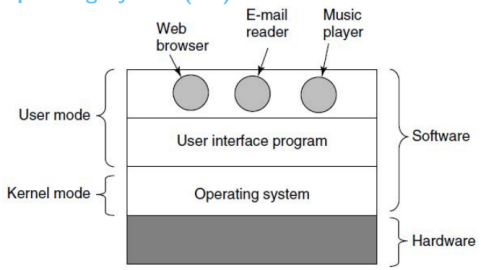
Betriebssysteme

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

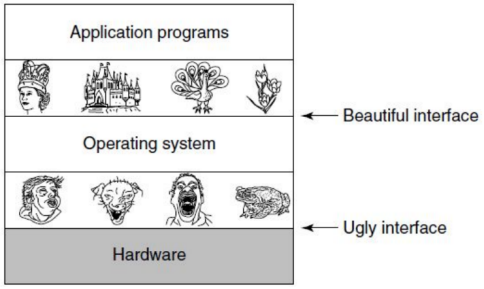
The OS provides two key interfaces:

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

The Operating System as an Extended Machine

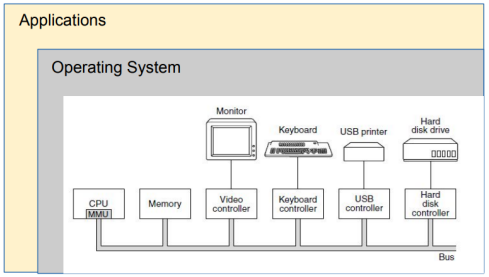
The operating system serves as an abstraction layer between hardware and applications (software that manages hardware)

- 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`

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:

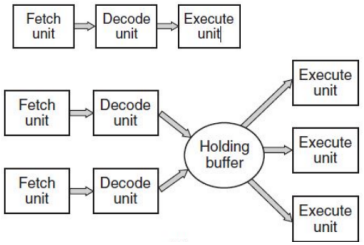
- ARM, RISC, X86, etc.
- Instructions are classified along Execution Privileges, enforced by CPU:
  - Intel: Priority Rings → User Mode (limited set of instructions) vs Kernel Mode (full set of instructions)
  - ARM: UnPrivileged Mode vs Privileged Mode

CPU cycles

Basic cycle of every CPU:

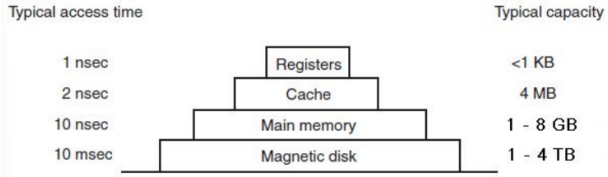
- Fetch instructions from memory into registers
- Decode the instruction to determine type and operands
- Execute the instruction
- Repeat for subsequent instructions

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:

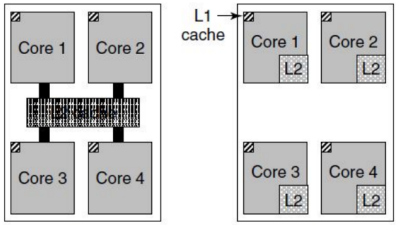
- Registers (fastest, smallest)
- Cache memory (L1, L2, L3)
- Main memory (RAM)
- Secondary storage (disks, SSDs)
- Tertiary storage (backup systems)



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

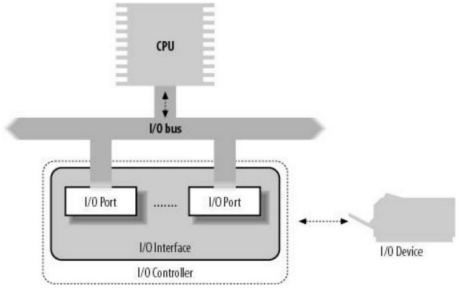
Input/Output (I/O) System

The I/O system comprises:

