Unit 7: Linux Case Study 5Hrs

7.1 Linux Overview

History of Linux

- **1991**: Linus Torvalds developed Linux as a free and open-source Unix-like kernel.
- **GNU(GNU's Not Unix) Project**: Combined with GNU utilities to form a complete OS (GNU/Linux).
- **Distributions**: Variants like Ubuntu, Fedora, Debian, and CentOS package the kernel with software.
- *Unix → GNU (tools) → Linux (kernel + GNU tools = complete OS)**

Key Milestones:

- **1992**: Linux licensed under GNU GPL (General Public License).
- **1996**: Adoption in servers and embedded systems.
- **2000s**: Dominance in servers, cloud computing, and Android (Linux-based).

7.2 Kernel Modules

Definition

- Loadable kernel modules (LKMs) extend kernel functionality without rebooting.
- Examples: Device drivers, filesystem support.

If there were **no Loadable Kernel Modules (LKM)** in Linux, then:

1. **All drivers would have to be built directly into the kernel (monolithic build).**
    - The kernel would become **large and bloated** because it must include support for every possible device.
    - Most of that code would be unused most of the time.
2. **No dynamic hardware support.**
    - If you connect a new keyboard, network card, or USB device, **the system wouldn't recognize it until you reboot** with a kernel that already has the driver compiled in.
3. **Harder maintenance and updates.**
    - To add or update a driver, you would need to **recompile and reinstall the whole kernel**, then reboot.
    - This would be slow and inconvenient, especially for servers.
4. **Less flexibility.**
    - Modern systems often load and unload modules on the fly (e.g., hotplug USB devices). Without LKMs, this dynamic capability is gone.

Advantages

 Modularity – Add/remove features dynamically.
 Reduced memory usage – Load only required modules.
 Easier development – Test modules without full kernel recompilation.

Disadvantages

 Stability risk – Poorly written modules can crash the kernel.
 Security risk – Malicious modules can exploit the system.

Commands for Module Management

Command	Description
`lsmod`	Lists loaded modules
`insmod`	Inserts a module
`rmmod`	Removes a module
`modinfo`	Displays module info

Example:

```
# Insert a module
sudo insmod example.ko

# Remove a module
sudo rmmod example
```

7.3 Process Management

Process vs. Thread

Feature	Process	Thread
Isolation	High (separate memory)	Low (shared memory)
Creation overhead	High	Low
Communication	IPC (slow)	Shared memory (fast)

Process States

1. Running (R)

- Process is either currently running on the CPU or ready to run (waiting in the run queue).
- Example: A process actively using CPU or waiting for CPU time.

2. Interruptible Sleep (S)

- Process is sleeping and **can be interrupted** by signals (e.g., waiting for user input, disk I/O).
- This is the most common sleep state.

3. Uninterruptible Sleep (D)

- Process is sleeping but **cannot be interrupted** (usually waiting on I/O like disk access).
- Important because such processes cannot be killed easily until they finish the I/O.

4. Stopped (T)

- Process has been stopped, usually by a user signal (`SIGSTOP`) or during debugging.
- Can be resumed later.

5. Zombie (Z)

- Process has finished execution but **still has an entry in the process table** because its parent has not read its exit status (using `wait()`).
- It's basically a "dead" process waiting for cleanup.

Process State Transition Diagram

System Calls for Process Management

- `fork()` – Creates a child process (copy of parent).
- `exec()` – Replaces process memory with a new program.
- `wait()` – Parent waits for child to terminate.
- `exit()` – Terminates a process.

Example:

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

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child Process\n");
    } else {
        printf("Parent Process\n");
    }
    return 0;
}
```

7.4 Scheduling

Linux Schedulers

1. **Completely Fair Scheduler (CFS)** (Default)
    - Uses a red-black tree to track process runtime.
    - Ensures fair CPU time allocation.
2. **Real-Time Scheduler**
    - For time-critical tasks (FIFO, Round Robin).

Comparison of Schedulers

Scheduler	Policy	Best For
CFS	Fair time-slicing	General-purpose systems
FIFO	First-In-First-Out	Real-time tasks
Round Robin	Time-sliced FIFO	Interactive tasks

7.5 Inter-Process Communication (IPC)

What is IPC?

- IPC allows **processes to exchange data and synchronize actions**.
- Essential because processes run independently but often need to collaborate.

Why IPC is Needed?

- Processes have separate address spaces; they can’t directly access each other’s memory.
- IPC provides methods to share data and signals safely and efficiently.

Common IPC Mechanisms in Linux

IPC Mechanism	Description	Use Case Example
Pipes	Unidirectional data channel between related processes (parent-child).	Sending output of one process as input to another
Named Pipes (FIFOs)	Like pipes but with a name in the filesystem; can be used between unrelated processes.	Communication between unrelated processes.
Message Queues	Allows processes to exchange discrete messages asynchronously.	Client-server communication.
Shared Memory	Fastest IPC; processes share a memory segment. Requires synchronization.	High-speed data sharing like multimedia apps.
Semaphores	Used to synchronize access to shared resources or memory.	Prevent race conditions in shared memory.
Sockets	Bidirectional communication over network or locally (Unix domain	Communication between processes over network or on

	sockets).	the same machine.
Signals	Software interrupts to notify processes of events.	Inform a process of events like termination or interrupts.

1. Pipes

- **Anonymous pipes** are used for communication between parent and child processes.
- Created using the `pipe()` system call.
- Data flows in one direction.
- Example usage: `ls | grep txt` in shell uses pipes.

2. Named Pipes (FIFOs)

- Unlike anonymous pipes, **FIFOs** have a name in the file system (`mkfifo` command).
- Can be used between unrelated processes.
- Communication is still unidirectional but can be opened for reading/writing by different processes.

3. Message Queues

- Provide a queue to send and receive messages in a structured way.
- Use `msgget()`, `msgsnd()`, `msgrcv()` system calls.
- Messages have priorities and types.
- Suitable for asynchronous communication.

4. Shared Memory

- Allows multiple processes to access the same memory space.
- The fastest IPC method because data doesn’t need to be copied.
- Requires synchronization (e.g., semaphores) to prevent conflicts.
- System calls: `shmget()`, `shmat()`, `shmdt()`, `shmctl()`.

5. Semaphores

- Used to control access to shared resources by multiple processes.
- Prevent race conditions and ensure data consistency.
- Two types:
  - **Binary semaphore** (mutex) – lock/unlock.
  - **Counting semaphore** – control access to multiple resources.
- System calls: `semget()`, `semop()`, `semctl()`.

6. Sockets

- Provide communication between processes either on the same machine (Unix domain sockets) or over a network (TCP/IP sockets).
- Support bidirectional data exchange.
- Used in client-server models and distributed applications.
- System calls: `socket()`, `bind()`, `listen()`, `accept()`, `connect()`, `send()`, `recv()`.

7. Signals

- Simple form of IPC to notify a process that an event occurred.

- Signals include `SIGINT`, `SIGTERM`, `SIGKILL`, `SIGCHLD`, etc.
- Handlers can be set up in processes to catch and respond to signals.
- System calls: `kill()`, `signal()`, `sigaction()`.

Summary Table:

IPC Type	Direction	Related Processes Only	Speed	Requires Sync?
Pipe	Unidirectional	Yes	Medium	No
Named Pipe (FIFO)	Unidirectional	No	Medium	No
Message Queue	Bidirectional	No	Medium	No
