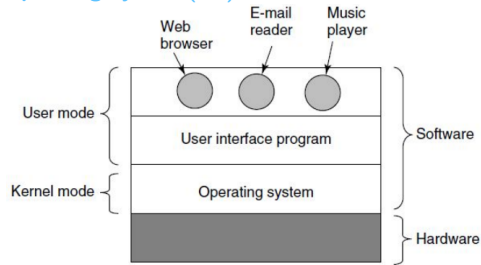


Introduction to Operating Systems

Basic Principles of Operating Systems

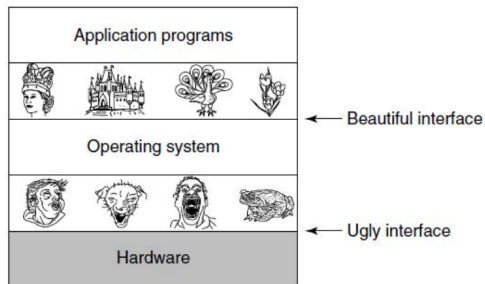
Operating System (OS)



The Operating System as an Extended Machine

The operating system serves as an abstraction layer between hardware and applications (software that manages hardware)

- Hardware: complex/difficult to program directly
- OSs create good abstractions for hardware resources (provide services for programs)
- These abstractions are implemented and managed by the OS
- Provide services for programs: Applications interact with these abstractions rather than directly with hardware



User Mode vs Kernel Mode

OS operate in two fundamental modes:

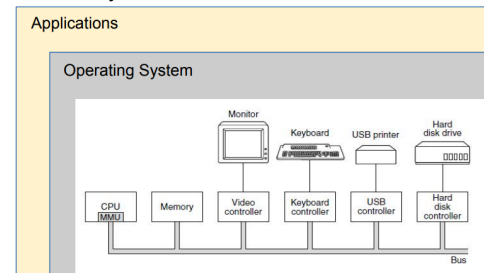
- **Kernel Mode:** Privileged execution mode with full access to all hardware resources and instructions
- **User Mode:** Restricted execution mode where applications run with limited privileges

The OS provides two key interfaces:

- **Northbound Interface:** User Interface Towards user applications
- **Southbound Interface:** Hardware Interface Towards hardware

The Operating System as a Resource Manager

- Process Management
- Memory Management
- File System Management
- Device Management
- Security



- Controls resource usage
- Grants resource requests
- Accounts for resource usage
- Mediates conflicting requests
- Ensures fair and efficient resource allocation

Operating System Variants

Operating System Variants

- Mainframe OS: IBM z/OS, IBM z/VM
- Server OS: Windows Server, Linux, Solaris
- Multiprocessor OS: Windows, Linux, Solaris
- Personal Computer OS: Windows, MacOS, Linux
- Handheld Computer OS: Android, iOS
- Embedded OS: VxWorks, QNX
- Real-Time OS: VxWorks, QNX
- Sensor Node OS: TinyOS, Contiki
- Cloud OS: OpenStack, OpenNebula
- Smart Card OS: JavaCard, MULTOS

Operating Systems vs. Distributions

- **Operating System (Kernel):** The core component that directly manages hardware resources
- **Distribution:** A complete package including kernel, utilities, and applications

Example:

Linux Kernel + GNU Tools + X11 + Gnome
+ Firefox + LibreOffice
= Ubuntu

Basic OS Concepts

Fundamental OS Concepts

- **Interacting with OS:** Through terminal or graphical user interface
- **Users:** Regular vs. privileged users
- **Data organization:** Files, directories, filesystems
- **Programs vs. Processes:** A program is a compiled executable, while a process is an instance of a running program
- **Multi-tasking:** Running multiple processes concurrently
- **Context-switching:** Switching execution from one process to another
- **System calls:** Interface for processes to request OS services
- **Inter-Process Communication:** Methods for processes to communicate
- **Signals:** Mechanism for notifying processes of events

OS Structures

Operating systems can be organized in different ways:

- **Monolithic Systems:** Single executable containing all OS functionality
- **Modular Systems:** Core kernel with loadable modules
- **Microkernels:** Minimal kernel with most services running in user space

To check CPU information in Linux:

```
1 # Display kernel version information
2 uname -a
3
4 # View CPU details
5 less /proc/cpuinfo
6
7 # Display CPU architecture information
8 lscpu
```

Working with Processes in Linux

Viewing running processes

- Use `ps aux` to display all processes
- Use `top` for an interactive process viewer
- Check environment variables with `env`

Process control

- Background a process with `&` or `bg`
- Bring to foreground with `fg`
- List background jobs with `jobs`
- Terminate a process with `kill PID`

System information

- View process hierarchy with `ps tree`
- Check system call activity with `strace`

Computer Hardware Review

CPU and Memory Architecture

CPU Central Processing Unit

- Basic cycle: Fetch, Decode, Execute
- CPUs feature some registers to hold key variables and temporary results
- Special registers for internal use: Program Counter (PC), Stack Pointer (SP), Program Status Word (PSW)
- Execution units and pipelines for processing instructions
- Cache memory organized in hierarchical levels (L1, L2, L3)

CPUs and their Instruction Sets are architecture-specific:

- ARM, RISC, X86, etc.
- Instructions are classified along Execution Privileges, enforced by CPU:
 - Intel: Priority Rings → User Mode (limited set of instructions) vs Kernel Mode (full set of instructions)
 - ARM: UnPrivileged Mode vs Privileged Mode

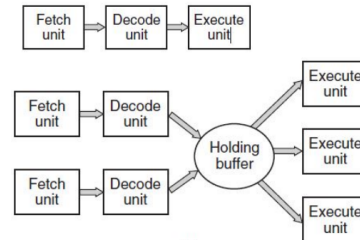
CPU cycles

Basic cycle of every CPU:

- Fetch instructions from memory into registers
- Decode the instruction to determine type and operands
- Execute the instruction
- Repeat for subsequent instructions

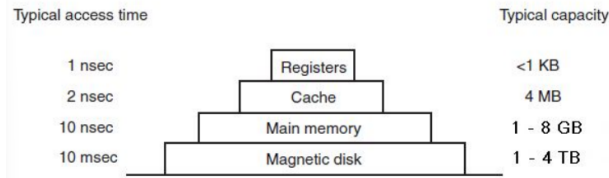
CPUs have multiple cores:

each having multiple execution units and parallel pipelines



Memory The memory system is constructed as a hierarchy of layers, based on access time:

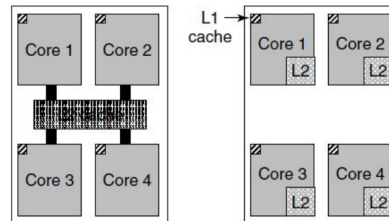
- Registers (fastest, smallest)
- Cache memory (L1, L2, L3)
- Main memory (RAM)
- Secondary storage (disks, SSDs)
- Tertiary storage (backup systems)



CPU Caches multiple levels of caches:

- L1: Small, fast, close to CPU
- L2: Larger, slower, further away
- L3: Even larger, even slower, even further away
- Caches are used to store frequently accessed data and instructions

Example: Quad-core chip with shared L2 cache and quad-core chip with separate L2 caches for each core.



I/O System

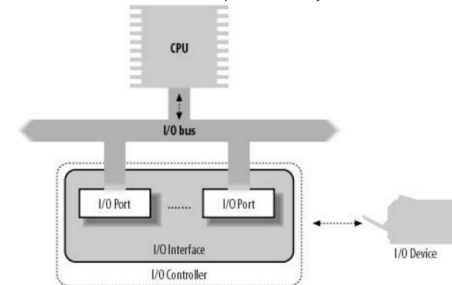
Input/Output (I/O) System

The I/O system comprises:

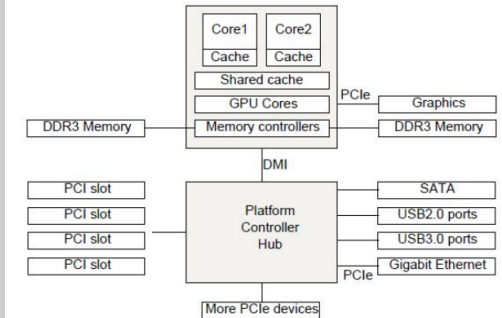
- Various devices (hard drives, network interfaces, serial ports, keyboards, etc.)
- Bus systems to connect devices (PCI, PCIe, SATA, USB, etc.) (software)
- I/O controllers (hardware)
- Device drivers (software)

Device drivers:

- Software that communicates with I/O controllers (commands/responses)
- Can run in Kernel or User Mode
- May be built into the kernel or modular (loaded as modules at boot/run time)



IO and Bus System X86 System with different Bus standards:



Terminal Basics

Terminal

A terminal is a text-based user interface program that allows users to:

- Accepts text input via a prompt
- Interprets text as commands
- Executes operations based on user input
- Returns output to the user

Terminal commands can be:

- Binary programs loaded from disk (e.g., `mkdir`)
- Built-in functions of the terminal (e.g., `cd`)
- Operations like regular expressions for complex command preparation

Essential Terminal Commands

System information

- `whoami` - Display current user
- `uname -a` - Display kernel information
- `lsmod` - List loaded kernel modules
- `dmesg` - Display kernel messages

Command output manipulation

- `command > file.txt` - Redirect output to a file
- `command | grep pattern`
 - Filter output through `grep`
- `clear` - Clear terminal screen

System control

- `env` - Display environment variables
- `exit` - Exit the terminal
- `shutdown` - Shutdown the system

User Management

User Management in Linux Tools:

- `users` - List users currently logged in
- `who` - Show who is logged in
- `id` - Display user and group IDs
- `passwd` - Change user password
- `usermod` - Modify user account
- `groupmod` - Modify a group

Administrative access:

- `root` - Account with full system privileges
- `sudo` - Execute command as root user
- `su` - Switch user
- `su --` - Switch user and load their environment

Process Management

Process Management Commands

Commands for monitoring and controlling processes:

- `ps` - Display current processes
- `top` - Interactive process viewer
- `pstree` - Display process tree
- `pidof` - Find process ID of a program
- `kill` - Send signal to process
- `kill -9` - Force terminate process
- `killall` - Kill processes by name

Files, Directories, and Filesystems

File System Operations

Basic file and directory operations:

- `pwd` - Print working directory
- `cd` - Change directory
- `ls -al` - List all files with details
- `touch` - Create empty file or update timestamp
- `mkdir` - Create directory
- `tree` - Display directory structure as a tree
- `cp` - Copy files/directories
- `mv` - Move/rename files/directories
- `rm` - Remove files/directories

File permissions and ownership:

- `chown` - Change file owner
- `chmod` - Change file permissions

Disk and filesystem management:

- `fdisk -l` - List disk partitions
- `mount` - Mount a filesystem

Working with Files in Linux

Creating and viewing files

- `touch filename` - Create empty file
- `cat filename` - Display entire file contents
- `less filename` - View file with pagination
- `tail filename` - Display last 10 lines of file
- `tail -f filename` - Continuously monitor file for changes

Text editing with vim

- `vim filename` - Open file in vim
- Press `i` for insert mode
- Press `Esc` to exit insert mode
- Type `:w` to save
- Type `:q` to quit
- Type `:wq` to save and quit
- Type `:q!` to quit without saving

System Information and Configuration

System Information Commands

Commands for system information/configuration:

- `hwinfo` - Hardware information
- `lshw` - List hardware
- `/proc` - Virtual filesystem for kernel information
- `/etc` - Configuration files directory
- `sysctl` - Read/modify kernel parameters
- `systemctl` - Control the systemd system and service manager

Network-related commands:

- `ifconfig` - Configure network interfaces
- `ip` - Show/manipulate routing, devices, policy routing
- `dig` - DNS lookup utility

- `hostname` - Show or set system hostname

Package management:

- `apt-get` - Package handling utility

Working with the Linux terminal

```
1 # Check current user
2 whoami
3
4 # View hardware information
5 lshw -short
6
7 # Monitor system messages
8 dmesg | grep usb
9
10 # Check disk usage
11 df -h
12
13 # Find a process ID
14 pidof firefox
15
16 # Install a package
17 sudo apt-get install htop
```

OpenStack Lab Environment Setup

Initial setup

- Connect to ZHAW VPN if not at ZHAW facilities
- Navigate to OpenStack Horizon dashboard: <https://ned.cloudlab.zhaw.ch>
- Log in with provided credentials
- Change password if using default credentials

Creating SSH key pair

- Go to Compute → Key Pairs → Create Key Pair
- Download the private key file (*.pem)
- Set appropriate permissions:
`chmod 600 key.pem`

Creating a VM

- Go to Compute → Instances → Launch Instance
- Select Ubuntu image
- Attach to 'internal' network
- Set SSH key and security groups
- Launch VM and associate a Floating IP

Connecting to VM

- SSH to the VM:
`ssh -i key.pem ubuntu@IP_ADDRESS`

Linux Lab Basic Tasks Creating/managing files:

```
1 # Create a test file
2 touch delta.txt
3 # Add content to the file
4 echo "Hello, this is a test" >
   delta.txt
5 # View the file content
6 cat delta.txt
7 # Stop and restart the VM from
   OpenStack dashboard
8 # Then check if the file persists
9 cat delta.txt
```

Booting an OS

Boot Process and Initialization

Boot Process Stages Typical boot process follows these steps:

- Hardware initialization (BIOS/UEFI)
- Bootloader execution
- Kernel loading and initialization
- System initialization (services, environment)
- User interface presentation

BIOS and Bootloader

BIOS and Hardware Initialization

BIOS (Basic Input Output System) is the first program loaded on boot:

- Stored in ROM on the motherboard
- Contains low-level I/O software for interfacing with devices
- Performs Power-On Self-Test (POST)
- Discovers devices by scanning PCI buses
- Initializes hardware devices
- Selects a boot device from the list in CMOS
- Reads the first sector from the boot device (Master Boot Record)
- Loads MBR into memory and executes it

BIOS vs UEFI

BIOS Limitations

Traditional BIOS has several limitations:

- Operates in 16-bit mode
- Relies on MBR (Master Boot Record) from 1983
- Limited partition number and size (2 TB max)
- Not designed for extendability
- Vulnerable to rootkit and bootkit attacks

Unified Extensible Firmware Interface (UEFI)

UEFI is the modern replacement for BIOS:

- Replaces MBR with GPT (GUID Partitioning Table)
- Supports arbitrary number of partitions
- Addresses disk space up to 2⁶⁴ bytes
- Uses unique UUIDs for partitions to avoid collisions
- Features modular design for extendability
- Includes Secure Boot to restrict which binaries can be executed
- Uses cryptographic signatures and X.509 certificates for verification

Bootloader Function

The bootloader (loaded from MBR):

- Accesses the boot partition containing the OS
- Loads the OS kernel into memory
- Sets up registers (Program Counter, Processor Status Word)
- Transfers control to the OS by jumping to the first instruction

GRUB (GRand Unified Bootloader)

GRUB is a common bootloader for Linux systems:

- Requires filesystem access to read the OS
- Contains device drivers and filesystem modules
- Provides a menu to select the OS to boot
- Loads the selected kernel and optional initial RAM disk (initrd)
- Passes control to the kernel

UEFI Functioning

UEFI uses an architecture-independent virtual machine:

- Executes special binary files compiled for UEFI (*.efi)
- These files can be device drivers, bootloaders, or extensions
- Files are stored in the EFI System Partition (ESP)
- ESP uses FAT filesystem and can be reused in multi-boot systems
- EFI Boot Manager configures which EFI binary to execute

Examining UEFI boot configuration:

```
1 # Display current boot entries
2 sudo efibootmgr -v
3
4 # List contents of EFI system
5 partition
6 sudo ls -la /boot/efi/EFI/
7
8 # View disk partition table format
9 (MBR or GPT)
10 sudo fdisk -l
```

OS Kernel Initialization

Kernel Initialization

The OS kernel initialization process:

- Queries hardware information
- Loads and initializes device drivers
- Initializes internal management structures (e.g., process table)
- Sets up virtual memory and memory management
- Creates system services
- Launches the first user process (init)

Linux Kernel Initialization Phases

- Architecture-specific assembly code
 - Sets up OS memory map
 - Identifies CPU type
 - Calculates total RAM
 - Disables interrupts
 - Enables MMU and caches
- C main() procedure
 - Initializes process tables, interrupt/system-call tables
 - Sets up virtual memory and page cache
 - Configures resource control
 - Loads drivers and initializes OS services