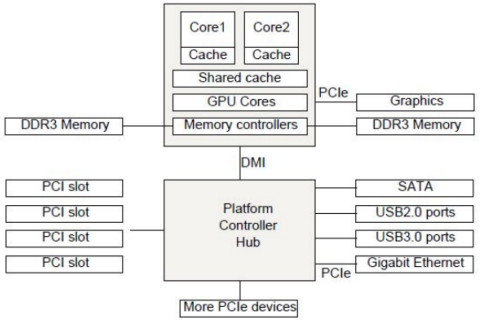
- Various devices (hard drives, network interfaces, serial ports, keyboards, etc.)
- Bus systems to connect devices (PCI, PCIe, SATA, USB, etc.) (software)
- I/O controllers (hardware)
- Device drivers (software)

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:



Linux

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 → 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.

Process 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

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;
}
```

Fork Process Trees

Fork() Process Tree Analysis

Understanding fork() behavior

- fork() creates an exact copy of the calling process
- Returns child PID to parent, 0 to child
- fork() $>$ 0 is true only in parent process
- Each fork() doubles the number of processes

Analysis method

- Draw a process tree starting with the original process
- For each fork(), branch into parent and child
- Track which path each process takes through if/else statements
- Count total processes and specific execution paths

Trace execution paths

- Parent: fork() returns child PID ($>$ 0)
- Child: fork() returns 0
- Follow if-else logic for each process

Counting strategy

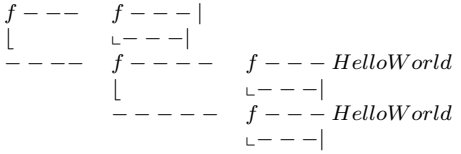
- Start with 1 process (the original)
- Each successful fork() adds 1 new process
- Count processes that reach specific code sections
- Be careful with nested forks and conditional execution

Count results

- Total processes = original + all children created
- Count specific outputs by tracing paths to printf statements

Process Creation Analysis Analyze the following code:

```c
if (fork() > 0)
    fork();
else {
    fork();
    if (fork() > 0)
        printf("Hello World\n");
}
```



Process tree and analysis:

- Original process P forks → creates child C1
- Parent P (fork() $>$ 0): executes second fork() → creates child C2
- Child C1 (fork() == 0): executes fork() → creates child C3
- Child C1: executes second fork() → creates child C4, then prints
- Child C3: executes second fork() → creates child C5, then prints

Results:

- Additional processes created: 5 (C1, C2, C3, C4, C5)
- "Hello World"printed: 2 times (by C1 and C3)

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/dispatched like regular processes
- Uninterruptible (!! 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)

Thread Management in Linux

Creating threads

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

POSIX Thread Programming

Essential POSIX thread functions

- pthread_create(): Create new thread
- pthread_join(): Wait for thread completion
- pthread_exit(): Terminate calling thread
- pthread_detach(): Detach thread (no join needed)

Programming pattern

- Include pthread.h header
- Define thread function with void* signature
- Create threads in loop if multiple needed
- Always join threads to avoid zombies
- Compile with -lpthread flag

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;
}
```

Thread Programming with POSIX

Write a program that creates two threads, both printing "Hello World:

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

void *hello_world(void *arg) {
    printf("Hello World\n");
    return NULL;
}

int main() {
    pthread_t thread[2];

    // Create two threads
    for (int i = 0; i < 2; i++) {
        pthread_create(&thread[i], NULL, hello_world, NULL);
    }

