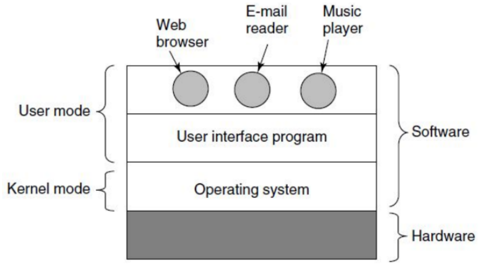
Betriebssysteme

Jil Zerndt
FS 2025

Introduction to Operating Systems

Basic Principles of Operating Systems

Operating System (OS)



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

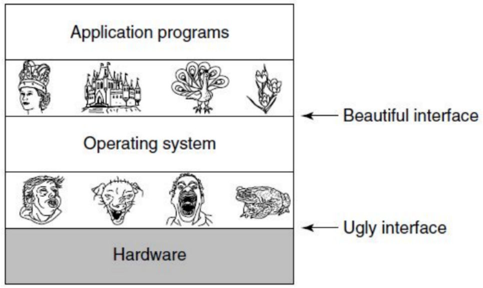
The OS provides two key interfaces:

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

The Operating System as an Extended Machine

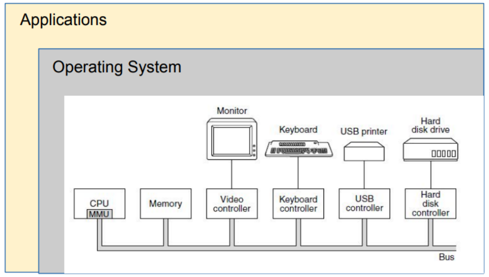
The operating system serves as an abstraction layer between hardware and applications (software that manages hardware)

- 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



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:

```
# Display kernel version information
uname -a

# View CPU details
less /proc/cpuinfo

# Display CPU architecture information
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 `pstree`
- Check system call activity with `strace`

CPU and Memory Architecture

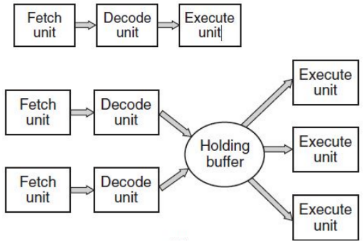
CPU Central Processing Unit

- 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



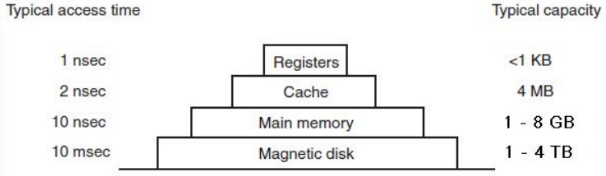
Basic cycle of every CPU:

- 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:

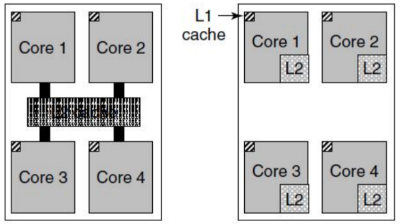
- Registers (fastest, smallest)
- Cache memory (L1, L2, L3)
- Main memory (RAM)
- Secondary storage (disks, SSDs)
- Tertiary storage (backup systems)



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.



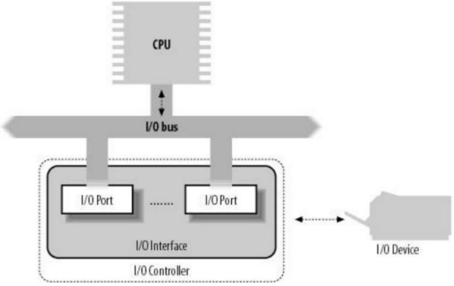
Input/Output (I/O) System

The I/O system comprises:

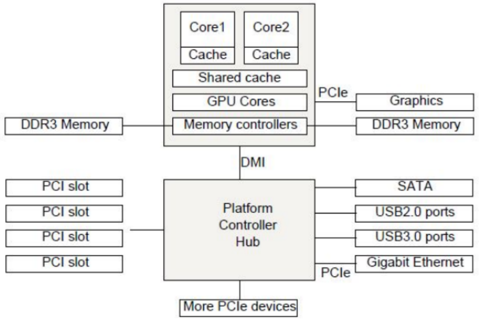
- Various devices (hard drives, network interfaces, serial ports, keyboards, etc.)
- Bus systems to connect devices (PCI, PCIe, SATA, USB, etc.) (software)
- I/O controllers (hardware)
- Device drivers (software)

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:



Linux Primer

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