Initial RAM Disk (initrd/initramfs)

Initial RAM Disk

The initial RAM disk addresses the "chicken-and-egg" problem:

- OS needs drivers to access hard drive and its filesystem
- These drivers are stored on the hard drive itself
- Solution: initrd/initramfs provides temporary root filesystem
- Contains kernel modules and basic device files
- Bootloader uncompresses both kernel and initrd into RAM
- Kernel mounts initrd as the initial filesystem
- Kernel uses tools found in initrd to find and mount the real filesystem

Examining GRUB configuration for kernel and initrd:

```
1 # View GRUB configuration
2 cat /boot/grub/grub.cfg
3
4 # Examine a specific menu entry showing kernel and initrd paths
5 grep -A 10 "menuentry 'Ubuntu'" /boot/grub/grub.cfg
```

System Services Initialization

System V Init

Traditional System V initialization:

- Init process runs initialization shell scripts from /etc/rc# directories
- Uses predefined runlevels (0-6) to determine system state
- Each runlevel has a set of services defined in scripts
- Dependencies among services are coded in the scripts themselves
- Results in complex initialization process

System V runlevels:

- 0: Halt - Shuts down the system
- 1: Single-user mode - Administrative tasks
- 2: Multi-user mode without networking
- 3: Multi-user mode with networking
- 4: Not used/user-definable
- 5: Multi-user mode with GUI
- 6: Reboot - Restarts the system

systemd

- Modern initialization system (systemd):
- Replacement for SysV Init
 - Provides coordinated and parallel service startup
 - Features on-demand activation and runtime management
 - Uses dependency-based service control logic
 - Takes a holistic management approach for the entire system

systemd User Services

- systemd supports user instance services:
- Services managed by individual users without requiring root privileges
 - User service units are stored in:
 - Units provided by packages: `/usr/lib/systemd/user/`
 - User-installed package units: `~/.local/share/systemd/user/`
 - System-wide user units: `/etc/systemd/user/`
 - User's own units: `~/.config/systemd/user/`
 - By default, user services start on login and stop on logout
 - Linger allows user services to start at boot without login

systemd Dependencies

- systemd supports various dependency types:
- **Requires=**
Units that must be started when this unit is started
 - **Wants=**
Units that should be started (but not required) when this unit is started
 - **Conflicts=**
Units that must be stopped when this unit is started
 - **After=**
This unit should be started after the listed units
 - **Before=**
This unit should be started before the listed units

systemd Units

- systemd organizes system components as 'units' which:
- encapsulate system objects (services, mounts, devices, etc.)
 - have states (active, inactive, activating, deactivating, failed)
 - can depend on other units
 - (most) are configured in unit configuration files
 - (some) can be created automatically or programmatically

Common unit types:

- `.service` - A system service
- `.target` - A group of systemd units (similar to runlevels)
- `.mount` - A filesystem mount point
- `.device` - A device file recognized by the kernel
- `.socket` - An inter-process communication socket
- `.timer` - A systemd timer

systemd Service Unit File

A systemd service unit file consists of three sections:

```
1 [Unit]
2 Description=Example Service
3 Documentation=https://example.com/docs
4 After=network.target
5 Wants=network-online.target
6 Requires=example-dependency.service
7
8 [Service]
9 Type=simple
10 ExecStart=/usr/bin/example-service
11 Restart=on-failure
12 User=exampleuser
13
14 [Install]
15 WantedBy=multi-user.target
```

The sections have specific purposes:

- Unit: Metadata and dependencies
- Service: Execution configuration for services
- Install: Configuration for enabling the unit

Working with systemd

Viewing unit status

- `systemctl status -all` - Show status of all units
- `systemctl status SERVICE` - Show status of specific service
- `systemctl list-units -t service` - List only service units
- `systemctl list-unit-files -type service` - List service unit files
- `systemctl cat SERVICE` - View the content of a unit file

Managing units

- `systemctl start SERVICE` - Start a service
- `systemctl stop SERVICE` - Stop a service
- `systemctl restart SERVICE` - Restart a service
- `systemctl enable SERVICE` - Enable service autostart
- `systemctl disable SERVICE` - Disable service autostart

Working with targets

- `systemctl list-units -t target` - List available targets
- `systemctl get-default` - Show default target
- `systemctl set-default TARGET` - Set default target

Creating a Simple systemd Service

Creating a custom service to write to a file when started:

```
1 # Create a service unit file
2 sudo nano /etc/systemd/system/myservice.service
3
4 # Add the following content
5 [Unit]
6 Description=My Simple Service
7 After=network.target
8
9 [Service]
10 Type=simple
11 ExecStart=/bin/bash -c 'echo "Service started at $(date)" >
12     /home/ubuntu/service_started'
13 Restart=on-failure
14
15 [Install]
16 WantedBy=multi-user.target
17
18 # Reload systemd to recognize the new service
19 sudo systemctl daemon-reload
20
21 # Start the service
22 sudo systemctl start myservice
23
24 # Verify it worked
25 cat /home/ubuntu/service_started
26
27 # Enable auto-start on boot
28 sudo systemctl enable myservice
```


Processes and Threads

Processes

Process Model

Programs vs. Processes A program is fundamentally different from a process:

- Program:** A compiled executable (binary)
 - Set of CPU instructions and related data
 - Targets a specific platform (OS + hardware)
 - Static entity stored on disk
- Process:** An active instance of a program
 - Has a program, input, output, and state
 - Dynamic entity in memory
 - Multiple instances of the same program can run as separate processes

Process Characteristics

- Processes have several important characteristics:
- Can run sequentially or in parallel (multi-tasking)
 - Selected for execution by the OS scheduler
 - Associated with an owner (defines access privileges)
 - Run within an environment (environment variables)
 - Can run in foreground (interactive) or background (non-interactive)
 - Can be user processes or system processes

Viewing process information in Linux

```
1 # List all processes with details
2 ps aux
3 # Interactive process viewer
4 top
5 # Show environment variables
6 env
7 # Run a process in the background
8 wc /dev/zero &
9 # List background jobs
10 jobs
11 # Bring a background job to foreground
12 fg %job_number
```

Process Creation

Process Creation by the OS

- When creating a process, the OS performs several operations:
- Loads the executable into RAM
 - Sets up the memory map
 - Updates scheduler and process table entries
 - Sets program counter and stack pointer
 - Switches from system mode to user mode

Memory Layout

- Process memory is typically divided into several segments:
- **Text:** Contains CPU instructions and constant data
 - **Data:** Global and static variables
 - Initialized data segment: Pre-initialized variables
 - Uninitialized data segment (BSS): Zero-initialized variables
 - **Stack:** Local variables and function call information (LIFO structure)
 - **Heap:** Dynamically allocated memory (controlled by the programmer)

Process Creation Events

- Processes can be created in several ways:
- System boot (initial processes)
 - User request (launching an application)
 - Process creating another process (fork/exec)
 - Scheduled creation (cron jobs)
 - System request (responding to interrupts)

Parent-Child Process Relationship

- When a process creates another process:
- The creating process becomes the parent
 - The new process becomes the child
 - Child's memory map is initially a copy of the parent's memory map
 - Two memory handling approaches:
 - Distinct address space: Separate memory regions for parent and child
 - Copy-on-write: Memory shared until a change is made by either process
 - Forms a process hierarchy (starting with PID 1)

Linux Process Creation

Linux terminal command execution involves process creation:

```
1 # When you run a command in the terminal:
2 ls -la
3
4 # The shell:
5 # 1. Forks itself (duplicate the process)
6 # 2. Child process executes the 'ls' binary
7 # 3. Parent waits for child to complete
8 # 4. Shell continues after child terminates
```

This process can be visualized with `pstree` showing the parent-child relationships.

Process Termination

Process Termination

- A process can terminate in several ways:
- Voluntary normal exit (job completed, return from `main()`)
 - Voluntary error exit (required resource unavailable)
 - Involuntary error exit (segmentation fault, division by zero)
 - Termination by another process (kill signal)

Process termination in Linux:

```
1 # Start a process
2 wc /dev/zero &
3
4 # Get its process ID
5 pidof wc
6
7 # Terminate the process
8 kill -9 <PID>
```

Process States

Process States

- Each process has a life-cycle with specific states:
- **Running:** The process is currently executing on the CPU
 - **Ready:** The process is ready to run but waiting for CPU allocation
 - **Blocked:** The process is unable to run (waiting for an event or resource)
 - Dependencies not met for running
 - Waiting for external resource
 - Waiting for I/O completion
 - Sleeping
 - Under job control or debugger

User Mode vs. System Mode

- CPU supports two execution modes:
- **User Mode:** Limited privileges
 - Application logic execution
 - Application data manipulation
 - Restricted access to hardware and system resources
 - **System Mode (Kernel Mode):** Full privileges
 - System management operations
 - Hardware access
 - Interrupt handling
 - Device management

Process State Changes

- Processes change state for various reasons:
- **Timer expiration:** CPU allocation time ended
 - **Interrupt:** Hardware/resource calls for service
 - **Page fault:** Data not in memory, requires disk access
 - **System call:** Explicit OS service request
- State changes involve a switch between user mode and system (kernel) mode.

Context Switch vs. Mode Switch

- Two types of switches occur during system operation:
- **Mode Switch:** Transition between user mode and kernel mode
 - Occurs during system calls, interrupts, exceptions
 - Same process continues execution in different mode
 - No scheduler involvement
 - **Context Switch:** Changing from one process to another
 - Involves saving the state of the current process
 - Loading the state of another process
 - Typically involves mode switches (user → kernel → user)
 - Requires scheduler involvement

Process Management

Process Control Block

- The OS maintains a Process Control Block (PCB) for each process:
- Process identification (PID, UID, GID)
 - Process state information
 - Program counter and CPU registers
 - CPU scheduling information (priority)
 - Memory management information
 - I/O status information
 - Accounting information
- In Linux, PCBs are implemented as `task_struct` entries in the process table.

Threads

Threads

Threads are execution entities within a process:

- Created and owned by a process
- Share the address space and all data of the owning process
- Allow multiple executions to take place within the same process environment
- Enable a process to continue even when some operations would block
- Cooperate toward the objective of the owning process

Each thread has:

- Program counter (tracks next instruction)
- Registers (hold working variables)
- Stack (with frames for procedure calls)

Thread Implementation Approaches

Threads can be implemented in different ways:

- **User Space Threading (M:1):**
 - All threads in user space appear as a single process to the OS
 - Thread functionality provided by a library
 - Scheduling handled by the process (no mode switch required)
 - Based on cooperation (threads voluntarily yield CPU)
 - Disadvantages: Single thread can monopolize CPU time, blocked threads block the entire process, no SMP advantage
- **Kernel-supported Threading (1:1):**
 - One kernel structure per user thread
 - Kernel schedules all threads
 - Advantages: Better handling of blocking I/O, full SMP exploitation
 - Disadvantages: Higher overhead, slower creation/removal, mode switching for scheduling

Kernel Threads

Kernel threads are special processes that:

- Run exclusively in system (kernel) mode
- Have a PID and state like any process
- Are listed in the process table but flagged as "kernel thread"
- Are scheduled and dispatched like regular processes
- Are uninterruptible (need to voluntarily yield CPU)
- Listen to kernel signals

Kernel threads are organized around working queues and thread pools:

- Working queues ordered by priority
- Mapped onto a pool of reusable kernel threads
- Number of threads dynamically managed based on workload
- Managed by the kthread daemon

Thread Implementation in Linux

Linux doesn't have a dedicated concept of threads:

- All threads are standard processes (tasks)
- A thread is a process that shares resources with other processes
- Shared resources can include: address space, file descriptors, sockets, signal handlers, etc.
- Implemented using system calls: `fork()` and `clone()`
- Two implementation frameworks:
 - LinuxThreads (legacy)
 - NPTL (Native POSIX Thread Library, current standard)

Linux Process and Thread Creation

Linux Process vs. Thread Creation

Linux offers different system calls for process and thread creation:

- `fork()`: Creates a child process by duplicating the parent
 - Child and parent run in separate memory spaces
- `clone()`: Provides precise control over what execution context is shared
 - Allows sharing of address space, file descriptors, signal handlers, etc.
 - Used to implement threads in Linux

Process and Thread Management in Linux

Viewing process information

- `ps aux` - List all processes
- `ps -ef` - List processes in full format
- `ps -eLF` - List processes and threads
- `top` - Interactive process viewer
- `top -H` - Show threads in top
- `pstree` - Display process tree

Creating processes

- Write a C program using `fork()` to create a child process
- Use `getpid()` to retrieve the process ID
- Use `sleep()` to pause execution

Creating threads

- Use POSIX threads library (pthread)
- Include `<pthread.h>`
- Create threads with `pthread_create()`
- Join threads with `pthread_join()`

Identifying zombie processes

- Create a parent process that doesn't wait for its child
- Child terminates while parent continues running
- Check process state with `ps` (state `SZ` indicates zombie)

Creating Processes in C

Simple process creation with `fork()`:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 int main() {
6     pid_t pid = fork();
7
8     if (pid < 0) {
9         // Error
10        fprintf(stderr, "Fork failed\n");
11        return 1;
12    } else if (pid == 0) {
13        // Child process
14        printf("Child process: PID = %d\n", getpid());
15        sleep(5);
16    } else {
17        // Parent process
18        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
19        sleep(10);
20    }
21
22    return 0;
23 }
```

Creating Threads in C

Simple thread creation with POSIX threads:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 void* thread_function(void* arg) {
7     int thread_id = *(int*)arg;
8     printf("Thread %d: running\n", thread_id);
9     sleep(3);
10    printf("Thread %d: exiting\n", thread_id);
11    return NULL;
12 }
13
14 int main() {
15     pthread_t threads[2];
16     int thread_args[2];
17
18     for (int i = 0; i < 2; i++) {
19         thread_args[i] = i;
20         pthread_create(&threads[i], NULL, thread_function, &thread_args[i]);
21         printf("Main: created thread %d\n", i);
22     }
23
24     // Wait for threads to finish
25     for (int i = 0; i < 2; i++) {
26         pthread_join(threads[i], NULL);
27     }
28
29     printf("Main: all threads completed\n");
30     return 0;
31 }
```

Scheduling

Scheduling Problem Domain

Scheduling is a resource-time management activity:

- Focuses on managing CPU time allocation among processes
- Requirements vary by platform (mobile vs. super-computer)
- Requirements vary by application type (batch processing vs. real-time)

Processes can be categorized by behavior:

- **CPU-bound:** Computationally intensive tasks (e.g., image processing)
- **I/O-bound:** Tasks that frequently wait for I/O operations (e.g., interactive applications)

When to Schedule

- When a new process is created
- When a process exits
- When a process blocks on I/O
- When an I/O interrupt occurs
- Regularly on a timer

Based on when scheduling occurs, schedulers can be:

- **Non-preemptive:** Only schedules when a process blocks or terminates
- **Preemptive:** Can interrupt a running process and schedule another

Queueing Theory

 base of Scheduling

- Deals with waiting for and dispatching resources
- Aims to provide sufficient resources to avoid under-capacity
- Ensures urgent tasks are not kept waiting

Scheduling Queues

 Several Types:

- **Ready queue:** Processes waiting to be executed
- **Device queues:** Processes waiting for specific devices
- **Job queue:** All processes in the system

The scheduler sorts the ready queue according to policy, and the dispatcher moves the process from the head of the queue to the CPU.

