

```
scripts/get_maintainer.pl -f kernel/sched/core.c
```

◈ This prints **email addresses of maintainers** responsible for the affected file.

📌 **Send the Patch via Email (Using `git send-email`)**

```
git send-email --to="maintainer@example.com" 0001-sched-Fix-typo-in-scheduler-comment.patch
```

◈ **All patches must be submitted to the Linux Kernel Mailing List (LKML)** for review.

📌 **Subscribe to LKML to Track Discussions:**

```
echo "subscribe linux-kernel" | mail -s subscribe majordomo@vger.kernel.org
```

19.3 Best Practices for Kernel Coding

19.3.1 Follow the Linux Kernel Coding Style

📌 **Check Code Formatting:**

```
scripts/checkpatch.pl --strict 0001-sched-Fix-typo-in-scheduler-comment.patch
```

◈ This tool **flags formatting issues** (e.g., missing spaces, incorrect indentation).

📌 **Kernel Code Formatting Example (Good vs. Bad Code):** ✅ **Good (Proper Indentation & Spacing):**

```
if (condition) {
    do_something();
} else {
    do_something_else();
}
```

❌ **Bad (Misaligned Braces & No Space):**

```
if(condition){
do_something();}
else{
do_something_else();}
```

📌 **Use `clang-format` for Automatic Formatting:**

```
clang-format -i my_driver.c
```

19.3.2 Documenting Your Code

📌 **Adding Kernel Documentation Comments:**

```c
/**
 * my_function - Short description
 * @arg1: Description of the argument
 * @arg2: Description of the argument
 *
 * This function does XYZ.
 */
void my_function(int arg1, int arg2) { ... }
```

🔷 Well-documented functions make it easier for **maintainers to review** patches.

19.3.3 Testing Your Kernel Code

📌 **Compile and Boot a New Kernel:**

```
make -j$(nproc)
sudo make modules_install
sudo make install
sudo reboot
```

🔷 Always **test patches before submission** to avoid breaking system functionality.

📌 **Use `kunit` for Kernel Unit Testing:**

```
make kunitconfig
./tools/testing/kunit/kunit.py run
```

Conclusion

In this chapter, we covered: ✅ **The Linux kernel development workflow** ✅ **How to write and submit patches to LKML** ✅ **Best practices for kernel coding and testing**

Chapter 20: Final Thoughts and Further Learning

As we conclude this book on **Linux Kernel Programming**, this final chapter provides: ✅ **Recommended Books and Resources** – Further reading to deepen your understanding. ✅ **Future of Linux Kernel Development** – Trends shaping the evolution of the Linux kernel.

20.1 Recommended Books and Resources

20.1.1 Books on Linux Kernel Development

📌 **Must-read books for Linux kernel programmers:**

Book	Author	Description
Linux Kernel Development (3rd Edition)	Robert Love	Beginner-friendly introduction to kernel internals.
Linux Device Drivers (3rd Edition)	Jonathan Corbet, Alessandro Rubini	Best guide for writing kernel drivers.
Understanding the Linux Kernel	Daniel P. Bovet, Marco Cesati	Deep dive into kernel mechanisms.
Linux System Programming	Robert Love	Covers system calls and low-level programming.

20.1.2 Online Documentation & Resources

📌 **Official Linux Kernel Documentation:**

- 📖 **Kernel.org Documentation** – The official kernel reference.
- 📖 **Linux Device Drivers Documentation**

📌 **Mailing Lists & Forums:**

- 📩 **Linux Kernel Mailing List (LKML)** – Where kernel discussions and patches happen.
- 🛠 **Linux Kernel Newbies** – Great for beginner contributors.

📌 **Interactive Learning & Tools:**

- 🏛 **LXR (Linux Cross Reference)** – Browse and search kernel source code.
- 🏗 **Linux From Scratch** – Learn by building Linux from source.

20.2 Future of Linux Kernel Development

The Linux kernel is constantly evolving to support **new hardware, security improvements, and performance optimizations**.

20.2.1 Key Trends in Kernel Development

🔹 Real-Time Linux (RT)

- The **PREEMPT_RT** patchset is being merged into mainline.
- This enables **low-latency, real-time performance** for industrial and automotive applications.

🔹 BPF and eBPF: The Future of Linux Observability

- **eBPF (Extended Berkeley Packet Filter)** is revolutionizing **tracing, security, and networking** in the kernel.
- Tools like **BPFtrace** and **XDP** allow efficient kernel monitoring without modifying code.

🔹 Security & Confidential Computing

- Features like **KPTI (Kernel Page Table Isolation)** and **Spectre/Meltdown mitigations** enhance security.
- **Confidential VMs** (e.g., AMD SEV, Intel TDX) protect kernel workloads in cloud environments.

🔹 RISC-V & New Hardware Architectures

- Linux is expanding support for **RISC-V**, a rapidly growing open-source CPU architecture.
- More **power-efficient and AI-optimized** hardware integration is expected.

20.3 Final Words

✅ This book covered the **fundamentals and advanced topics** of Linux kernel programming. ✅ You now have the knowledge to **develop kernel modules, write drivers, debug issues, and contribute to the Linux kernel**. ✅ The best way to learn further is by **experimenting with real-world kernel development and participating in the community**.