Shared Memory	Bidirectional	No	Very fast	Yes
Semaphore	N/A	No	N/A	N/A
Socket	Bidirectional	No	Medium	No
Signal	N/A	No	Very fast	N/A

7.6 Memory Management

Linux uses a **complex memory management system** designed to efficiently allocate, protect, and manage memory in a multitasking environment.

1. Memory Layout in Linux

- **User Space:** Memory where user processes run. Each process has its own virtual address space.
- **Kernel Space:** Reserved for the Linux kernel, device drivers, and kernel modules.
- Typically, the split is **3GB user / 1GB kernel** on 32-bit systems (configurable).

2. Virtual Memory

- Linux uses **virtual memory**, meaning each process has its own virtual address space.
- Virtual memory is mapped to physical memory (RAM) using **paging**.
- If RAM is insufficient, Linux swaps out pages to **swap space** on disk.

3. Paging

- Memory is divided into fixed-size blocks called **pages** (usually 4 KB).
- Physical memory divided into **frames** of same size.
- Linux maintains a **page table** for each process to map virtual pages to physical frames.
- Uses **demand paging**: loads pages only when needed.

4. Page Replacement

- When memory is full, Linux swaps out less-used pages to disk (swap space).

- Uses algorithms like **Least Recently Used (LRU)** to decide which pages to swap out.

5. Kernel Memory Allocation

- Kernel memory cannot be swapped.
- Linux uses **slab allocator** and **buddy allocator** to efficiently allocate and free kernel memory.
- **Slab allocator:** caches commonly used objects to speed up allocation.
- **Buddy system:** manages free memory in blocks of sizes that are powers of two.

6. Shared Memory

- Linux supports shared memory IPC via **POSIX** and **System V** shared memory.
- Processes can map the same physical memory to their virtual address spaces.

7. Memory Overcommit

- Linux can **overcommit memory**, allowing processes to allocate more memory than physically available.
- This relies on the assumption that not all allocated memory will be used at the same time.
- Configurable via `/proc/sys/vm/overcommit_memory`.

8. Out Of Memory (OOM) Killer

- When system runs out of memory, Linux invokes the **OOM killer** to terminate processes to free memory.
- It tries to kill processes that free up the most memory with least impact.

9. Monitoring Memory in Linux

- `free -h` — Shows total, used, free, and swap memory.
- `top` or `htop` — Interactive view of memory usage by processes.
- `/proc/meminfo` — Detailed memory statistics.

10. Memory-Related System Calls

- `brk()` and `sbrk()` — Adjust the end of data segment for heap management.
- `mmap()` — Map files or devices into memory; used for shared memory or memory-mapped files.
- `munmap()` — Unmap previously mapped memory.

Summary Table:

Feature	Description
Virtual Memory	Each process has independent address space.
Paging	Memory divided into pages; mapped via page tables.
Swapping	Moves inactive pages to disk swap space.
Slab Allocator	Efficient kernel memory allocator.
Overcommit	Allows allocating more memory than physically present.
OOM Killer	Kills processes when memory exhausted.

Advantages

- Efficient memory use via paging.
- Protection between processes.

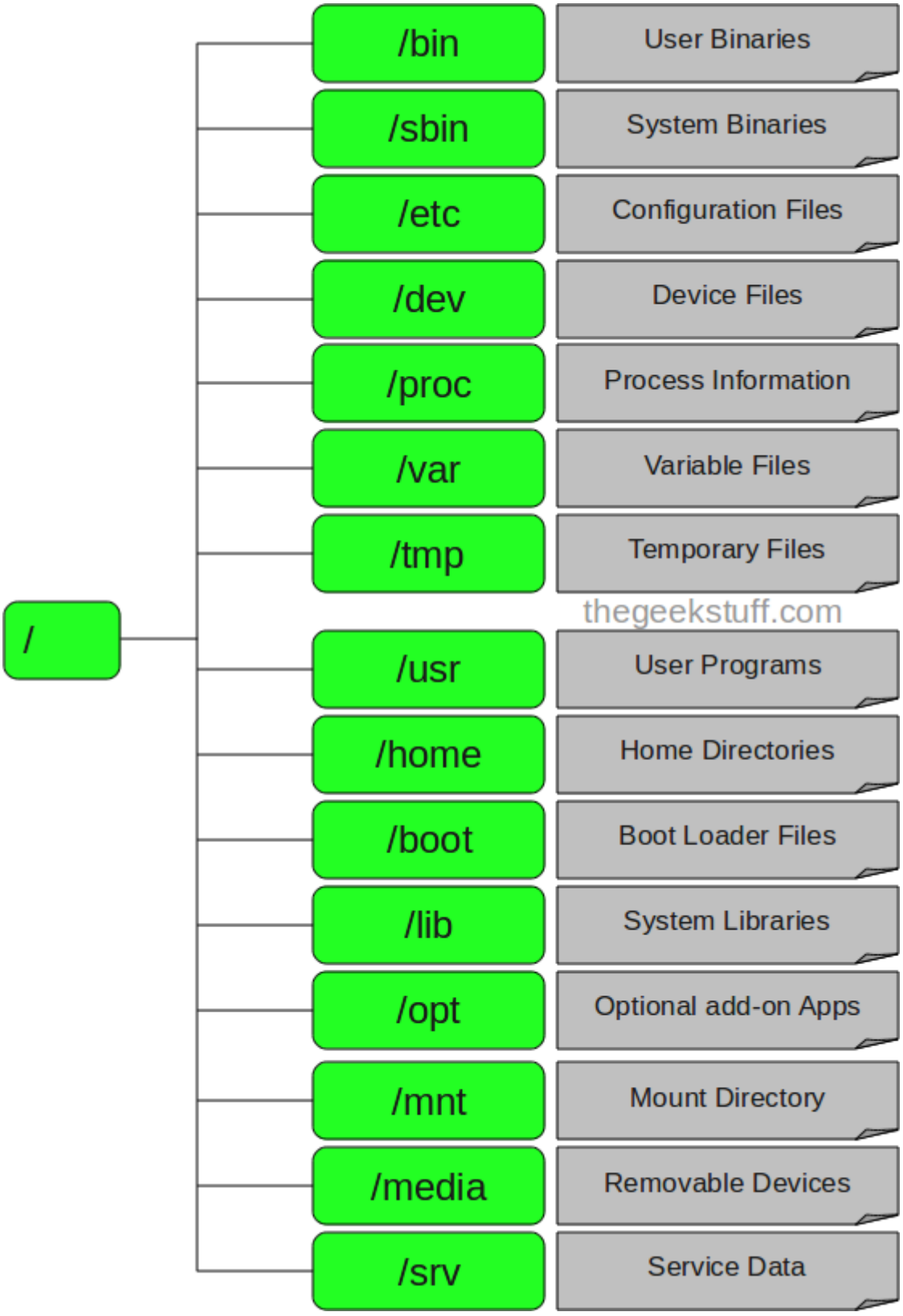
Disadvantages

- Overhead due to page tables.
- Thrashing if excessive swapping occurs.

7.7 File System Management

Linux File Systems

File System	Features
Ext4	Default, journaling, large file support
XFS	High performance, scalability
Btrfs	Snapshots, checksums



Disk Management Commands

Command	Description
`df`	Disk space usage
`du`	Directory space usage
`fsck`	File system check

Example:

```
# Check disk usage
df -h

# Format a partition
mkfs.ext4 /dev/sda1
```

7.8 Device Management

Approaches

1. **Device Files (/dev)** – Represent hardware as files.
2. **udev** – Dynamically manages device nodes.

Example:

```
# List connected devices
ls /dev/sd*
```

Comparison: Monolithic vs. Microkernel

Feature	Linux (Monolithic)	Microkernel
Performance	Faster (in-kernel services)	Slower (IPC overhead)
Stability	Kernel crash affects all	More resilient