Scheduling Metrics

 used to evaluate performance:

- **CPU utilization:** Keeping the CPU as busy as possible
- **Throughput:** Number of processes completed per time unit
- **Turnaround time:** Time from submission to completion
- **Waiting time:** Time spent in the ready queue
- **Response time:** Time from request to first response
- **Fairness:** Equal distribution of CPU time

Scheduling Algorithms

Simple Schedulers

- **Uniprocessing:** One machine, one task - no scheduler required
- **Multiprocessing:** Single task running at one time with job control
- **Multitasking:** Multiple tasks running with scheduler

Multi-level Schedulers

 address priority needs:

- Multiple queues with different priorities
- Round Robin within each queue
- Challenge: High-priority tasks can cause starvation of low-priority tasks

Multi-level Feedback Queues

Addresses starvation (tasks move between queues):

- Task priority sinks after each execution interval
- Low-priority queues may get larger time quantum
- Balances responsiveness with throughput

Fair Share Scheduling

 aim for equal CPU time:

- Maintains a running clock of CPU time used per process
- Uses a Virtual Clock (VC) to track usage
- Ensures average run-time is roughly equal for all tasks
- Better balance between CPU-bound and I/O-bound tasks

First-In-First-Out (FIFO)

basic scheduling algorithm:

- Processes are executed in the order they arrive
- Single queue, no preemption
- Processes run to completion
- Simple to implement
- Non-preemptive
- No awareness of process type (interactive vs. batch)

→ also known as First-Come-First-Served (FCFS)

Round Robin (RR)

 time-sharing algorithm:

- Each process runs for a fixed time slice (quantum)
- After quantum expires, process is moved to the end of the queue
- Preemptive, as running tasks are interrupted after quantum
- Arriving tasks have priority over adjourned tasks (by convention)
- No starvation, as all processes eventually get CPU time
- Performance depends on quantum size
 - Too large: approaches FIFO
 - Too small: too much context switching overhead

Real-Time Schedulers

Real-Time Systems

 Real-time systems have specific scheduling needs:

- Need responsiveness to I/O
- Must fulfill deadlines (hard or soft)
- Hard deadlines **MUST** be met, soft deadlines can occasionally be missed
- Deadlines may be in milliseconds, seconds, or hours

A Real-Time Operating System (RTOS):

- Completes system calls in deterministic time
- Schedules user tasks to meet deadlines
- Facilitates hard real-time requirements

Rate Monotonic Scheduling

 Static-priority

scheduling algorithm for real-time systems:

- Higher repetition rate tasks → higher priority
- Guaranteed schedule if utilization meets criteria:

$$U = \sum_{i=1}^n \frac{C_e}{T_r} \leq n(2^{1/n} - 1)$$

Where C_e is execution time and T_r is period

- Max. guaranteed utilization converges to ~69%
- Schedule determined at compile-time, not run-time

Earliest Deadline First (EDF)

Dynamic scheduling algorithm for real-time systems:

- Scheduler determines the task with the next deadline
- Task with the earliest deadline gets highest priority
- Schedule is achievable if utilization does not exceed 100%
- Can achieve full CPU utilization
- Schedule determined at run-time, not compile-time

Linux Scheduling

Linux Scheduling Policies

Real-Time Policy: For time-critical tasks

- SCHED_DEADLINE: Earliest Deadline First + Constant Bandwidth Server
- SCHED_FIFO & SCHED_RR (see algorithms)

Normal Policy: For regular tasks

- SCHED_OTHER: default (CFS)
- SCHED_BATCH: For batch processing tasks
- SCHED_IDLE: extremely low priority bg. tasks

Linux Priority System

 Kernel vs. User Space:

- **Kernel space:** Priorities from high to low
 - RT: 0-99
 - Normal: 100-139
- **User space:** 'nice' values from -20 (high) to +19 (low)
 - Maps to kernel priorities:
nice + 20 = kernel priority - 100
 - Not used in RT policies

SCHED_DEADLINE

 Highest priority scheduler:

- Implements Earliest Deadline First + Constant Bandwidth Server
- Takes parameters: runtime, period, and deadline
- Tasks scheduled with this cannot fork
- Tasks may yield CPU time when not needed

SCHED_FIFO

 FIFO real-time scheduler:

- Uses one queue per priority level (1-99)
- Higher priority than SCHED_RR at same priority level
- Immediately preempts any Normal policy thread
- Runs to completion unless:
 - Preempted by a higher priority RT thread
 - Blocked by I/O call
 - Voluntarily yields the CPU

SCHED_RR

 Round Robin real-time scheduler:

- Similar to SCHED_FIFO but with time quanta
- When quantum expires, thread moved to end of its priority queue
- Quantum size configurable (default 100ms)
- RT bandwidth limiting prevents RT tasks from monopolizing CPU

Completely Fair Scheduler (CFS)

Default scheduler in Linux (SCHED_OTHER):

- Uses a red-black tree sorted by execution time ($O(\log N)$ operations)
- Tracks virtual runtime to achieve fairness
- Considers 'nice' values to adjust CPU share
- Tasks can be grouped and scheduled together
- Aims to model an 'ideal, precise multi-tasking CPU'
- Time accounting managed according to configurable granularity

SCHED_BATCH and SCHED_IDLE

Low-priority schedulers:

- **SCHED_BATCH:**
 - Same static priority as SCHED_OTHER
 - Designed for batch-type, CPU-intensive tasks
 - Applies penalty due to CPU usage
 - SCHED_OTHER has precedence over SCHED_BATCH at same nice value
- **SCHED_IDLE:**
 - Extremely low priority
 - Lower than static priority 0 and nice 19
 - Used for background tasks that should only run when system is idle

Multi-core Scheduling

- Load Balancing** implemented on multicore systems:
- Dynamic distribution of tasks across CPU cores
 - Applied based on scheduling policy
 - Balances competing concerns:
 - Moving tasks incurs management overhead
 - Moving tasks incurs cache penalty (lost cache advantage)

- Cache Affinity** affects scheduling decisions:
- Task data may remain in CPU cache after context switch
 - Rerunning on the same core avoids cache misses
 - Linux considers estimated cache live-time when migrating tasks
 - Default cache live-time:
/proc/sys/kernel/sched_migration_cost_ns

Analyzing Scheduling in Linux

- Viewing scheduler information
- Check process priorities: `ps -el` (PRI and NI columns)
 - View real-time processes: `ps -eo pid,cls,pri,rtprio,nice,cmd | grep -E 'CLS|FIFO|RR'`
 - Check scheduler statistics: `cat /proc/schedstat`
 - See current I/O scheduler: `cat /sys/block/sda/queue/scheduler`

- Modifying process priorities
- Start process with nice value: `nice -n [value] command`
 - Change nice value: `renice [value] -p [pid]`
 - Set real-time priority: `chrt -f [priority] command` (SCHED_FIFO)
 - Set round-robin priority: `chrt -r [priority] command` (SCHED_RR)

- Analyzing schedule
- Trace scheduling events: `trace-cmd record -e sched`
 - View trace results: `trace-cmd report`
 - Check CPU affinity: `taskset -p [pid]`
 - Set CPU affinity: `taskset -c [cpu_list] -p [pid]`

FIFO and RR Scheduling Comparison Given the following task list:

Task	Arrival Time	Burst Time
T1	0	10
T2	3	6
T3	7	1
T4	8	3

FIFO Schedule:
T1 runs from 0-10, T2 runs from 10-16, T3 runs from 16-17, T4 runs from 17-20

Round Robin Schedule (quantum = 2):
T1(0-2), T1(2-4), T2(4-6), T1(6-8), T2(8-10), T3(10-11), T4(11-13), T1(13-15), T2(15-17), T1(17-19), T1(19-20)

RR provides better response time but potentially longer turnaround time due to context switches.

Rate Monotonic Scheduling Given tasks with the following characteristics:

Task	WCET (C)	Period (T)
T1	10	20
T2	10	50
T3	5	30

- Analysis:**
1. T1 has the highest priority (shortest period)
 2. Utilization: $U = \frac{10}{20} + \frac{10}{50} + \frac{5}{30} = 0.5 + 0.2 + 0.167 = 0.867$
 3. Maximum utilization for 3 tasks: $3(2^{1/3} - 1) \approx 0.779$
 4. Since $0.867 > 0.779$, there is no guaranteed schedule
- However, a feasible schedule might still exist. Testing would be required to confirm.

Resource Control

Resource Competition

Operating systems must manage competition for resources (res.):

- **Physical resources:** Devices, bus systems, memory, etc.
- **Virtual resources:** Timers, locks, etc.

Resource control requires:

- **Visibility:** Knowing what resources are available
- **Access control:** Determine who can use res.
- **Usage monitoring:** Track resource consumption
- **Unified approach:** Coordinated res. management

Nice Process Concept

Traditional Unix approach to resource control:

- Users can be 'nice' by voluntarily releasing resources
- Modified by the nice command
- Limitations:
 - No global definition, only relative values
 - No strict and precise enforcement
 - Users can only modify niceness of their own processes
 - Limited applicability to specific resources

Linux Control Groups (cgroups)

Control Groups (cgroups) Linux kernel feature for organizing tasks into hierarchical groups:

- Allows processes to be organized and controlled collectively
- Strict enforcement of access and usage criteria
- Applies to sets of processes & all future children
- Controls various types of resources (CPU, memory, I/O, etc.)

cgroups Terminology Key concepts in cgroups:

- **cgroupp**: Collection of processes bound to limits/parameters
- **Subsystem** (Controller): Kernel component related to a resource type
- **Hierarchy**: Arrangement of cgroups in a tree structure
- **Resource Controller**: Implements behavior for specific resource type

Various controllers have been implemented:

- CPU controller: Limits CPU time
- Memory controller: Limits memory usage
- I/O controller: Controls disk I/O
- Network controller: Manages network bandwidth
- Devices controller: Controls device access

cgroups Hierarchy

cgroups are organized in a hierarchical structure:

- Created by making subdirectories in the cgroup filesystem
- Limits defined at each level of the hierarchy
- Limits apply throughout the sub-hierarchy
- Descendant cgroups cannot exceed limits of ancestor cgroups

cgroups Implementation Approach

cgroups uses a filesystem-based approach:

- Communicates with Linux kernel via filesystem
- Virtual filesystem, stored in RAM
- Provides structured, standardized operations via I/O system calls
- Similar approach to /proc filesystem
- Implementation steps:
 - Create tmpfs mount
 - Create directory
 - Mount resource control interfaces as files in directory
 - Configure controls by editing files
 - Associate processes (PIDs) with control configuration

cgroups Controllers

CPU Controller manages CPU usage:

- Controls upper and lower limits of CPU shares
- Lower limit guarantees minimum CPU time when system is busy
- Upper limit restricts CPU time in each scheduling period
- Does not limit CPU usage if CPUs are not busy

Cpuset Controller manages CPU assignment:

- Binds processes to specific CPUs
- Allows process isolation on multi-core systems
- Can be used to improve cache utilization

Memory Controller manages memory usage:

- Reports and limits process memory, kernel memory, and swap usage
- Sets hard and soft limits on memory consumption
- Can enforce OOM (Out Of Memory) killing within cgroups

Blkio Controller manages block device I/O:

- Limits access to block devices through throttling
- Sets upper I/O rate limits on devices
- Implements proportional-weight time-based division of disk I/O
- Can limit both read and write operations

CPU Controller Example Limiting CPU usage for a group of processes:

```
1 # Create a cgroup for CPU control
2 mkdir /sys/fs/cgroup/cpu/limited_group
3 # Set maximum CPU usage to 10% (100ms in a 1000ms period)
4 echo 100000 > /sys/fs/cgroup/cpu/limited_group/cpu.cfs_period_us
5 echo 10000 > /sys/fs/cgroup/cpu/limited_group/cpu.cfs_quota_us
6 # Add a process to the cgroup
7 echo $PID > /sys/fs/cgroup/cpu/limited_group/cgroup.procs
8 # Run a CPU-intensive task and observe it being limited
9 # Even on an idle system, it won't exceed 10% of one CPU
```

Device Controller Example Restricting access to a device:

```
1 # Create a cgroup for device control
2 mkdir /sys/fs/cgroup/devices/group0
3 # By default, full permissions exist
4 cat /sys/fs/cgroup/devices/group0/devices.list
5 # Output: a *:rwm (all devices, all permissions)
6
7 # Deny access to /dev/null (major 1, minor 3)
8 echo 'c 1:3 rwm' > /sys/fs/cgroup/devices/group0/devices.deny
9 # Add the current shell to the group
10 echo $$ > /sys/fs/cgroup/devices/group0/tasks
11 # Try to use /dev/null - it will fail
12 echo "test" > /dev/null
13 # Output: bash: /dev/null: Operation not permitted
```

cgroups Versions

cgroups v1 vs v2 Linux has two implementations of cgroups:

cgroups v1:

- Initial release in Linux 2.6.24
- Rapid adoption but uncoordinated development
- Some inconsistencies between controllers

cgroups v2:

- Added in Linux 4.5
- Intended to replace v1 (but v1 continues to exist for compatibility)
- Currently implements a subset of v1 controllers
- Both versions can coexist, but a controller can't be used in both simultaneously

cgroups v1 two approaches to resource control:

- **Individual Resource Control:**
 - Each controller mounted against a separate filesystem
 - One controller associated with one hierarchy
- **Collective Resource Control:**
 - Multiple controllers co-mounted against a single filesystem
 - Co-mounted controllers manage the same hierarchy

Each cgroup is represented by a directory:

- Child cgroups represented as subdirectories
- Configuration files under each directory
- Files reflect resource limits and properties

cgroups v2 key differences from v1:

- All mounted controllers reside in a single unified hierarchy
- Simplified controller set
 - io (replaces blkio)
 - memory
 - pids
 - perf_event
 - rdma
 - cpu
 - freezer
- Supports delegation (non-privileged users can manage subtrees)
- Supports thread mode (thread-level control)

Tasks vs Processes in cgroups v1

cgroups v1 distinguishes between tasks and processes:

- Process can consist of multiple threads (tasks)
- cgroups v1 allows independent manipulation of thread cgroup membership
- This can cause problems for controllers like memory (all threads share an address space)
- This ability has been limited in cgroups v2

Working with cgroups

Working with cgroups

Exploring cgroups

- List available subsystems: `cat /proc/cgroups`
- View cgroup hierarchy: `tree /sys/fs/cgroup`
- Check cgroups created by systemd: `systemd-cgls`
- Monitor cgroup usage: `systemd-cgtop`

Creating and managing cgroups

- Create a cgroup: `mkdir /sys/fs/cgroup/[controller]/[name]`
- Add process to cgroup: `echo [PID] > /sys/fs/cgroup/[controller]/[name]/cgroup.procs`
- Set CPU limit: `echo [value] > /sys/fs/cgroup/cpu/[name]/cpu.cfs_quota_us`
- Set memory limit: `echo [value] > /sys/fs/cgroup/memory/[name]/memory.limit_in_bytes`
- Remove cgroup: `rmdir /sys/fs/cgroup/[controller]/[name]`

Working with systemd cgroups

- Create transient cgroup: `systemd-run -unit=[name] -slice=[slice] [command]`
- Set resource limits: `systemctl set-property [unit] [property]=[value]`
- Example: `systemctl set-property user.slice CPUQuota=20%`

Mounting cgroups v1 Mounting cgroups controllers:

```
1 # Create a tmpfs mount for cgroups
2 sudo mount -t tmpfs -o size=10M tmpfs /sys/fs/cgroup
3
4 # Mount a single controller (individual resource control)
5 sudo mount -t cgroup -o cpu none /sys/fs/cgroup/cpu
6
7 # Co-mount multiple controllers (collective resource control)
8 sudo mount -t cgroup -o cpu,cpuacct none /sys/fs/cgroup/cpu,cpuacct
```

It's not possible to mount the same controller against multiple hierarchies.

Creating and Managing cgroups v1 Basic cgroup management:

```
1 # Create a new cgroup
2 mkdir /sys/fs/cgroup/cpu/cg1
3
4 # Add the current process to the cgroup
5 echo $$ > /sys/fs/cgroup/cpu/cg1/cgroup.procs
6
7 # Add a specific process to the cgroup
8 echo <PID> > /sys/fs/cgroup/cpu/cg1/cgroup.procs
9
10 # Remove a cgroup (must be empty of processes and child cgroups)
11 rmdir /sys/fs/cgroup/cpu/cg1
```

When adding a process to a cgroup, all threads in the process are moved together.