```
# Check current user
whoami

# View hardware information
lshw -short

# Monitor system messages
dmesg | grep usb

# Check disk usage
df -h

# Find a process ID
pidof firefox

# Install a package
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:

```
# Create a test file
touch delta.txt
# Add content to the file
echo "Hello, this is a test" > delta.txt
# View the file content
cat delta.txt
# Stop and restart the VM from OpenStack dashboard
# Then check if the file persists
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

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

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 2^{64} bytes
- Uses unique UUIDs for partitions to avoid collisions
- Features modular design for extendability
- Includes SSecure Boot"to restrict which binaries can be executed
- Uses cryptographic signatures and X.509 certificates for verification

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:

```
# Display current boot entries
sudo efibootmgr -v

# List contents of EFI system partition
sudo ls -la /boot/efi/EFI/

# View disk partition table format (MBR or GPT)
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:

```
# View GRUB configuration
cat /boot/grub/grub.cfg

# Examine a specific menu entry showing kernel and initrd paths
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
- Lingering 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:

```
[Unit]
Description=Example Service
Documentation=https://example.com/docs
After=network.target
Wants=network-online.target
Requires=example-dependency.service

[Service]
Type=simple
ExecStart=/usr/bin/example-service
Restart=on-failure
User=exampleuser

[Install]
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:

```
# Create a service unit file
sudo nano /etc/systemd/system/myservice.service

# Add the following content
[Unit]
Description=My Simple Service
After=network.target

[Service]
Type=simple
ExecStart=/bin/bash -c 'echo "Service started at $(date)" > /home/ubuntu/service_started'
Restart=on-failure

[Install]
WantedBy=multi-user.target

# Reload systemd to recognize the new service
sudo systemctl daemon-reload

# Start the service
sudo systemctl start myservice

# Verify it worked
cat /home/ubuntu/service_started

# Enable auto-start on boot
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

```
# List all processes with details
ps aux
# Interactive process viewer
top
# Show environment variables
env
# Run a process in the background
wc /dev/zero &
# List background jobs
jobs
# Bring a background job to foreground
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:

```
# When you run a command in the terminal:
ls -la

# The shell:
# 1. Forks itself (duplicate the process)
# 2. Child process executes the 'ls' binary
# 3. Parent waits for child to complete
# 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:

```
# Start a process
wc /dev/zero &

# Get its process ID
pidof wc

# Terminate the process
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 $\rightarrow$ kernel $\rightarrow$ 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 SZïndicates zombie)

Creating Processes in C

Simple process creation with fork():

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

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        // Error
        fprintf(stderr, "Fork failed\n");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Child process: PID = %d\n", getpid());
        sleep(5);
    } else {
        // Parent process
        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
        sleep(10);
    }

    return 0;
}
```

Creating Threads in C

Simple thread creation with POSIX threads:

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

void* thread_function(void* arg) {
    int thread_id = *(int*)arg;
    printf("Thread %d: running\n", thread_id);
    sleep(3);
    printf("Thread %d: exiting\n", thread_id);
    return NULL;
}

int main() {
    pthread_t threads[2];
    int thread_args[2];

    for (int i = 0; i < 2; i++) {
        thread_args[i] = i;
        pthread_create(&threads[i], NULL, thread_function, &thread_args[i]);
        printf("Main: created thread %d\n", i);
    }

    // Wait for threads to finish
    for (int i = 0; i < 2; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Main: all threads completed\n");
    return 0;
}
```

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. supercomputer)
- 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 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 Terminology
Key concepts in cgroups:

- **cgroup**: 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 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:

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

Device Controller Example
Restricting access to a device:

```
# Create a cgroup for device control
mkdir /sys/fs/cgroup/devices/group0
# By default, full permissions exist
cat /sys/fs/cgroup/devices/group0/devices.list
# Output: a *:* rwm (all devices, all permissions)

# Deny access to /dev/null (major 1, minor 3)
echo 'c 1:3 rwm' > /sys/fs/cgroup/devices/group0/devices.deny
# Add the current shell to the group
echo $$ > /sys/fs/cgroup/devices/group0/tasks
# Try to use /dev/null - it will fail
echo "test" > /dev/null
# 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

Mounting cgroups v1

Mounting cgroups controllers:

```
# Create a tmpfs mount for cgroups
sudo mount -t tmpfs -o size=10M tmpfs /sys/fs/cgroup

# Mount a single controller (individual resource control)
sudo mount -t cgroup -o cpu none /sys/fs/cgroup/cpu

# Co-mount multiple controllers (collective resource control)
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:

```
# Create a new cgroup
mkdir /sys/fs/cgroup/cpu/cg1

# Add the current process to the cgroup
echo $$ > /sys/fs/cgroup/cpu/cg1/cgroup.procs

# Add a specific process to the cgroup
echo <PID> > /sys/fs/cgroup/cpu/cg1/cgroup.procs

# Remove a cgroup (must be empty of processes and child cgroups)
rmdir /sys/fs/cgroup/cpu/cg1
```