Using systemd Resource Control Controlling resource limits with systemd:

```
1 # View current system slices
2 systemctl list-units --type=slice
3
4 # Check user slice settings
5 systemctl show user.slice
6
7 # Limit CPU usage for all user processes to 20%
8 sudo systemctl set-property user.slice CPUQuota=20%
9
10 # Create a resource-limited service
11 cat << EOF > /etc/systemd/system/limited-service.service
12 [Unit]
13 Description=Resource Limited Service
14
15 [Service]
16 ExecStart=/usr/bin/sha1sum /dev/zero
17 CPUQuota=10%
18 MemoryLimit=100M
19
20 [Install]
21 WantedBy=multi-user.target
22 EOF
23
24 # Reload systemd and start the service
25 sudo systemctl daemon-reload
26 sudo systemctl start limited-service
27
28 # Monitor resource usage
29 systemd-cgtop
```

Creating a Custom CPU Control Creating a CPU affinity control from scratch:

```
1 # Create a tmpfs mount
2 sudo mkdir -p /mnt/cgroups
3 sudo mount -t tmpfs none /mnt/cgroups
4
5 # Mount cpuset controller
6 sudo mkdir -p /mnt/cgroups/cpuset
7 sudo mount -t cgroup -o cpuset none /mnt/cgroups/cpuset
8
9 # Create a cgroup
10 sudo mkdir /mnt/cgroups/cpuset/group1
11
12 # Configure required memory settings first
13 echo 0 > /mnt/cgroups/cpuset/group1/cpuset.mems
14
15 # Restrict to CPUs 0 and 1 only (assuming 4 CPUs)
16 echo "0-1" > /mnt/cgroups/cpuset/group1/cpuset.cpus
17
18 # Run CPU-intensive processes
19 for i in {1..4}; do
20     # Create processes
21     sha1sum /dev/zero &
22     # Get PID and add to cgroup
23     PID=$!
24     echo $PID > /mnt/cgroups/cpuset/group1/cgroup.procs
25     echo "Added process $PID to CPU-restricted group"
26 done
27
28 # Check CPU assignment with taskset
29 for pid in $(cat /mnt/cgroups/cpuset/group1/cgroup.procs); do
30     taskset -p $pid
31 done
```

Memory Management

Memory Management Introduction

Critical system resource managed by the OS:

- The Memory Manager controls main memory allocation and usage
- Secondary memory: buffer zone (swap)

Memory is organized in a hierarchy:

- Fast cache in 1-3 layers (L1, L2, L3)
- Main memory (RAM) - slower but larger
- Secondary memory (hard disks, SSDs)
 - for programs and files
- Tertiary memory (backup storage, tapes)

Memory Allocation Approaches

Memory Division and Fragmentation

- **Static memory division:**
 - Memory divided into fixed-size segments
 - Problem: Internal fragmentation (wasted space within allocated segments)
- **Dynamic memory division:**
 - Memory divided according to process needs
 - Problem: External fragmentation (free space becomes fragmented)
 - Solution: Compaction (expensive operation)

Free Space Management

Finding free space during allocation requires efficient algorithms:

- **Bitmap approach:**
 - Space-efficient representation
 - One bit per allocation unit
 - Fast free-space finding
- **Linked list approach:**
 - List of free blocks
 - Supports various placement algorithms (first/next/best fit)

Memory Management Tasks

OSs handle several memory management tasks:

- Determine how much memory processes require
- Deciding where in memory processes are located (position of residency)
- Managing how long processes remain in memory (length of residency)
- Subdividing memory for co-existence of multiple processes
- Handling memory fragmentation

Buddy Algorithm

- Division/Fragmentation:
- Memory divided into blocks of power-of-2 sizes
 - When a request arrives, the system:
 - Finds the smallest block that fits the request
 - If no suitable block exists, splits a larger block into two 'buddies'
 - Allocates one buddy and keeps the other free
 - When a block is freed, the system:
 - Checks if its buddy is also free
 - If both are free, merge into a larger block
 - Continues merging recursively if possible
 - Simpler to implement than other DAC
 - Still experiences some internal fragmentation

Swapping

- Free Space Management:
- Secondary memory: temp. storage for processes
 - When processes are suspended or exit, their memory is freed
 - When processes restart, they are reloaded into memory
 - Allows more processes to run concurrently than physical memory would permit

Virtual Memory

Virtual Memory

Leverages program behavior characteristics:

- Programs exhibit spatial locality (tend to use a limited area of code at any time)
- Entire processes don't need to be fully resident in memory
- Non-required code/data can be swapped out when not immediately needed
- Enables more processes to run concurrently
- Must be transparent to programmer/process

Address Translation

logical \Rightarrow physical
Virtual memory requires translation between logical and physical addresses:

- Logical address consists of page-nr. and -offset
- Page nr. used to look up frame nr. in page table
- Physical address formed by combining frame number with page offset
- Translation process:
 - Extract page nr. and offset from logical addr.
 - Use page nr. to index page table
 - Retrieve frame nr. from page table
 - Combine frame nr. with offset to form physical address

Translation Lookaside Buffer (TLB)

The TLB addresses the performance overhead of page table lookups:

- Cache for recently accessed page table entries
- Small (typically 64 entries)
- Uses content-addressable memory (CAM) for fast lookups
- Memory access process with TLB:
 - Check TLB for page number
 - If found (TLB hit), use frame number directly
 - If not found (TLB miss), search page table
 - If not in page table, trigger page fault
 - Add entry to TLB for future accesses

Paging mechanism that enables virtual memory:

- Process memory is divided into fixed-size pages
- Physical memory is divided into frames of the same size
- Pages are loaded into frames as needed
- Memory Management Unit (MMU) manages mapping between pages and frames
- Typically uses 'on-demand paging' (lazy loading)
 - Only loads pages when they are accessed
 - Sometimes prefetches additional pages based on locality
- Process has set of resident pages in memory
 - Resident set: All process pages in memory
 - Working set: Pages currently being used

Page Replacement

→ when memory is full and a new page is needed

- System must decide which resident page to evict
- Can use global strategy (any process's pages) or local strategy (only faulting process's pages)

Common page replacement algorithms:

- Optimal: Replace page used furthest in future (theoretical only)
- Least Recently Used (LRU):
 - Replace page unused for longest time
- First-In-First-Out (FIFO): Replace oldest page

Page Tables

Maintain mapping between pages and frames:

- Each entry contains a frame number
- Entries also include status bits:
 - Valid bit: Indicates if page holds valid data
 - Present bit: Indicates if page is in memory
 - Modified bit (dirty): Indicates if page has been written to
 - Referenced bit: Indicates recent usage
 - Protection bits: Control read/write/execute permissions
- Page tables are stored in main memory
- Can be very large for large address spaces

Linux Memory Management

Linux Page Table Organization

Linux uses a hierarchical page table structure:

- Multi-level page directory to reduce size and improve lookup speed
- Typically 4-level structure:
 - Page Global Directory (PGD)
 - Page Upper Directory (PUD)
 - Page Middle Directory (PMD)
 - Page Table Entry (PTE)
- Allows efficient handling of sparse address spaces
- Only allocates page tables for used parts of address space

Huge Pages supported to improve performance:

- Standard page size is 4KB
- Huge pages can be 2MB or 1GB (architecture-dependent)
- Advantages:
 - Reduces TLB pressure (fewer entries needed to cover same memory)
 - Improves performance for memory-intensive applications
 - More efficient for large memory allocations
- Uses higher-level page table entries (PUD/PMD)
- Requires explicit configuration

Memory Zones

Linux divides physical memory into zones to handle hardware limitations:

- **ZONE_DMA**: Memory addressable by DMA controllers (typically below 16MB)
- **ZONE_NORMAL**: Regularly mapped memory in kernel space
- **ZONE_HIGHMEM**: Memory beyond what the kernel can directly address
- Zones are managed separately to accommodate different hardware constraints
- Each zone has its own free lists and allocation policies

Buddy Allocator Linux uses the buddy system for frame allocation:

- Fundamental allocation unit is the page frame
- Maintains lists of free blocks of various sizes (powers of 2)
- When a process requests memory:
 - System finds the smallest block size that fits the request
 - If necessary, splits larger blocks into "buddies"
 - Allocates memory from appropriate free list
- When memory is freed:
 - System checks if buddy is also free
 - If so, merges buddies to form larger block
 - Continues merging recursively if possible
- Maintains free lists up to MAX_ORDER-1 (typically 10, so up to 512 contiguous pages)

Slab Allocator Linux uses the slab allocator for kernel objects:

- Kernel often requires small allocations for data structures
- Pages (4KB) are too large for many kernel objects
- Slab allocator:
 - Gets pages from buddy allocator
 - Divides them into smaller objects of specific types
 - Maintains caches of frequently used object types
 - Reuses recently freed objects (helps prevent fragmentation)
 - Preserves object state between uses
- Improves memory utilization and allocation speed
- Minimizes internal fragmentation

Memory Compaction Linux performs memory compaction to address fragmentation:

- Problem: Buddy allocator may not find large contiguous blocks
- Solution: kcompactd daemon performs compaction
- Process:
 - Balances memory zones by swapping out non-working-set pages
 - Moves movable pages toward the top of the memory zone
 - Leaves bottom of memory free for new allocations
 - Creates larger contiguous free blocks
- Performed on-demand or periodically
- Enables allocation of huge pages and other large memory blocks

Shared Libraries Linux uses shared libraries to reduce memory usage:

- Multiple processes can use the same library code
- Libraries compiled with -fPIC (Position Independent Code)
- Dynamically linked with processes using ld.so
- Read-only code pages memory-mapped into processes
- Benefits:
 - Reduces memory footprint
 - Shared code/text pages only loaded once
 - Only data pages need to be process-specific
- Challenge: Version compatibility ("DLL hell")

Page Reclamation Linux uses page reclamation to recover memory:

- Working set: Pages actively in use by processes
- Resident set: All pages in memory
- A page is in the working set if:
 - Accessed via process address space
 - Accessed via system call
 - Accessed via device driver
- Linux identifies non-working set pages using a bitmap
- Pages marked idle can be reclaimed when memory is needed

Least Recently Used in Linux

Linux implements a two-stage LRU algorithm:

- Maintains two lists of page frames:
 - Active list: Recently accessed pages
 - Inactive list: Less recently accessed pages
- Pages move between lists based on access patterns
- Inactive pages are candidates for reclamation
- Recently accessed inactive pages can be promoted to active list
- Linux uses a global strategy for page reclamation
- Recent development: Multi-Generational LRU (MGLRU)
 - Assigns generation numbers to page frames based on recent access
 - Older generations are reclaimed first
 - Improves performance and responsiveness

Out-of-Memory (OOM) Killer Linux has an OOM Killer to handle critical memory shortages:

- Activates when system is critically low on memory
- Linux tends to be optimistic in memory allocation
 - Processes typically request more memory than needed
 - System may over-allocate (more than physical memory)
- OOM Killer selects processes to terminate based on heuristics
 - Considers memory usage, runtime, nice value, etc.
 - Each process has an oom_score in /proc/\$PID/oom_score
 - Can be adjusted through oom_score_adj
- Prioritizes system stability over individual process survival

Memory Management Analysis in Linux

Basic memory information

- Display system memory usage: free -h
- View memory details: cat /proc/meminfo
- Check process memory usage: ps -eo pid,ppid,cmd,vsz,rss
- Interactive memory monitor: top or htop

Process memory analysis

- Check process memory maps: cat /proc/\$PID/maps
- View process memory status: cat /proc/\$PID/status
- Analyze memory usage in detail: pmap \$PID
- Track memory over time: smem

Memory page information

- Get page size: getconf PAGE_SIZE
- Check huge pages: cat /proc/meminfo | grep Huge
- View page stats: cat /proc/pagetypeinfo
- Check page faults: ps -o min_flt,maj_flt \$PID

Memory limits and control

- Set memory limits: ulimit -m [size]
- Control cgroup memory: echo [value] > /sys/fs/cgroup/memory/[group]/memory.limit_in_bytes
- Check swappiness: cat /proc/sys/vm/swappiness
- Adjust swappiness: sysctl vm.swappiness=[value]

Analyzing Process Memory Check memory usage details for a process:

```
1 # Get PID of a process
2 PID=$(pidof firefox)
3
4 # Check virtual and resident memory size
5 ps -o pid,comm,vsz,rss -p $PID
6
7 # Analyze memory map segments
8 cat /proc/$PID/maps | head -10
9
10 # View detailed memory status
11 grep -E 'VmSize|VmRSS|VmData|VmStk|VmExe' /proc/$PID/status
12
13 # Map process address space in detail
14 pmap -x $PID | head -20
15
16 # Check page faults
17 ps -o min_flt,maj_flt -p $PID
```

Explanation of memory terms:

- VSZ (Virtual Size): Total virtual memory allocated to process
- RSS (Resident Set Size): Actual physical memory used
- VmData: Size of data segment
- VmStk: Size of stack
- VmExe: Size of text segment
- min_flt: Minor page faults (page in memory but not in process's page table)
- maj_flt: Major page faults (page had to be loaded from disk)

Working with Page Size and Memory Allocation Analyzing page size and memory allocation:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/mman.h>
5
6 int main() {
7     // Get process ID
8     pid_t pid = getpid();
9     printf("Process ID: %d\n", pid);
10
11     // Get page size
12     int page_size = getpagesize();
13     printf("Page size: %d bytes\n", page_size);
14
15     // Allocate memory for 10 pages
16     size_t size = 10 * page_size;
17     char *buffer = malloc(size);
18     printf("Allocated %zu bytes (%zu pages)\n",
19           size, size / page_size);
20
21     // Check page faults before access
22     printf("Check page faults with: ps -o min_flt,maj_flt %d\n", pid);
23     printf("Press Enter to continue...\n");
24     getchar();
25
26     // Access the memory (causes page faults)
27     for (int i = 0; i < size; i += page_size) {
28         buffer[i] = 1; // Touch one byte per page
29     }
30
31     // Check page faults after access
32     printf("Check page faults again with: ps -o min_flt,maj_flt %d\n", pid);
33     printf("Press Enter to continue...\n");
34     getchar();
35
36     // Allocate page-aligned memory
37     void *aligned_buf = NULL;
38     int result = posix_memalign(&aligned_buf, page_size, size);
39     if (result == 0) {
40         printf("Allocated %zu bytes aligned on page boundary\n", size);
41     }
42
43     // Free memory
44     free(buffer);
45     free(aligned_buf);
46
47     return 0;
48 }
```

This example demonstrates:

- Getting the system page size
- Allocating memory
- Observing page faults due to lazy allocation
- Creating page-aligned memory allocations

I/O Challenges I/O management presents unique challenges for operating systems:

- Huge variety of I/O devices (keyboards, mice, drives, sensors, etc.)
- Diverse interfaces (ATA, SATA, USB, PCI, etc.)
- Wide range of speeds and capacities
- Different data transfer characteristics
- Need for a unified approach to device interaction

The operating system must abstract this heterogeneity by providing consistent interfaces for applications to access diverse devices.

Device Categories From a user perspective, devices fall into two main categories:

Block Devices:

- Operate on fixed-size blocks of data
- Support random access to any block
- Example: Hard drives, SSDs
- Block operations are independent from each other
- Block size defined when formatting (logical)
- Sector size is the physical unit on the device

Character Devices:

- Operate on streams of characters
- Generally sequential access
- Example: Keyboards, mice, printers
- Characters are interpretations of bit patterns according to specifications (ASCII, Unicode)
- Character devices ultimately operate on bit-level too

I/O Hardware Architecture

I/O Hardware Components

Key components in I/O hardware architecture:

- **I/O Controller:** Electronic interface to the device
 - Typically one per device category (SCSI, IDE, USB, Ethernet)
 - Contains registers that control device operation
 - Maintains buffers for read/write operations
 - Communicates with CPU and main memory
- **I/O Ports:** Addresses that point to controller registers
- **Buffers:** Memory areas for data transfer
- **Bus System:** Connects CPU, memory, and I/O devices

I/O Address Space

X86 architecture has two approaches for I/O address allocation:

- **Port Mapped I/O (PMIO):**
 - Peripheral I/O registers assigned to a dedicated address range
 - Distinct from system memory
 - Uses dedicated I/O instructions (IN and OUT)
 - Limited to 64K ports (16-bit addressing)
 - Common for older peripherals (e.g., ISA cards)
 - Listed in `/proc/ioports`
- **Memory Mapped I/O (MMIO):**
 - Peripheral registers mapped into main memory address space
 - Uses regular memory instructions (e.g., MOV)
 - Default in modern systems (e.g., PCIe, PCI)
 - Listed in `/proc/iomem`

Direct Memory Access (DMA)

DMA improves I/O performance:

- Allows devices to transfer data directly to/from memory
- Bypasses CPU for bulk data transfer
- CPU sets up transfer parameters:
 - Memory address
 - Count of bytes to transfer
 - Direction (read/write)
 - Device-specific parameters
- DMA controller handles the actual transfer
- Interrupts CPU when transfer completes
- Reduces CPU overhead for data-intensive I/O operations

I/O Access Methods

Programmed I/O:

- CPU does all the work
- CPU executes instructions to transfer data
- CPU repeatedly checks device status (polling/busy-waiting)
- Synchronous operation (CPU waits for I/O completion)
- Simple but inefficient for slow devices

Interrupt-driven I/O:

- Devices signal CPU when they need attention
- CPU init. operation then continues other work
- Device interrupts when operation completes
- Asynchronous operation
- More efficient, especially for slow devices

Interrupt Handling

Interrupts are fundamental to efficient I/O:

- **Interrupt:** Event that defers the normal flow of CPU execution
- When an interrupt occurs:
 - Current execution is suspended
 - CPU state is saved
 - Special routine (interrupt handler) is executed
 - After completion, previous execution resumes
- **Interrupt types:**
 - **Synchronous:** Generated by executing an instruction (e.g., divide by zero)
 - **Asynchronous:** Generated by external events (e.g., I/O completion)
- **Interrupt classifications:**
 - **Maskable:** Can be ignored by setting interrupt mask
 - **Non-maskable:** Cannot be ignored (critical events)

Interrupt Flow

Hardware interrupt flow:

- Device raises interrupt on its IRQ line
 - Programmable Interrupt Controller (PIC) converts IRQ to vector number
 - PIC signals CPU via INTR pin
 - CPU acknowledges interrupt
 - CPU executes the appropriate interrupt handler
- In Linux, interrupt handling occurs in three phases:
- **Critical:** Minimal processing, acknowledge interrupt
 - **Immediate:** Essential processing, can't be deferred
 - **Deferred:** Non-critical processing, scheduled for later

I/O Access Patterns

Different access patterns affect I/O performance:

Exclusive vs. Shared Access:

- **Exclusive:** Device dedicated to one process
- **Shared:** Multiple processes access same device
- Shared access requires scheduling (e.g., disk I/O scheduling)

Sequential vs. Random Access:

- **Sequential:** Data accessed in order (e.g., tape)
- **Random:** Data accessed in any order (e.g., disk)

Blocking vs. Non-Blocking:

- **Blocking:** Process waits until I/O completes
- **Non-Blocking:** Process continues, checks completion later

Synchronous vs. Asynchronous:

- **Synchronous:** Process execution synchronized with I/O completion
- **Asynchronous:** Process continues, notified of I/O completion

Buffered vs. Direct I/O

Buffering improves I/O performance:

- **Buffered I/O:**
 - Data passes through intermediate buffer
 - Decouples data access from data generation
 - Handles different speeds between devices
 - Enables rate control
 - Allows data manipulation before final transfer
 - Supports data verification
- **Direct I/O:**
 - Data transferred directly to/from device
 - Bypasses system caches
 - Reduces memory usage and CPU overhead
 - Useful for applications with their own caching
 - May be slower for some workloads

Error Handling

I/O systems must handle errors effectively:

- Errors can occur at various levels:
 - Bit-level, byte-level, packet-level, block-level
 - Hardware vs. software detection
 - User space vs. kernel space handling
- Error handling approaches:
 - **Error Detection:** Identify errors (e.g., checksums, parity)
 - **Error Correction:** Fix errors without retransmission
 - **Error Recovery:** Return to consistent state after error
- Trade-offs between performance and reliability

Linux Device Model

The Linux Device Model maintains the state and structure of the system:

- Maintains information about devices, drivers, buses, etc.
- Key entities:
 - **Device**: Physical device attached to a bus
 - **Driver**: Software entity that operates devices
 - **Bus**: Device to which other devices can be attached
 - **Class**: Type of device with similar behavior
 - **Subsystem**: View on the system structure
- Represented in user space via sysfs (mounted at /sys)

sysfs

virtual filesystem that exposes the Linux Device Model:

- Located at /sys
- Key directories:
 - /sys/block: Block devices
 - /sys/bus: Bus types
 - /sys/class: Device classes
 - /sys/devices: Hierarchical device structure
 - /sys/firmware: Firmware information
 - /sys/fs: Filesystem information
 - /sys/kernel: Kernel status
 - /sys/module: Loaded modules
 - /sys/power: Power management
- Contains attributes in files for configuration and status

/dev Directory The /dev directory contains special files representing devices:

- Each file represents an I/O device
- Allows standard file operations (open, read, write, close) on devices
- When accessed, kernel routes operations to appropriate device drivers
- Types of device files:
 - Character device files: For character devices
 - Block device files: For block devices
- Naming conventions:
 - /dev/sdX: SCSI/SATA disk devices
 - /dev/ttyX: Terminal devices
 - /dev/nullX: Special device files

Device Access in Linux

Linux provides a unified interface for device access:

- Applications use standard file operations:
 - open(): Open device
 - read(): Read from device
 - write(): Write to device
 - ioctl(): Device-specific operations
 - close(): Close device
- Kernel translates these operations to device-specific commands
- Virtual File System (VFS) provides abstraction layer
- Device drivers implement the specific operations

udev

userspace device manager for Linux:

- Part of systemd
- systemd-udevd listens to kernel events
- Executes rules based on device information from sysfs
- Creates or removes device nodes in /dev
- Provides consistent device naming
- Enables automatic device setup
- Supports user-defined rules

Working with I/O in Linux

Exploring device information

- List I/O ports: cat /proc/ioports
- List I/O memory: cat /proc/iomem
- View interrupts: cat /proc/interrupts
- List block devices: lsblk
- Show hardware: lshw
- Display PCI devices: lspci
- List USB devices: lsusb

Working with devices

- Check device information: udevadm info --query=all --name=/dev/sda
- Monitor device events: udevadm monitor
- Show device properties: udevadm info --attribute-walk --name=/dev/sda
- Check I/O performance: iostat -x
- Monitor I/O activity: iotop

Device file operations

- Create device file: mknod /dev/example c 1 3
- Read from device: cat /dev/input/mouse0 | hexdump
- Write to device: echo "test" >/dev/tty1
- Control device: ioctl system call in C programs

I/O Performance Testing Testing disk write performance with different options:

```
1 # Basic write test (with caching)
2 dd if=/dev/zero of=speedtest bs=10M count=100
3 rm speedtest
4 # Write test with synchronous I/O (forces data to disk)
5 dd if=/dev/zero of=speedtest bs=10M count=100 conv=fdatasync
6 rm speedtest
7 # Write test with direct I/O (bypasses the page cache)
8 dd if=/dev/zero of=speedtest bs=10M count=100 oflag=direct
9 rm speedtest
10 # Monitor disk I/O activity during test
11 iostat -x 1
12 # Check disk utilization statistics
13 iostat -p sda
14
15 # Explanation:
16 # - Standard dd shows high performance but may not be on disk yet
17 # - fdatasync ensures data is physically written to disk
18 # - direct bypasses the OS cache, showing raw device performance
```

Device Information Analysis Exploring device information in Linux:

```
1 # List block devices with details
2 lsblk -f
3 # Get detailed information about a specific device
4 udevadm info --query=all --name=/dev/sda1
5 # Examine sysfs entries for the device
6 ls -l /sys/block/sda/sda1/
7 # Check device attributes
8 cat /sys/block/sda/queue/scheduler
9 cat /sys/block/sda/queue/read_ahead_kb
10 # Get PCI information for a disk controller
11 lspci | grep -i sata
12 # Check all properties of a device
13 udevadm info --attribute-walk --name=/dev/sda1 | less
14 # Monitor udev events when plugging in a USB device
15 udevadm monitor --property
16 # (Now plug in a USB device to see events)
```

Building and Using a Custom Linux Kernel

- Custom Kernel Motivation** There are various reasons to build a custom kernel:
- Create a minimalist kernel (disable unused features, load needed ones as modules)
 - Add custom OS features for specific requirements
 - Support special hardware that may not be in the standard kernel
 - Optimize for specific workloads or hardware platforms
 - Learn about kernel internals and development processes

- Linux Kernel Development Process**
- New major kernel releases every 2-3 months
 - Development cycle phases:
 - Merge window (first two weeks): New features merged into mainline
 - Release candidates (weekly): Bug fixes only, no new features
 - Final release: After several release candidates
 - Long-term support (LTS) kernels receive updates for extended periods
 - Community development model with maintainers for various subsystems

Building a Custom Linux Kernel

- Getting the source code
- Download from kernel.org: `wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.xx.tar.xz`
 - Extract: `tar xf linux-5.xx.tar.xz`
 - Change directory: `cd linux-5.xx`
- Installing build dependencies
- On Debian/Ubuntu: `sudo apt-get install build-essential gcc bc bison flex libssl-dev libncurses5-dev libelf-dev`
- Configuring the kernel
- Start from existing config (recommended):
 - Current running kernel: `cp /boot/config-$(uname -r) ./config`
 - Distribution default: `make defconfig`
 - Update config for new options: `make oldconfig`
 - Interactive configuration tools:
 - Text-based menu: `make menuconfig`
 - GUI-based: `make xconfig` or `make gconfig`
 - Important configuration options:
 - Custom version name: `CONFIG_LOCALVERSION`
 - Module support: `CONFIG_MODULES`
 - CPU and architecture options, Device drivers & File systems

- Building the kernel
- Compile: `make -j$(nproc)`
 - Build modules: `make modules`
 - Create Debian/Ubuntu packages: `make -j$(nproc) deb-pkg`
 - For RPM-based systems: `make -j$(nproc) rpm-pkg`

- Installing the kernel
- Debian packages: `sudo dpkg -i ../linux-*.deb`
 - RPM packages: `sudo rpm -ivh ../linux-*.rpm`
 - Manual installation:
 - Install modules: `sudo make modules_install`
 - Install kernel: `sudo make install`
 - Update bootloader: `sudo update-grub` (GRUB)

- Booting the new kernel
- Reboot: `sudo reboot`
 - Select the new kernel at the bootloader menu and verify running kernel: `uname -a`

- Choosing a Kernel Version** version matters!
- Mainline: Latest development version (might be unstable)
 - Stable: Recent release with proven stability
 - Long-term: Longer support period (good for production systems)
 - Distribution-specific: Modified by Linux distributions
- Version numbers indicate:
- First number: Major version (rarely changes)
 - Second number: Minor version (major features)
 - Third number: Patch level (bug fixes etc.)

Linux Kernel Modules

- Kernel Modules Concept** Kernel modules extend kernel functionality without rebuilding:
- Loadable code that can be added to or removed from a running kernel
 - Allow dynamic extension of kernel capabilities
 - Provide device drivers, filesystem drivers, system calls, etc.
 - Reduce the size of the base kernel
 - Enable support for hardware that is hot-pluggable
 - Support for special features used in specific applications

- Module Structure** A kernel module follows a specific structure:
- Must include necessary kernel headers
 - Initialization function (`init_module()` or custom named)
 - Called when the module is loaded
 - Sets up resources, registers with kernel subsystems
 - Returns success (0) or error code
 - Cleanup function (`cleanup_module()` or custom named)
 - Called when the module is unloaded
 - Releases resources, unregisters from kernel subsystems
 - Uses `MODULE_LICENSE()` macro to specify license
 - Can include other macros for author, description, etc.

Hello World Kernel Module Minimal kernel module example:

```
1 #include <linux/module.h>      /* Needed by all modules */
2 #include <linux/kernel.h>      /* Needed for KERN_INFO */
3 #include <linux/init.h>        /* Needed for macros */
4
5 MODULE_LICENSE("GPL");
6 MODULE_AUTHOR("Your Name");
7 MODULE_DESCRIPTION("A simple Hello World module");
8
9 static int __init hello_init(void)
10 {
11     printk(KERN_INFO "Hello, World!\n");
12     return 0;    /* Success */
13 }
14
15 static void __exit hello_exit(void)
16 {
17     printk(KERN_INFO "Goodbye, World!\n");
18 }
19
20 module_init(hello_init);
21 module_exit(hello_exit);
```

- Key components:
- `module_init` and `module_exit` macros register functions
 - `printk` is the kernel's version of `printf`
 - `KERN_INFO` sets the message priority level
 - `MODULE_LICENSE` declares the license (important for symbol exports)

- Module Building Process** Building a kernel module requires:
- Pre-built kernel with module support
 - Module source code
 - Makefile defining the build process
 - Build tools and headers

- The build process:
- Kbuild system builds `<module_name>.o` from source
 - Links to create `<module_name>.ko` (kernel object)
 - Kernel modules must be built against the same kernel version they will run on
 - Distribution-specific kernel headers needed for module compatibility

Module Makefile

Example Makefile for a kernel module:

```
1 # Define the module name
2 obj-m := hello.o
3
4 # For standalone module build
5 all:
6     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
7
8 # Clean up
9 clean:
10    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

For multiple source files:

```
1 # Module with multiple source files
2 hello-y := hello_main.o hello_func.o
3 obj-m := hello.o
4
5 # For standalone module build
6 all:
7     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
8
9 # Clean up
10 clean:
11    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Working with Kernel Modules

Building a module

- Create module source file
- Create Makefile
- Build module: make
- Result: <module_name>.ko file

Loading and unloading modules

- List loaded modules: lsmod
- Insert module: sudo insmod <module_name>.ko
- Remove module: sudo rmmod <module_name>
- Load module with dependencies: sudo modprobe <module_name>
- Unload module and dependencies: sudo modprobe -r <module_name>
- View kernel messages: dmesg

Module information

- Show module info: modinfo <module_name>.ko
- Check if module is loaded: lsmod | grep <module_name>
- Display module parameters: systool -vm <module_name>
- View module details in sysfs: ls -l /sys/module/<module_name>/

Module autoloading

- Install module: sudo make modules_install
- Update module dependencies: sudo depmod -a
- Configure autoloading: echo <module_name> sudo tee /etc/modules-load.d/<module_name>.conf

Character Device Drivers

Character Device Drivers

Character device drivers are a common type of Linux device driver:

- Handle devices that transfer data as a stream of bytes
- Support operations like read, write, open, release, ioctl
- Examples: serial ports, keyboards, mice, sensors
- Appear as files in /dev with major and minor numbers
- Major number identifies the driver
- Minor number identifies the specific device

Basic Character Device Driver

Structure of a simple character device driver:

```
1 #include <linux/kernel.h>
2 #include <linux/module.h>
3 #include <linux/fs.h>
4 #include <linux/uaccess.h>
5 MODULE_LICENSE("GPL");
6 /* Prototypes */
7 static int device_open(struct inode *, struct file *);
8 static int device_release(struct inode *, struct file *);
9 static ssize_t device_read(struct file *, char *, size_t, loff_t *);
10 static ssize_t device_write(struct file *, const char *, size_t, loff_t *);
11 #define DEVICE_NAME "chardev"
12 #define BUF_LEN 80
13 /* Global variables */
14 static int Major; /* Major number assigned */
15 static int Device_Open = 0; /* Is device open? */
16 static char msg[BUF_LEN]; /* Message for the device */
17 static char *msg_Ptr;
18
19 static struct file_operations fops = {
20     .read = device_read,
21     .write = device_write,
22     .open = device_open,
23     .release = device_release
24 };
25 int init_module(void) { /* Initialization function */
26     Major = register_chrdev(0, DEVICE_NAME, &fops);
27     if (Major < 0) {
28         printk(KERN_ALERT "Failed to register with %d\n", Major);
29         return Major;
30     }
31     printk(KERN_INFO "Major number assigned: %d\n", Major);
32     printk(KERN_INFO "Create device: 'mknod /dev/%s c %d 0'\n", DEVICE_NAME, Major);
33     return 0;
34 }
35 void cleanup_module(void) { /* Cleanup function */
36     unregister_chrdev(Major, DEVICE_NAME);
37 }
38 /* Device methods */
39 static int device_open(struct inode *inode, struct file *file) {
40     static int counter = 0;
41     if (Device_Open) return -EBUSY;
42     Device_Open++;
43     sprintf(msg, "Called device_open %d times\n", counter++);
44     msg_Ptr = msg;
45     try_module_get(THIS_MODULE);
46     return 0;
47 }
48 static int device_release(struct inode *inode, struct file *file) {
49     Device_Open--;
50     module_put(THIS_MODULE);
51     return 0;
52 }
53 static ssize_t device_read(struct file *filp, char *buffer, size_t length, loff_t
54     *offset) {
55     int bytes_read = 0;
56     if (*msg_Ptr == 0) return 0; /* End of message */
57     while (length && *msg_Ptr) { /* Transfer data to user space */
58         put_user>(*msg_Ptr++, buffer++);
59         length--;
60         bytes_read++;
61     }
62     return bytes_read;
63 }
64 static ssize_t device_write(struct file *filp, const char *buff, size_t len, loff_t
65     *off) {
66     printk(KERN_ALERT "Operation not supported\n");
67     return -EINVAL;
68 }
```


Character Device Driver Key Components

- `file_operations` structure defines callbacks for file operations
- `register_chrdev` allocates a major number
- Device methods handle open, read, write, and close operations
- `put_user` safely copies data from kernel to user space
- Reference counting with `try_module_get` and `module_put`

Building Installing and Using a Character Device Module Complete workflow for a character device driver:

```
1 # 1. Create source file (chardev.c) and Makefile as shown above
2
3 # 2. Build the module
4 make
5
6 # 3. Load the module
7 sudo insmod chardev.ko
8
9 # 4. Check kernel messages for major number
10 dmesg | tail
11
12 # 5. Create device node (assuming major number is 250)
13 sudo mknod /dev/chardev c 250 0
14 sudo chmod 666 /dev/chardev
15
16 # 6. Test reading from the device
17 cat /dev/chardev
18
19 # 7. Check module information in sysfs
20 ls -l /sys/module/chardev/
21 cat /sys/module/chardev/parameters/* 2>/dev/null
22
23 # 8. Unload the module when done
24 sudo rmmod chardev
25
26 # 9. Clean up the device node
27 sudo rm /dev/chardev
```

This demonstrates:

- Building and loading a custom character device driver
- Creating a device node with appropriate permissions
- Interacting with the device through standard file operations
- Examining module information through `sysfs`
- Proper cleanup when the module is no longer needed

Linux Kernel Module Debugging

Using `printk`

- Add debug messages: `printk(KERN_DEBUG "Debug: %d \n", value);`
- Set console log level: `echo 7 > /proc/sys/kernel/printk`
- View kernel messages: `dmesg`
- Follow kernel messages: `dmesg -w`
- Clear buffer: `dmesg -c`

Debug filesystem

- Mount debugfs: `mount -t debugfs none /sys/kernel/debug`
- Create debug files in your module:
 - Include `<linux/debugfs.h>`
 - Create directory: `debugfs_create_dir()`
 - Create files: `debugfs_create_file()`
- Access from user space: `cat /sys/kernel/debug/mymodule/myfile`

Module parameters

- Define parameters: `module_param(name, type, permissions);`
- Load with parameters: `insmod mymodule.ko debug=1`
- Change at runtime: `echo 1 > /sys/module/mymodule/parameters/debug`

Kernel/system crashes

- Configure `kdump`: Install and configure `kdump-tools`
- Analyze crash dumps with `crash` utility
- Check system logs after reboot: `journalctl -b -1`

I/O Performance Testing with Custom I/O Scheduler Testing I/O performance with different schedulers:

```
1 # 1. Check available I/O schedulers
2 cat /sys/block/sda/queue/scheduler
3
4 # 2. Test performance with default scheduler
5 echo 3 > /proc/sys/vm/drop_caches # Clear cache
6 dd if=/dev/zero of=testfile bs=1M count=1000 oflag=direct
7 rm testfile
8
9 # 3. Change to a different scheduler (e.g., BFQ)
10 echo bfq > /sys/block/sda/queue/scheduler
11
12 # 4. Test performance with new scheduler
13 echo 3 > /proc/sys/vm/drop_caches # Clear cache
14 dd if=/dev/zero of=testfile bs=1M count=1000 oflag=direct
15 rm testfile
16
17 # 5. Test with different I/O priorities
18 # Start a background write process
19 dd if=/dev/zero of=bg_file bs=1M count=2000 oflag=direct &
20 BG_PID=$!
21
22 # Run a foreground process with higher priority
23 ionice -c2 -n0 dd if=/dev/zero of=fg_file bs=1M count=500 oflag=direct
24
25 # Clean up
26 kill $BG_PID
27 rm bg_file fg_file
28
29 # 6. Return to the default scheduler
30 echo deadline > /sys/block/sda/queue/scheduler
```

Introduction to File Systems

File System Requirements

File systems address three essential requirements for long-term information storage:

- It must be possible to store very large amounts of information
- Information must survive termination of processes using it
- Multiple processes must be able to access information concurrently

File systems organize, store, access, and manage persistent data by:

- Supporting multiple storage media
- Providing fast data location and protection mechanisms
- Enabling efficient read and write operations on fixed-size blocks

File System Architecture

File systems consist of two main organizational levels:

- **Logical organization** (user perspective):
 - Hierarchical directory structure
 - File abstractions with names and attributes
 - Access control and permissions
- **Physical organization** (OS perspective):
 - Block allocation and management
 - Data structure layout on storage media
 - Device driver interfaces

Files - User Perspective

File Concept and Structure

A file is an abstraction with multiple perspectives:

- **Logical perspective:** Abstract object with a name and access methods for storing data
- **Physical perspective:** Set of equivalent records or bytes on persistent media
- Files are created and accessed through processes
- Files provide persistent storage beyond process lifetime

File naming conventions:

- File names are strings with system-specific rules
- Many systems support multi-part names separated by periods
- File extension typically follows the last period
- Extensions often indicate file type or intended application

File Types

Operating systems recognize several types of files:

- **Ordinary/Regular Files:** User files containing programs and data
- **Directory Files:** System files containing filesystem structure information
- **Character Special Files:** Represent character-oriented I/O devices
- **Block Special Files:** Represent block-oriented I/O devices
- **Link Files:** References to other files in the filesystem
- **Socket Files:** Communication endpoints for network or IPC
- **Named Pipe Files:** Communication channels between processes

File Access Methods

Files can be accessed in different ways:

- **Sequential Access:** Data read/written in order from beginning to end
 - Common for streaming data, logs, backups
 - Simple to implement and efficient for linear processing
- **Random Access:** Data can be read/written at any position
 - Essential for databases, binary files, memory-mapped files
 - Requires address calculation and seek operations

File Attributes and Permissions

Files have various attributes and access rights:

- **Basic attributes:**
 - Name: Human-readable identifier
 - Size: Number of bytes in the file
 - Timestamps: Creation, modification, access times
 - Owner: User and group ownership
 - Type: Regular, directory, device, etc.
- **Unix/Linux permissions:**
 - Read (r): Ability to read file contents
 - Write (w): Ability to modify file contents
 - Execute (x): Ability to execute file or search directory
 - Applied to three categories: owner, group, others
- **Links:** Number of hard links pointing to the file

File Operations

Common file operations supported by operating systems:

- **Creation and deletion:** Create new files, delete existing files
- **Opening and closing:** Establish/terminate access to files
- **Reading and writing:** Transfer data to/from files
- **Seeking:** Change current position in file for random access
- **Attribute manipulation:** Get and set file metadata
- **Renaming:** Change file names
- **Linking:** Create references to existing files

Basic file operations in C:

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <unistd.h>
4
5 int main() {
6     int fd;
7     char buffer[100];
8     ssize_t bytes_read;
9
10    // Open file for reading
11    fd = open("example.txt", O_RDONLY);
12    if (fd == -1) {
13        perror("Error opening file");
14        return 1;
15    }
16
17    // Read from file
18    bytes_read = read(fd, buffer, sizeof(buffer) - 1);
19    if (bytes_read > 0) {
20        buffer[bytes_read] = '\0';
21        printf("Read: %s\n", buffer);
22    }
23
24    // Close file
25    close(fd);
26
27    return 0;
28 }
```

Directory Organization

File Organization on Storage

Files are organized on storage devices through:

- **Volumes:** Logical storage units (can span multiple physical devices)
- **Partitions:** Subdivisions of physical storage devices
- **Directories:** Hierarchical organization of files and subdirectories

Two main organizational approaches:

- **Physical volume with partitions:** Single device divided into sections
- **Logical volume with multiple disks:** Multiple devices combined into one logical unit

Directory Systems

Directory systems range from simple to complex:

- **Flat file system:**
 - Only one root directory containing all files
 - Used in embedded devices, simple systems
 - No subdirectories or hierarchical organization
- **Hierarchical file system:**
 - Root directory and subdirectories
 - Allows grouping of related files
 - Enables multiple files with the same name in different directories
 - Supports different user areas and system organization

Directory Navigation

Hierarchical systems use various directory concepts:

- **Root directory:** Top-level directory (/ in Unix/Linux)
- **Home directory:** User's personal directory (~ in Unix/Linux)
- **Working directory:** Current directory (. in Unix/Linux)
- **Parent directory:** Directory containing current directory (.. in Unix/Linux)
- **Absolute path:** Complete path from root (e.g., /usr/local/bin/program)
- **Relative path:** Path from current directory (e.g., ../docs/readme.txt)

Directory Operations

Common directory operations:

- Create and delete directories
- Open and close directories for reading
- Read directory entries
- Change current working directory
- Rename directories
- Create and remove links between directories and files

Shared Files and Linking

File systems support file sharing through linking:

- **Hard links:**
 - Directory entries that point to the same inode
 - Multiple names for the same file
 - Cannot span filesystem boundaries
 - Cannot link to directories (usually)
 - File persists until all hard links are removed
- **Symbolic (soft) links:**
 - Special files containing paths to other files
 - Can span filesystem boundaries
 - Can link to directories
 - Become invalid if target is moved or deleted
 - More flexible but with additional indirection overhead

Working with Files and Directories in Linux

Basic file operations

- Create empty file: `touch filename`
- Copy files: `cp source destination`
- Move/rename files: `mv oldname newname`
- Delete files: `rm filename`
- View file contents: `cat filename, less filename`
- Edit files: `nano filename, vim filename`

Directory operations

- Show current directory: `pwd`
- Change directory: `cd /path/to/directory`
- List directory contents: `ls -la`
- Create directory: `mkdir dirname`
- Remove empty directory: `rmdir dirname`
- Remove directory and contents: `rm -rf dirname`
- Display directory tree: `tree dirname`

File permissions and ownership

- View permissions: `ls -l filename`
- Change permissions: `chmod 755 filename`
- Change ownership: `chown user:group filename`
- Change group: `chgrp group filename`

Linking

- Create hard link: `ln source linkname`
- Create symbolic link: `ln -s source linkname`
- Find hard links: `find /path -samefile filename`
- Show link details: `ls -l linkname`

File System Implementation

File System Layout

File systems are typically organized on storage devices with the following layout:

- **Boot block:** Contains boot code for loading the operating system
- **Superblock:** Contains metadata about the filesystem
 - Filesystem type and version
 - Block size and total number of blocks
 - Number of inodes and free space information
 - Pointers to key data structures
- **Inode table:** Contains file metadata and allocation information
- **Data blocks:** Contain actual file data and directory contents

File Allocation Methods

Contiguous Allocation

Files are stored in contiguous blocks on the storage device:

- **Advantages:**
 - Simple to implement
 - Excellent read performance (no seeking)
 - Minimal metadata required (start block + length)
- **Disadvantages:**
 - External fragmentation over time
 - Difficult to grow files (may require moving entire file)
 - Need to know maximum file size in advance
- **Use cases:** Read-only media, multimedia files

Linked-List Allocation

Files stored as linked lists of disk blocks:

- Each block contains a pointer to the next block
- Directory entry contains pointer to first block
- **Advantages:**
 - No external fragmentation
 - Files can grow dynamically
 - Only need to store address of first block
- **Disadvantages:**
 - Very slow random access (must traverse links)
 - Storage overhead for pointers
 - Vulnerability to pointer corruption

Indexed Allocation (Inodes)

Files use index structures to track data blocks:

- Each file has an index node (inode) containing metadata and block addresses
- Inode structure typically includes:
 - File attributes (size, permissions, timestamps)
 - Direct pointers to data blocks
 - Indirect pointers for large files
 - Double and triple indirect pointers for very large files
- **Advantages:**
 - Fast random access
 - No external fragmentation
 - Efficient for small and large files
- **Disadvantages:**
 - Additional storage overhead for inodes
 - Complex management of indirect blocks

Directory Implementation

Directory Storage

Directories map file names to file location information:

- Directory entries contain:
 - File name
 - Inode number (in Unix-like systems)
 - File type indicator
 - Optional cached attributes
- Two main approaches for storing file attributes:
 - In directory entry (simpler, but larger directories)
 - In separate inode structure (more efficient, used by Unix/Linux)

File Name Length Handling

Different approaches to variable-length file names:

- **Fixed-length names:** Simple but wastes space or limits name length
- **Variable-length inline:** Names stored directly in directory entries
- **Variable-length with pointers:** Directory entries contain pointers to names stored elsewhere
- **Length prefix:** Each name preceded by its length

Free Space Management

Free Block Tracking

File systems must track available storage space:

- **Linked list of free blocks:**
 - Free blocks linked together
 - Simple but requires disk access to find free space
- **Bitmap:**
 - One bit per block (0 = free, 1 = used)
 - Fast to scan and modify
 - Compact representation
- **Grouping:** Free blocks grouped together with one block containing addresses of other free blocks
- **Counting:** Store starting address and count of consecutive free blocks

Block Size Considerations

Choosing the right block size involves trade-offs:

- **Small blocks:**
 - Less internal fragmentation
 - More blocks per file (more metadata overhead)
 - Slower access due to more operations
- **Large blocks:**
 - Higher data transfer rates
 - More internal fragmentation for small files
 - Fewer blocks per file (less metadata overhead)
- Optimal block size depends on file size distribution and usage patterns

File System Reliability

File System Consistency

File systems can become inconsistent due to system crashes:

- **Consistency checking:** Tools like `fsck` examine filesystem structures
 - Check block allocation consistency
 - Verify directory structure integrity
 - Cross-reference inodes and directory entries
 - Identify and fix common inconsistencies
- Common problems detected:
 - Missing blocks (allocated but not referenced)
 - Duplicate blocks (referenced by multiple files)
 - Free blocks marked as allocated
 - Directory inconsistencies

Journaling File Systems

Journaling improves filesystem reliability:

- **Goal:** Maintain filesystem consistency in the face of system failures
- **Concept:** Keep a log (journal) of intended changes before making them
- **Process:**
 - Write planned changes to journal
 - Mark journal entry as committed
 - Apply changes to filesystem
 - Mark journal entry as completed
- **Recovery:** After crash, replay uncommitted journal entries
- **Types:**
 - Metadata journaling: Only filesystem metadata is journaled
 - Full journaling: Both metadata and data are journaled

Understanding Inodes and Links

Exploring inodes and linking in Linux:

```
1 # Create a test file
2 echo "Hello, World!" > testfile.txt
3
4 # Check inode number
5 ls -li testfile.txt
6
7 # Create a hard link
8 ln testfile.txt hardlink.txt
9
10 # Check that both files have the same inode
11 ls -li testfile.txt hardlink.txt
12
13 # Check link count
14 ls -l testfile.txt
15
16 # Create a symbolic link
17 ln -s testfile.txt symlink.txt
18
19 # Check inode numbers (symlink has different inode)
20 ls -li testfile.txt hardlink.txt symlink.txt
21
22 # Check what symlink points to
23 ls -l symlink.txt
24
25 # Delete the original file
26 rm testfile.txt
27
28 # Hard link still works
29 cat hardlink.txt
30
31 # Symbolic link is broken
32 cat symlink.txt
33 # Output: cat: symlink.txt: No such file or directory
34
35 # Clean up
36 rm hardlink.txt symlink.txt
```

This demonstrates:

- Hard links share the same inode as the original file
- Symbolic links have their own inode and store a path
- Hard links remain valid even if the original name is removed
- Symbolic links break if the target is removed

File System Analysis

Examining filesystem structure and usage:

```
1 # Show filesystem usage
2 df -h
3
4 # Show inode usage
5 df -i
6
7 # Display filesystem type
8 lsblk -f
9
10 # Check filesystem properties
11 tune2fs -l /dev/sda1 | head -20
12
13 # Find large files
14 find /home -type f -size +100M 2>/dev/null | head -10
15
16 # Check filesystem fragmentation (ext2/3/4)
17 e2fsck -fn /dev/sda1
18
19 # Monitor filesystem activity
20 iotop
21
22 # Check for filesystem errors in logs
23 journalctl -u systemd-fsck* --no-pager
```


Advanced Storage Management

Beyond Basic Partitioning

Modern storage management goes beyond simple partitioning:

- Traditional approach: Fixed partitions with static sizing
- Advanced approach: Dynamic, flexible storage allocation
- Enables efficient use of storage resources
- Supports features like snapshots, thin provisioning, and live resizing
- Provides better fault tolerance and performance optimization

Non-persistent Storage: tmpfs

tmpfs (Temporary File System)

tmpfs is a file system that stores files in volatile memory:

- Files stored in RAM (not on disk)
- Extremely fast read and write operations based on memcpy
- Data is lost on system crash or shutdown (non-persistent)
- Used by many distributions for temporary directories (/tmp, /run)
- Can be mounted with size limits to prevent memory exhaustion
- Ideal for fast access to non-critical, temporary data