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

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%`

Using systemd Resource Control

Controlling resource limits with systemd:

```
# View current system slices
systemctl list-units --type=slice

# Check user slice settings
systemctl show user.slice

# Limit CPU usage for all user processes to 20%
sudo systemctl set-property user.slice CPUQuota=20%

# Create a resource-limited service
cat << EOF > /etc/systemd/system/limited-service.service
[Unit]
Description=Resource Limited Service

[Service]
ExecStart=/usr/bin/sha1sum /dev/zero
CPUQuota=10%
MemoryLimit=100M

[Install]
WantedBy=multi-user.target
EOF

# Reload systemd and start the service
sudo systemctl daemon-reload
sudo systemctl start limited-service

# Monitor resource usage
systemd-cgtop
```

Creating a Custom CPU Control

Creating a CPU affinity control from scratch:

```
# Create a tmpfs mount
sudo mkdir -p /mnt/cgroups
sudo mount -t tmpfs none /mnt/cgroups

# Mount cpuset controller
sudo mkdir -p /mnt/cgroups/cpuset
sudo mount -t cgroup -o cpuset none /mnt/cgroups/cpuset

# Create a cgroup
sudo mkdir /mnt/cgroups/cpuset/group1

# Configure required memory settings first
echo 0 > /mnt/cgroups/cpuset/group1/cpuset.mems

# Restrict to CPUs 0 and 1 only (assuming 4 CPUs)
echo "0-1" > /mnt/cgroups/cpuset/group1/cpuset.cpus

# Run CPU-intensive processes
for i in {1..4}; do
  # Create processes
  sha1sum /dev/zero &
  # Get PID and add to cgroup
  PID=$!
  echo $PID > /mnt/cgroups/cpuset/group1/cgroup.procs
  echo "Added process $PID to CPU-restricted group"
done

# Check CPU assignment with taskset
for pid in $(cat /mnt/cgroups/cpuset/group1/cgroup.procs); do
  taskset -p $pid
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 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

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)

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 Concept
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

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

Address Translation
logical ⇆ physical

Virtual memory requires transl. between 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

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

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

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
Linux supports huge pages 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:

```
# Get PID of a process
PID=$(pidof firefox)

# Check virtual and resident memory size
ps -o pid,comm,vsz,rss -p $PID

# Analyze memory map segments
cat /proc/$PID/maps | head -10

# View detailed memory status
grep -E 'VmSize|VmRSS|VmData|VmStk|VmExe' /proc/$PID/status

# Map process address space in detail
pmap -x $PID | head -20

# Check page faults
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:

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

int main() {
    // Get process ID
    pid_t pid = getpid();
    printf("Process ID: %d\n", pid);

    // Get page size
    int page_size = getpagesize();
    printf("Page size: %d bytes\n", page_size);

    // Allocate memory for 10 pages
    size_t size = 10 * page_size;
    char *buffer = malloc(size);
    printf("Allocated %zu bytes (%zu pages)\n",
           size, size / page_size);

    // Check page faults before access
    printf("Check page faults with: ps -o min_flt,maj_flt %d\n", pid);
    printf("Press Enter to continue...\n");
    getchar();

    // Access the memory (causes page faults)
    for (int i = 0; i < size; i += page_size) {
        buffer[i] = 1;  // Touch one byte per page
    }

    // Check page faults after access
    printf("Check page faults again with: ps -o min_flt,maj_flt %d\n", pid);
    printf("Press Enter to continue...\n");
    getchar();

    // Allocate page-aligned memory
    void *aligned_buf = NULL;
    int result = posix_memalign(&aligned_buf, page_size, size);
    if (result == 0) {
        printf("Allocated %zu bytes aligned on page boundary\n", size);
    }

    // Free memory
    free(buffer);
    free(aligned_buf);

    return 0;
}
```

This example demonstrates:

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

Input / Output - Part I

Input/Output Basics

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 Principles

I/O Access Methods
Two primary methods for CPU to interact with I/O devices:
- **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 initiates 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 I/O Subsystem

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

sysfs is a 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

udev

udev is the 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

/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

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:

```
# Basic write test (with caching)
dd if=/dev/zero of=speedtest bs=10M count=100
rm speedtest

# Write test with synchronous I/O (forces data to disk)
dd if=/dev/zero of=speedtest bs=10M count=100 conv=fdatasync
rm speedtest

# Write test with direct I/O (bypasses the page cache)
dd if=/dev/zero of=speedtest bs=10M count=100 oflag=direct
rm speedtest

# Monitor disk I/O activity during test
iostat -x 1

# Check disk utilization statistics
iostat -p sda

# Explanation:
# - Standard dd shows high performance but may not be on disk yet
# - fdatasync ensures data is physically written to disk
# - direct bypasses the OS cache, showing raw device performance
```

Device Information Analysis

Exploring device information in Linux:

```
# List block devices with details
lsblk -f

# Get detailed information about a specific device
udevadm info --query=all --name=/dev/sda1

# Examine sysfs entries for the device
ls -l /sys/block/sda/sda1/

# Check device attributes
cat /sys/block/sda/queue/scheduler
cat /sys/block/sda/queue/read_ahead_kb

# Get PCI information for a disk controller
lspci | grep -i sata

# Check all properties of a device
udevadm info --attribute-walk --name=/dev/sda1 | less

# Monitor udev events when plugging in a USB device
udevadm monitor --property
# (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

Linux follows a time-based release process:

- New major kernel releases every 2-3 months (e.g., 5.11, 5.12)
- 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

Choosing a Kernel Version

When building a custom kernel, version selection 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 and minor improvements)

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
- Verify running kernel: `uname -a`

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:

```c
#include <linux/module.h>     /* Needed by all modules */
#include <linux/kernel.h>     /* Needed for KERN_INFO */
#include <linux/init.h>       /* Needed for macros */

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Hello World module");

static int __init hello_init(void)
{
    printk(KERN_INFO "Hello, World!\n");
    return 0;    /* Success */
}

static void __exit hello_exit(void)
{
    printk(KERN_INFO "Goodbye, World!\n");
}

module_init(hello_init);
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:

```makefile
# Define the module name
obj-m := hello.o

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

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

For multiple source files:

```makefile
# Module with multiple source files
hello-y := hello_main.o hello_func.o
obj-m := hello.o

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

# Clean up
clean:
 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:

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

MODULE_LICENSE("GPL");

/* Prototypes */
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

#define DEVICE_NAME "chardev"
#define BUF_LEN 80

/* Global variables */
static int Major;                /* Major number assigned */
static int Device_Open = 0;      /* Is device open? */
static char msg[BUF_LEN];        /* Message for the device */
static char *msg_Ptr;

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

/* Initialization function */
int init_module(void)
{
    Major = register_chrdev(0, DEVICE_NAME, &fops);
    if (Major < 0) {
        printk(KERN_ALERT "Failed to register with %d\n", Major);
        return Major;
    }

    printk(KERN_INFO "Major number assigned: %d\n", Major);
    printk(KERN_INFO "Create device with: 'mknod /dev/%s c %d 0'\n",
           DEVICE_NAME, Major);
    return 0;
}

/* Cleanup function */
void cleanup_module(void)
{
    unregister_chrdev(Major, DEVICE_NAME);
}

/* Device methods */
static int device_open(struct inode *inode, struct file *file)
{
    static int counter = 0;

    if (Device_Open)
        return -EBUSY;

    Device_Open++;
    sprintf(msg, "Called device_open %d times\n", counter++);
    msg_Ptr = msg;
    try_module_get(THIS_MODULE);

    return 0;
}

static int device_release(struct inode *inode, struct file *file)
{
    Device_Open--;
```

Building Installing and Using a Character Device Module

Complete workflow for a character device driver:

```
# 1. Create source file (chardev.c) and Makefile as shown above

# 2. Build the module
make

# 3. Load the module
sudo insmod chardev.ko

# 4. Check kernel messages for major number
dmesg | tail

# 5. Create device node (assuming major number is 250)
sudo mknod /dev/chardev c 250 0
sudo chmod 666 /dev/chardev

# 6. Test reading from the device
cat /dev/chardev

# 7. Check module information in sysfs
ls -l /sys/module/chardev/
cat /sys/module/chardev/parameters/* 2>/dev/null

# 8. Unload the module when done
sudo rmmod chardev

# 9. Clean up the device node
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. Check available I/O schedulers
cat /sys/block/sda/queue/scheduler

# 2. Test performance with default scheduler
echo 3 > /proc/sys/vm/drop_caches  # Clear cache
dd if=/dev/zero of=testfile bs=1M count=1000 oflag=direct
rm testfile

# 3. Change to a different scheduler (e.g., BFQ)
echo bfq > /sys/block/sda/queue/scheduler

# 4. Test performance with new scheduler
echo 3 > /proc/sys/vm/drop_caches  # Clear cache
dd if=/dev/zero of=testfile bs=1M count=1000 oflag=direct
rm testfile

# 5. Test with different I/O priorities
# Start a background write process
dd if=/dev/zero of=bg_file bs=1M count=2000 oflag=direct &
BG_PID=$!

# Run a foreground process with higher priority
ionice -c2 -n0 dd if=/dev/zero of=fg_file bs=1M count=500 oflag=direct

# Clean up
kill $BG_PID
rm bg_file fg_file

# 6. Return to the default scheduler
echo deadline > /sys/block/sda/queue/scheduler
```