Working with tmpfs:

```
1 # Create a directory for tmpfs mount
2 mkdir -p /mnt/tmp
3
4 # Mount tmpfs with 2GB size limit
5 sudo mount -t tmpfs -o size=2G tmpfs /mnt/tmp
6
7 # Check filesystem usage
8 df -h /mnt/tmp
9
10 # Test performance
11 dd if=/dev/zero of=/mnt/tmp/speedtest bs=10M count=100
12
13 # Compare with disk-based filesystem
14 dd if=/dev/zero of=/tmp/speedtest bs=10M count=100
15
16 # Unmount when done
17 sudo umount /mnt/tmp
```

Logical Volume Management (LVM)

Logical Volume Management

LVM provides a method of allocating space on mass-storage devices that is more flexible than conventional partitioning:

- **Physical Volumes (PV):** Physical disks or partitions providing block storage
- **Volume Groups (VG):** Collections of physical volumes that form a storage pool
- **Logical Volumes (LV):** Virtual partitions created from volume group space
- Logical volumes can span multiple physical devices
- Enables dynamic resizing without unmounting filesystems
- Supports advanced features like snapshots and thin provisioning

LVM Features

LVM provides several advanced storage management features:

- **Free space management:** Allocate space from any available physical volume
- **Online space management:** Resize logical volumes while mounted
- **Snapshots:** Create point-in-time copies using copy-on-write
- **Balancing:** Distribute data evenly across physical volumes
- **Thin provisioning:** Allocate space on demand rather than upfront
- **Software RAID:** Combine multiple disks for redundancy or performance

Setting up LVM

Initialize physical volumes

- Prepare disks: `sudo pvcreate /dev/sdb /dev/sdc`
- Display PVs: `sudo pvdisplay`
- Scan for PVs: `sudo pvscan`

Create volume group

- Create VG: `sudo vgcreate myVG /dev/sdb /dev/sdc`
- Display VGs: `sudo vgdisplay`
- Extend VG: `sudo vgextend myVG /dev/sdd`

Create logical volumes

- Create LV: `sudo lvcreate -L 10G -n myLV myVG`
- Create LV using percentage: `sudo lvcreate -l 50%FREE -n myLV2 myVG`
- Display LVs: `sudo lvdisplay`

Resize logical volumes

- Extend LV: `sudo lvextend -resizefs -L +5G /dev/myVG/myLV`
- Reduce LV: `sudo lvreduce -resizefs -L -2G /dev/myVG/myLV`
- Resize to specific size: `sudo lvresize -resizefs -L 20G /dev/myVG/myLV`

Snapshot management

- Create snapshot: `sudo lvcreate -size 1G -snapshot -name myLV-snap /dev/myVG/myLV`
- Mount snapshot: `sudo mount /dev/myVG/myLV-snap /mnt/snapshot`
- Remove snapshot: `sudo lvremove /dev/myVG/myLV-snap`

Copy-on-Write and Snapshots

Copy-on-Write (COW)

COW is a resource management technique for efficient duplication:

- **Goal:** Efficiently duplicate or copy modifiable resources
- **Principle:** If a resource is shared but not modified, no copy is needed
- **Implementation:** Copy is made only upon modification (write operation)
- **Benefits:**
 - Saves storage space for identical data
 - Reduces initial duplication time
 - Minimizes I/O operations
- **Applications:** Virtual machines, snapshots, process forking

Snapshots

Snapshots provide point-in-time copies of data:

- **Purpose:** Allow full backup without disabling write access
- **Implementation:** Fix the state of a filesystem at a specific time using COW
- **Process:**
 - Create snapshot at time T
 - Original data continues to be accessible
 - New writes create copies, preserving original blocks
 - Snapshot contains the data as it existed at time T
- **Use cases:** Backups, testing, rollback scenarios

BtrFS (B-tree File System)

BtrFS Overview

BtrFS is a modern copy-on-write filesystem for Linux:

- Open source general-purpose COW filesystem (developed since 2007)
- Highly scalable (max volume size: 16 EB, max file size: 16 EB)
- Consolidates advanced features from multiple filesystems
- Includes copy-on-write, snapshots, logical volume management
- Built-in data and metadata integrity checking with checksums
- Integrated RAID support and compression capabilities
- Foundation for distributed storage systems like Ceph

BtrFS Design Goals

BtrFS aims to address limitations of traditional filesystems:

- Work well for diverse workloads and hardware configurations
- Maintain performance as the filesystem ages (avoid fragmentation issues)
- Scale from small devices (smartphones) to enterprise systems
- Provide scalability in multiple dimensions (disk space, memory, CPUs)
- Ensure data integrity through checksums and metadata duplication
- Support diverse storage devices (HDDs, SSDs, mixed arrays)
- Deliver performance comparable to or better than ext4 and XFS

BtrFS Key Concepts

BtrFS introduces several important concepts:

- **COW filesystem:** Uses copy-on-write for all updates instead of in-place modifications
- **Subvolumes:** Independent file trees that can be mounted separately
- **Extents:** Mappings from logical file areas to contiguous physical areas
- **Physical chunks:** Division of each device into manageable segments
- **Logical chunks:** Groupings of physical chunks for RAID and allocation
- **B-tree forest:** Multiple B-trees manage different aspects of the filesystem

BtrFS Architecture

BtrFS is constructed from a forest of COW-friendly B-trees:

- **Superblock:** Fixed location anchor point
- **Tree of tree roots:** Manages the forest of B-trees
- **Filesystem trees:** Manage directories and files within subvolumes
- **Extent tree:** Manages allocation of data blocks
- **Chunk tree:** Maps logical chunks to physical chunks
- **Device tree:** Manages multiple storage devices
- **Checksum tree:** Stores checksums for data integrity

BtrFS Advanced Features

BtrFS provides numerous advanced features:

- **Efficient small file storage:** Store file data inline within inodes
- **Data integrity:** Checksums for both data and metadata
- **Compression:** ZLIB (strong) or LZO (fast) compression at extent level
- **Multi-device support:** RAID 0, 1, 10 via logical chunks
- **Dynamic device management:** Add, remove, and rebalance devices online
- **Writable snapshots:** Memory-efficient clones as "first-class citizens"
- **Online operations:** Resize, defragmentation, extent relocation

BtrFS Setup and Management

Setting up and using BtrFS features:

```
1 # Create BtrFS filesystem on a single device
2 sudo mkfs.btrfs /dev/sdb1
3
4 # Mount the filesystem
5 sudo mount /dev/sdb1 /mnt/btrfs
6
7 # Create subvolumes
8 sudo btrfs subvolume create /mnt/btrfs/home
9 sudo btrfs subvolume create /mnt/btrfs/var
10
11 # List subvolumes
12 sudo btrfs subvolume list /mnt/btrfs
13
14 # Create a snapshot of a subvolume
15 sudo btrfs subvolume snapshot /mnt/btrfs/home /mnt/btrfs/home-snapshot
16
17 # Add another device to the filesystem
18 sudo btrfs device add /dev/sdc1 /mnt/btrfs
19
20 # Balance data across devices
21 sudo btrfs balance start -draid1 -mraid1 /mnt/btrfs
22
23 # Check filesystem status
24 sudo btrfs filesystem show /mnt/btrfs
25 sudo btrfs filesystem usage /mnt/btrfs
26
27 # Enable compression
28 sudo mount -o remount,compress=zlib /mnt/btrfs
29
30 # Defragment with compression
31 sudo btrfs filesystem defragment -czlib /mnt/btrfs/home
32
33 # Check for errors
34 sudo btrfs scrub start /mnt/btrfs
35 sudo btrfs scrub status /mnt/btrfs
```

RAID (Redundant Array of Independent Disks)

RAID Concept

RAID combines multiple physical drives for performance and/or reliability:

- **Goal:** Build performant and reliable mass storage systems
- **Methods:** Use multiple disks, parity information, and data striping
- **Benefits:** Improved performance, fault tolerance, or both
- **Implementation:** Hardware RAID (dedicated controllers) or software RAID (OS-based)

Common RAID Levels

Different RAID levels provide various combinations of performance and redundancy:

- **RAID 0 (Striping):**
 - Data striped across multiple drives
 - Improved performance, no redundancy
 - Total capacity equals sum of all drives
 - Failure of any drive results in total data loss
- **RAID 1 (Mirroring):**
 - Data duplicated on multiple drives
 - High redundancy, no performance improvement for writes
 - Total capacity equals smallest drive
 - Can survive failure of all but one drive
- **RAID 5 (Striping with parity):**
 - Data and parity information striped across drives
 - Good balance of performance, storage efficiency, and redundancy
 - Can survive failure of one drive
 - Requires minimum of three drives
- **RAID 10 (Stripe of mirrors):**
 - Combines RAID 0 and RAID 1
 - High performance and redundancy
 - Requires minimum of four drives
 - Can survive multiple drive failures if not in same mirror set

Union File Systems

Union File Systems Concept

Union file systems overlay multiple file systems to create a unified view:

- **Purpose:** Combine read-only and read-write filesystems
- **Original motivation:** Enable writing on read-only media (LiveCDs, DVDs)
- **Modern applications:** Container technologies, software distribution
- **Key implementations:** AuFS (Another Union File System), OverlayFS

OverlayFS

OverlayFS is the mainline Linux union filesystem:

- **Components:**
 - **lowerdir:** Read-only base layer(s)
 - **upperdir:** Read-write layer for modifications
 - **workdir:** Working directory for atomic operations
 - **merged:** Combined view of all layers
- **Operation:**
 - All modifications go to the upper layer
 - Lower layers remain unchanged
 - Files in upper layer shadow files in lower layers
 - Deletions marked with "whiteout" files
- **Use cases:** Docker containers, software packaging, live systems

OverlayFS Setup

Creating and using an overlay filesystem:

```
1 # Create directory structure
2 mkdir -p /tmp/overlay/{lower,upper,work,merged}
3
4 # Populate lower directory (read-only base)
5 echo "Original file from lower" > /tmp/overlay/lower/file1.txt
6 echo "Lower layer file" > /tmp/overlay/lower/file2.txt
7
8 # Create the overlay mount
9 sudo mount -t overlay overlay \
10     -o
11         lowerdir=/tmp/overlay/lower,upperdir=/tmp/overlay/upper,workdir=/tmp/overlay/work
12         \
13         /tmp/overlay/merged
14
15 # View merged contents
16 ls -la /tmp/overlay/merged/
17
18 # Modify a file (goes to upper layer)
19 echo "Modified content" > /tmp/overlay/merged/file1.txt
20
21 # Create a new file (goes to upper layer)
22 echo "New file in upper" > /tmp/overlay/merged/file3.txt
23
24 # Check what's in each layer
25 echo "=== Lower layer ==="
26 ls -la /tmp/overlay/lower/
27
28 echo "=== Upper layer ==="
29 ls -la /tmp/overlay/upper/
30
31 echo "=== Merged view ==="
32 ls -la /tmp/overlay/merged/
33
34 # Delete a file from lower layer (creates whiteout)
35 rm /tmp/overlay/merged/file2.txt
36
37 # Check for whiteout file
38 ls -la /tmp/overlay/upper/
39
40 # Unmount overlay
41 sudo umount /tmp/overlay/merged
```

Working with Advanced Filesystems

Partition and format management

- Create GPT partition table: `sudo parted /dev/sdb mklabel gpt`
- Create partition: `sudo parted /dev/sdb mkpart primary 0% 100%`
- Format with different filesystems:
 - ext4: `sudo mkfs.ext4 /dev/sdb1`
 - XFS: `sudo mkfs.xfs /dev/sdb1`
 - Btrfs: `sudo mkfs.btrfs /dev/sdb1`

Filesystem mounting and options

- Mount with compression: `sudo mount -o compress=zlib /dev/sdb1 /mnt`
- Mount with specific options: `sudo mount -o noatime,nodiratime /dev/sdb1 /mnt`
- Check mount options: `mount | grep /mnt`
- Add to fstab: `echo '/dev/sdb1 /mnt btrfs defaults,compress=zlib 0 2' | sudo tee -a /etc/fstab`

Performance testing

- Test sequential write: `dd if=/dev/zero of=/mnt/testfile bs=1M count=1000`
- Test random I/O: `fio -name=random-rw -ioengine=posix -rw=randrw -bs=4k -numjobs=1 -size=1g -runtime=60 -time_based -filename=/mnt/testfile`
- Monitor I/O: `iostat -a`
- Check compression ratio: `btrfs filesystem usage /mnt`

Filesystem maintenance

- Check filesystem: `sudo fsck /dev/sdb1`
- Defragment (ext4): `sudo e4defrag /mnt`
- Defragment (Btrfs): `sudo btrfs filesystem defragment /mnt`
- Balance (Btrfs): `sudo btrfs balance start /mnt`
- Scrub (Btrfs): `sudo btrfs scrub start /mnt`

Comparing Filesystem Performance

Testing different filesystems for performance characteristics:

```
1 # Create test partitions
2 sudo parted /dev/sdb mklabel gpt
3 sudo parted /dev/sdb mkpart primary 0% 33%
4 sudo parted /dev/sdb mkpart primary 33% 66%
5 sudo parted /dev/sdb mkpart primary 66% 100%
6
7 # Format with different filesystems
8 sudo mkfs.ext4 /dev/sdb1
9 sudo mkfs.xfs /dev/sdb2
10 sudo mkfs.btrfs -f /dev/sdb3
11
12 # Create mount points
13 sudo mkdir -p /mnt/{ext4,xfs,btrfs}
14
15 # Mount filesystems
16 sudo mount /dev/sdb1 /mnt/ext4
17 sudo mount /dev/sdb2 /mnt/xfs
18 sudo mount -o compress=zlib /dev/sdb3 /mnt/btrfs
19
20 # Test file creation performance
21 for fs in ext4 xfs btrfs; do
22     echo "Testing $fs file creation..."
23     time sh -c "for i in {1..1000}; do touch /mnt/$fs/file_$i; done"
24 done
25
26 # Test large file write performance
27 for fs in ext4 xfs btrfs; do
28     echo "Testing $fs large file write..."
29     time dd if=/dev/zero of=/mnt/$fs/bigfile bs=1M count=500 2>/dev/null
30 done
31
32 # Test compression ratio (Btrfs only)
33 echo "=== Btrfs compression analysis ==="
34 sudo btrfs filesystem usage /mnt/btrfs
35
36 # Clean up
37 sudo umount /mnt/{ext4,xfs,btrfs}
```

Container Storage with OverlayFS

Demonstrating container-like storage layering:

```
1 # Create base image layer
2 mkdir -p /tmp/container/{base,app,runtime,merged,work}
3
4 # Base layer (OS files)
5 echo "#!/bin/bash" > /tmp/container/base/entrypoint.sh
6 echo "echo 'Hello from base layer'" >> /tmp/container/base/entrypoint.sh
7 chmod +x /tmp/container/base/entrypoint.sh
8
9 # Application layer
10 echo "config_file=app.conf" > /tmp/container/app/app.config
11 echo "#!/bin/bash" > /tmp/container/app/start-app.sh
12 echo "echo 'Starting application...'" >> /tmp/container/app/start-app.sh
13 chmod +x /tmp/container/app/start-app.sh
14
15 # Create overlay combining base and app layers
16 sudo mount -t overlay overlay \
17     -o
18     lowerdir=/tmp/container/app:/tmp/container/base,upperdir=/tmp/container/runtime,workdir=/tmp/container/work
19     \
20     /tmp/container/merged
21
22 # View combined filesystem
23 ls -la /tmp/container/merged/
24
25 # Make runtime changes (goes to runtime layer)
26 echo "Runtime log entry" > /tmp/container/merged/runtime.log
27 echo "user_data=changed" > /tmp/container/merged/app.config
28
29 # Show layering
30 echo "=== Base layer ==="
31 ls -la /tmp/container/base/
32
33 echo "=== App layer ==="
34 ls -la /tmp/container/app/
35
36 echo "=== Runtime layer ==="
37 ls -la /tmp/container/runtime/
38
39 echo "=== Merged view ==="
40 ls -la /tmp/container/merged/
41
42 # Unmount
43 sudo umount /tmp/container/merged
```


Networking

Networking Requirements on Operating Systems

Operating System Networking Requirements

Modern operating systems must support comprehensive networking capabilities:

- **Local interprocess communication:** Stream or datagram communication between processes
- **External network connectivity:** Connect machines to networks for distributed applications
- **Security and protection:** Protect against malicious traffic and network-related attacks
- **Traffic routing:** Support and route traffic between different network interfaces
- **Virtualization support:** Network namespaces and virtual interfaces for cloud computing
- **Distributed applications:** Support for mail, web services, remote access, printing, etc.

Network Interface Hardware

Network Interface Card (NIC) Architecture

Network interface cards connect systems to networks via the system bus:

- **Physical Layer Device (PHY):**
 - Maintains electrical characteristics defined by the protocol
 - Establishes communication with counterpart devices
 - Decodes, descrambles, and deserializes incoming signals
 - Encodes, scrambles, and serializes outgoing data
 - Delivers bit stream to the Media Access Controller
- **Media Access Controller (MAC):**
 - Handles Ethernet Layer-2 protocols and traffic
 - Configures PHY via MDIO interface
 - Processes frame integrity (FCS, buffer management)
 - Places data into main memory buffers
 - Activates NIC driver via interrupts

DMA and Data Transfer

Modern NICs use Direct Memory Access for efficient data transfer:

- MAC-memory data transfer performed by DMA
- Supports multiple concurrent DMA operations
- Each operation described by buffer descriptors
- Often uses Scatter-Gather mechanism for efficiency
- Prevents buffer underflow and overflow conditions
- Reduces CPU overhead for bulk data transfers

NIC Device Drivers

NIC drivers operate in kernel space and manage hardware interaction:

- **Receive (Rx) side:**
 - Notified of incoming frames via interrupts
 - Checks frame status and discards erroneous frames
 - Formats frames for software stack consumption
- **Transmit (Tx) side:**
 - Initializes MAC DMA channels for transmission
 - Notifies MAC when frames need transmission
 - Monitors transmission status
- Requires host-side buffering system for efficient operation

Network Stack Architecture

Zero-Copy Stack Design

Traditional layered networking creates inefficiencies:

- **Naive approach:** Frame copied between each layer
- **Problems:** Multiple memory copies, CPU intensive operations
- **Zero-copy solution:**
 - All frames pooled in buffers at Layer 2
 - Only pointers to frame metadata passed between layers
 - Frame payload remains in original location
 - Significantly reduces memory operations and CPU usage

Linux sk_buff Structure

Linux implements zero-copy networking using sk_buff (SKB):

- Fundamental structure containing frame processing variables
- Enables communication between network layers
- SKB cloned during layer transfers (metadata copied, payload referenced)
- NIC maintains ring buffer of SKB pointers in kernel memory
- Scatter-Gather DMA transfers data directly to SKB structures
- Provides efficient frame processing with minimal copying

Layer 2 - Data Link Layer

Layer 2 Processing

The data link layer handles frame-level operations:

- Differentiates between ARP and other frame types
- Checks Ethernet frame characteristics
- Validates frame integrity and format
- Passes frames to appropriate upper layer protocols
- Implements bridging between network segments
- Manages local network addressing (MAC addresses)

Layer 2 Bridging

Bridging connects multiple network segments:

- **Network bridging:** Connects two or more LAN segments
- **Interface bonding:** Combines multiple NICs for performance or redundancy
 - Load balancing across multiple interfaces
 - Failover protection
 - Increased aggregate bandwidth
- **Virtual bridges:** Software-based bridging for virtual networks
- Integration with Linux netfilter system for filtering and processing

Layer 3 - Network Layer

Layer 3 Network Processing

The network layer provides internetworking capabilities:

- **Addressing:** IP addressing beyond LAN boundaries
- **Routing:** Forwards packets to destinations beyond local subnets
- **Fragmentation:** Handles packet size limitations
- **Quality of Service:** Manages traffic priorities and bandwidth
- **Security:** Firewall functionality for traffic filtering

Linux Netfilter System

Netfilter provides packet filtering and manipulation:

- **Functions:**
 - Packet selection and filtering
 - Network Address Translation (NAT)
 - Packet mangling and modification
 - Connection tracking and state management
 - Statistics collection and logging
- **Implementation:** Hook system with five points in routing path
- **Hook return values:**
 - NF_ACCEPT: Packet continues traversal
 - NF_DROP: Discard packet silently
 - NF_STOLEN: Packet extracted for special processing
 - NF_QUEUE: Queue packet for userspace processing
 - NF_REPEAT: Call hook function again
- **Frontends:** iptables, nftables for user configuration

IP Packet Processing

Linux IP layer processing involves several steps:

- Network driver calls IP handler (ip_rcv)
- Perform sanity checks on IP header (length, checksum)
- Routing subsystem determines packet destination
- Local delivery or forwarding decision
- Integration with netfilter hooks for security and processing
- Connection tracking for stateful operations

Layer 4 - Transport Layer

Transport Layer Protocols

Layer 4 provides end-to-end communication services:

- **UDP (User Datagram Protocol):**
 - Thin wrapper around IP frames
 - Provides unreliable, connectionless transmission
 - Minimal overhead for simple request-response applications
 - Used for DNS, DHCP, streaming media
- **TCP (Transmission Control Protocol):**
 - Reliable, connection-oriented transport
 - Manages connections, flow control, congestion control
 - Assembles data streams from individual segments
 - Provides error detection and recovery
 - Used for HTTP, FTP, SSH, email

Layer 7 - Application Interface

Berkeley Sockets

Sockets provide the user-facing API for network programming:

- Connect port numbers and network addresses with protocols and processes
- **Socket types:**
 - SOCK_STREAM: Reliable, ordered, connection-based (TCP)
 - SOCK_DGRAM: Unreliable, unordered, connectionless (UDP)
 - SOCK_RAW: Direct access to IP layer
- **Address families:**
 - AF_INET: IPv4 Internet protocols
 - AF_INET6: IPv6 Internet protocols
 - AF_UNIX: Local inter-process communication
 - AF_NETLINK: Kernel-userspace communication
- Sockets have states and support standard file operations

Special Socket Types

Linux supports various specialized socket types:

- **Unix Domain Sockets:**
 - Inter-process communication using socket interface
 - Operates over the filesystem namespace
 - Cannot traverse host boundaries
 - Higher performance than network sockets for local communication
- **Loopback Interface:**
 - Host network interface with itself
 - Can be accessed like any network interface
 - Used for local services and testing
- **Netlink Sockets:**
 - Kernel-userspace IPC mechanism
 - Used for configuration and monitoring
 - Families include NETLINK_ROUTE, NETLINK_AUDIT, etc.
 - Standard message format with adaptable headers

Basic TCP Socket Programming

Example of TCP socket server and client:

```
1 // TCP Server
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4 #include <arpa/inet.h>
5
6 int server_fd, client_fd;
7 struct sockaddr_in address;
8 int addrlen = sizeof(address);
9
10 // Create socket
11 server_fd = socket(AF_INET, SOCK_STREAM, 0);
12
13 // Setup address
14 address.sin_family = AF_INET;
15 address.sin_addr.s_addr = INADDR_ANY;
16 address.sin_port = htons(8080);
17
18 // Bind socket to address
19 bind(server_fd, (struct sockaddr *)&address, sizeof(address));
20
21 // Listen for connections
22 listen(server_fd, 3);
23
24 // Accept connection
25 client_fd = accept(server_fd, (struct sockaddr *)&address,
26                   (socklen_t*)&addrlen);
27
28 // Read/write data
29 char buffer[1024] = {0};
30 read(client_fd, buffer, 1024);
31 send(client_fd, "Hello from server", 17, 0);
32
33 // TCP Client
34 int sock = 0;
35 struct sockaddr_in serv_addr;
36
37 // Create socket
38 sock = socket(AF_INET, SOCK_STREAM, 0);
39
40 // Setup server address
41 serv_addr.sin_family = AF_INET;
42 serv_addr.sin_port = htons(8080);
43 inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);
44
45 // Connect to server
46 connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
47
48 // Send/receive data
49 send(sock, "Hello from client", 17, 0);
50 read(sock, buffer, 1024);
```

Network Configuration Tools

Linux Network Configuration

Linux provides various tools for network management:

- **iproute2 suite** (modern, preferred):
 - ip: Network interface and routing configuration
 - tc: Traffic control and QoS management
 - ss: Socket statistics and monitoring
 - bridge: Bridge configuration and management
- **net-tools suite** (legacy, deprecated):
 - ifconfig: Interface configuration
 - route: Routing table manipulation
 - netstat: Network statistics
 - arp: ARP table management
- Both suites use different underlying mechanisms (netlink vs ioctl)

Network Configuration and Troubleshooting

Interface management

- Show interfaces: `ip link show`
- Bring interface up: `sudo ip link set eth0 up`
- Assign IP address: `sudo ip addr add 192.168.1.100/24 dev eth0`
- Show IP addresses: `ip addr show`
- Remove IP address: `sudo ip addr del 192.168.1.100/24 dev eth0`

Routing management

- Show routing table: `ip route show`
- Add default route: `sudo ip route add default via 192.168.1.1`
- Add specific route: `sudo ip route add 10.0.0.0/8 via 192.168.1.1`
- Delete route: `sudo ip route del 10.0.0.0/8`
- Monitor route changes: `ip monitor route`

Network diagnostics

- Test connectivity: `ping 8.8.8.8`
- Trace route: `traceroute google.com`
- DNS lookup: `dig google.com`
- Show listening sockets: `ss -tuln`
- Show network statistics: `ss -s`
- Monitor network traffic: `tcpdump -i eth0`

Firewall configuration

- List iptables rules: `sudo iptables -L`
- Allow SSH: `sudo iptables -A INPUT -p tcp -dport 22 -j ACCEPT`
- Block specific IP: `sudo iptables -A INPUT -s 192.168.1.100 -j DROP`
- Save rules: `sudo iptables-save > /etc/iptables/rules.v4`
- Modern alternative: `sudo nft list ruleset`

Network Interface Configuration

Complete network interface setup:

```
1 # Show current network configuration
2 ip addr show
3
4 # Configure a static IP address
5 sudo ip addr add 192.168.1.100/24 dev eth0
6 sudo ip link set eth0 up
7
8 # Add default gateway
9 sudo ip route add default via 192.168.1.1
10
11 # Configure DNS (temporary)
12 echo "nameserver 8.8.8.8" | sudo tee /etc/resolv.conf
13
14 # Test connectivity
15 ping -c 3 8.8.8.8
16 ping -c 3 google.com
17
18 # Show routing table
19 ip route show
20
21 # Show ARP table
22 ip neigh show
23
24 # Monitor network activity
25 ss -tuln | grep :80
26
27 # Configure persistent networking (systemd-networkd)
28 cat << EOF | sudo tee /etc/systemd/network/10-eth0.network
29 [Match]
30 Name=eth0
31
32 [Network]
33 Address=192.168.1.100/24
34 Gateway=192.168.1.1
35 DNS=8.8.8.8
36 DNS=8.8.4.4
37 EOF
38
39 # Enable systemd-networkd
40 sudo systemctl enable systemd-networkd
41 sudo systemctl restart systemd-networkd
```

Socket Programming Example

Simple client-server application:

```
1 // Simple UDP Echo Server
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7
8 int main() {
9     int sockfd;
10    struct sockaddr_in servaddr, cliaddr;
11    char buffer[1024];
12    socklen_t len = sizeof(cliaddr);
13
14    // Create UDP socket
15    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
16    if (sockfd < 0) {
17        perror("socket creation failed");
18        exit(EXIT_FAILURE);
19    }
20
21    // Setup server address
22    memset(&servaddr, 0, sizeof(servaddr));
23    servaddr.sin_family = AF_INET;
24    servaddr.sin_addr.s_addr = INADDR_ANY;
25    servaddr.sin_port = htons(8080);
26
27    // Bind socket
28    if (bind(sockfd, (const struct sockaddr *)&servaddr,
29        sizeof(servaddr)) < 0) {
30        perror("bind failed");
31        exit(EXIT_FAILURE);
32    }
33
34    printf("UDP Echo Server listening on port 8080...\n");
35
36    // Echo loop
37    while (1) {
38        int n = recvfrom(sockfd, buffer, 1024, 0,
39            (struct sockaddr *)&cliaddr, &len);
40        buffer[n] = '\0';
41        printf("Received: %s\n", buffer);
42
43        sendto(sockfd, buffer, strlen(buffer), 0,
44            (const struct sockaddr *)&cliaddr, len);
45    }
46
47    return 0;
48 }
49
50 // Compile: gcc -o udp_server udp_server.c
51 // Test: echo "Hello" | nc -u localhost 8080
```

Network Namespace Isolation

Creating isolated network environments:

```
1 # Create a new network namespace
2 sudo ip netns add isolated
3
4 # List network namespaces
5 ip netns list
6
7 # Show interfaces in the new namespace (should be empty except lo)
8 sudo ip netns exec isolated ip link show
9
10 # Create a virtual ethernet pair
11 sudo ip link add veth0 type veth peer name veth1
12
13 # Move one end to the isolated namespace
14 sudo ip link set veth1 netns isolated
15
16 # Configure the host-side interface
17 sudo ip addr add 10.0.0.1/24 dev veth0
18 sudo ip link set veth0 up
19
20 # Configure the namespace-side interface
21 sudo ip netns exec isolated ip addr add 10.0.0.2/24 dev veth1
22 sudo ip netns exec isolated ip link set veth1 up
23 sudo ip netns exec isolated ip link set lo up
24
25 # Test connectivity
26 ping -c 3 10.0.0.2
27
28 # Run commands in the isolated namespace
29 sudo ip netns exec isolated ping -c 3 10.0.0.1
30
31 # Show routing in the namespace
32 sudo ip netns exec isolated ip route show
33
34 # Add default route in namespace (through host)
35 sudo ip netns exec isolated ip route add default via 10.0.0.1
36
37 # Enable IP forwarding on host for internet access
38 echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward
39
40 # Add NAT rule for namespace traffic
41 sudo iptables -t nat -A POSTROUTING -s 10.0.0.0/24 -j MASQUERADE
42
43 # Test internet connectivity from namespace
44 sudo ip netns exec isolated ping -c 3 8.8.8.8
45
46 # Clean up
47 sudo ip netns delete isolated
48 sudo ip link delete veth0
```

Advanced Network Monitoring

Comprehensive network monitoring and analysis:

```
1 # Monitor real-time network connections
2 watch -n 1 'ss -tuln'
3
4 # Show detailed socket information
5 ss -tulpn | grep :80
6
7 # Monitor network traffic with detailed statistics
8 iftop -i eth0
9
10 # Capture and analyze packets
11 sudo tcpdump -i eth0 -w capture.pcap
12
13 # Analyze captured packets
14 tcpdump -r capture.pcap -n | head -20
15
16 # Monitor bandwidth usage per process
17 sudo nethogs eth0
18
19 # Show network interface statistics
20 cat /proc/net/dev
21
22 # Monitor netlink messages
23 ip monitor all
24
25 # Check connection tracking
26 sudo cat /proc/net/nf_conntrack | head -10
27
28 # Show detailed network stack statistics
29 ss --info --processes --numeric
30
31 # Monitor DNS queries
32 sudo tcpdump -i eth0 port 53
33
34 # Show kernel network parameters
35 sysctl -a | grep net.ipv4 | head -10
36
37 # Monitor network errors
38 netstat -i
```

Extended Berkeley Packet Filter (eBPF)

Modern Linux networking includes advanced packet filtering:

- **eBPF**: Extended packet filtering beyond traditional BPF
- **Features**:
 - Programmable packet processing in kernel space
 - JIT compilation for performance
 - Safe execution environment with verification
 - Integration with various kernel subsystems
- **Applications**:
 - High-performance networking
 - Container networking
 - Network security and monitoring
 - Traffic analysis and manipulation
- Represents the future of Linux packet filtering and network programming

Network Service Management

Service discovery and management

- Check listening services: `sudo ss -tulpn`
- Find process using port: `sudo lsof -i :80`
- Check service status: `systemctl status nginx`
- Control network services: `sudo systemctl start/stop/restart nginx`

Network security

- Check open ports: `nmap localhost`
- Monitor failed connections: `sudo journalctl -f | grep ssh`
- Configure fail2ban: `sudo systemctl enable fail2ban`
- Check SELinux network policies: `getsebool -a | grep httpd`

Performance optimization

- Tune network buffers: `sysctl -w net.core.rmem_max=16777216`
- Enable TCP window scaling: `sysctl -w net.ipv4.tcp_window_scaling=1`
- Configure congestion control: `sysctl -w net.ipv4.tcp_congestion_control=bbr`
- Monitor network performance: `iperf3 -c server_ip`