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

    return 0;
}
```

Note: Threads created with pthread_create() start immediately.

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 vs. Thread Differences

Explain the difference between processes and threads:

Solution: The main difference is that processes have their own memory space, while threads (belonging to the same process) share the process memory space.

Processes:

- Independent memory spaces
- Higher creation/switching overhead
- Better isolation and fault tolerance
- Communication via IPC mechanisms

Threads:

- Shared memory space within process
- Lower creation/switching overhead
- Faster communication via shared memory
- Risk of interference between threads

Analyzing Process vs. Thread Questions

Key comparison points

- Memory organization: separate vs. shared address space
- Creation overhead: high vs. low
- Communication methods: IPC vs. shared memory
- Isolation level: strong vs. weak
- Context switching cost: expensive vs. cheap

ADD EXERCISE 1D SEP07 USER VS KERNEL-LEVEL-THREADS

Parallelism and Concurrency Basics

Important Definitions

- **Parallelism:** Multiple processes or threads executing simultaneously
- **Concurrency:** Multiple processes or threads making progress independently
- **Mutual Exclusion:** Ensures that only one thread accesses a resource at a time
- **Synchronization:** Mechanisms to control the order of execution of threads
- **Critical Section:** A part of code that accesses shared resources and must not be executed by more than one thread at a time
- **Deadlock:** A situation where two or more threads are blocked forever, waiting for each other
- **Livelock:** A situation where threads are continuously changing states without making progress
- **Starvation:** A thread is perpetually denied access to a resource it needs for execution
- **Locks and Semaphores:** Tools to manage access to shared resources
- **Condition Variables:** Used for signaling between threads

Famous Problems

- **Producer-Consumer Problem:** Managing a shared buffer between producer and consumer threads
- **Reader-Writer Problem:** Allowing multiple readers or one writer to access a shared resource
- **Dining Philosophers Problem:** Managing resource sharing among multiple threads with potential deadlock
- **Sleeping Barber Problem:** Synchronizing access to a barber shop with limited resources

Process States

Process States (for details see a few pages ago)

- New: Process is being created
- Ready: Process is waiting to be assigned to a processor
- Running: Instructions are being executed
- Waiting: Process is waiting for some event to occur (e.g., I/O completion)
- Terminated: Process has finished execution

Process State Analysis

State identification

- Running: Has CPU, actively executing
- Ready: Waiting for CPU only
- Blocked: Waiting for external event/resource
- Additional states: Swapped, Zombie, Created

Transition triggers

- Timer interrupts (quantum expiration)
- I/O operations (blocking)
- System calls
- Resource availability
- Scheduler decisions

Process State Transitions

Name and explain the three main states processes can transition between:

The three fundamental process states:

- **Running**: Process is currently executing on CPU
- **Ready**: Process is ready to execute, waiting for CPU assignment
- **Blocked/Sleeping**: Process cannot execute, waiting for an event (e.g., I/O completion)

State transitions:

- Ready → Running: Scheduler assigns CPU
- Running → Ready: Time quantum expires or preemption
- Running → Blocked: Process waits for I/O or resource
- Blocked → Ready: Awaited event occurs

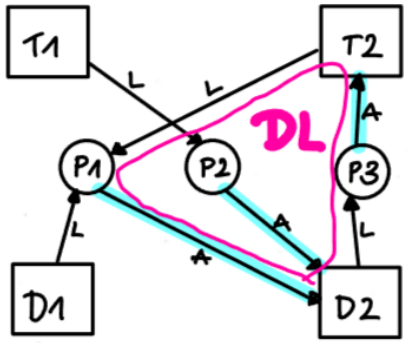
Resource Graphs

Resource Graph Analysis

Ein Rechnersystem besitzt zwei Tapestationen (T1, T2) und zwei Disks (D1, D2). Zur Zeit laufen drei Prozesse (P1, P2, P3), wobei folgendes gilt:

- Prozess P1 kopiert Daten von Disk D1 auf die Tapestation T2 und möchte Daten auf den Disk D2 schreiben
- Prozess P2 hat Tapestation T1 alloziert und möchte Daten auf Disk D2 schreiben
- Prozess P3 hat Disk D2 alloziert und möchte Daten nach Tapestation T2 kopieren

Ist das eine Deadlocksituation, wenn die Ressourcen exklusiv alloziert werden und wenn möchte schreiben das Gleiche wie anfordern bedeutet? Begründen Sie Ihre Antwort (Ressourcengraphen zeichnen und analysieren).



Analyse:

- P1 hält D1, möchte T2 und D2
- P2 hält T1, möchte D2
- P3 hält D2, möchte T2

Es entsteht ein Zyklus: P1 → D2 → P3 → T2 → P1

Bedingungen für Deadlock:

- Mutual Exclusion: Ressourcen werden exklusiv alloziert
- No Preemption: Ressourcen können nicht weggenommen werden
- Hold & Wait: Prozesse halten Ressourcen und warten auf weitere
- Circular Wait: Es gibt einen Zyklus im Ressourcengraph

Alle vier Bedingungen sind erfüllt → Deadlock!

Deadlock Detection in Resource Allocation Graphs

Resource Graph erstellen

- Kreise: Prozesse (P1, P2, P3, ...)
- Rechtecke: Ressourcen (R1, R2, R3, ...)
- Pfeile von Prozess zu Ressource: Reqüst (Anforderung)
- Pfeile von Ressource zu Prozess: Allocation (Zuteilung)

Deadlock-Analyse

- Suche nach Zyklen im Graph
- Ein Zyklus bedeutet Deadlock bei Single-Instance Ressourcen
- Bei Multi-Instance Ressourcen: Prüfe ob alle Instanzen blockiert

Deadlock-Bedingungen prüfen

- Mutual Exclusion: Exklusive Ressourcennutzung
- Hold & Wait: Halten und gleichzeitig warten
- No Preemption: Keine Unterbrechung möglich
- Circular Wait: Zyklische Warteabhängigkeiten

Lösungsansätze

- Prevention: Eine der vier Bedingungen verhindern
- Avoidance: Banker's Algorithm
- Detection & Recovery: Deadlock erkennen und auflösen

Synchronization with Semaphores

Synchronization Problem Solving

- Identify critical sections
- Determine synchronization requirements
- Choose appropriate mechanism (mutex, semaphore, monitor)
- Ensure deadlock avoidance
- Test for race conditions

Semaphores

- Semaphores are synchronization primitives used to control access to shared resources.
- They can be binary (0 or 1) or counting (any non-negative integer).
- Operations:
  - `wait(S)`: Decrement the semaphore S. If S is less than 0, block the process.
  - `signal(S)`: Increment the semaphore S. If S was negative, wake up a blocked process.

Semaphore-basierte Synchronisation

Semaphore verstehen

- Semaphore S ist ein Zähler mit atomaren Operationen
- down(S) oder P(S): Dekrementiert S, blockiert bei $S \leq 0$
- up(S) oder V(S): Inkrementiert S, weckt wartende Prozesse
- Initialisierung bestimmt verfügbare Ressourcen

Synchronisationsmuster

- Mutual Exclusion: Binary Semaphore (0/1) um kritische Bereiche
- Signaling: Ein Prozess signalisiert einem anderen (Producer-Consumer)
- Rendezvous: Zwei Prozesse warten aufeinander
- Barrier: Mehrere Prozesse warten aufeinander

Typische Synchronisationsprobleme

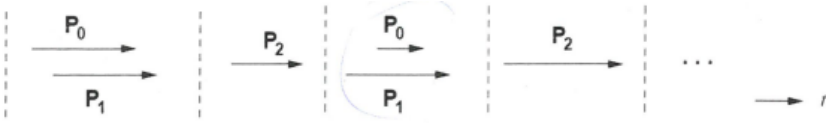
- Producer-Consumer: Buffer-Management mit vollen/leeren Plätzen
- Reader-Writer: Mehrere Leser oder ein Schreiber
- Dining Philosophers: Deadlock-Vermeidung bei zyklischen Abhängigkeiten
- Barrier Synchronization: Alle warten aufeinander

Implementierungsschritte

- Identifiziere Synchronisationspunkte
- Bestimme benötigte Semaphore und deren Initialisierung
- Verwende down() vor kritischen Bereichen/Warten
- Verwende up() nach kritischen Bereichen/Signaling
- Teste auf Deadlock-Freiheit und korrekte Reihenfolge

Semaphore Implementation

Gegeben sind drei Prozess P0, P1, und P2 die nach folgendem Schema abgearbeitet werden soll:



Die Verarbeitung startet mit den beiden Prozessen P0 und P1, die parallel verarbeitet werden sollen (es spielt kein Rolle, welcher der beiden Prozesse zürst mit seiner Verarbeitung beginnt oder aufhört). Wenn beide Prozesse eine Iteration ihrer Funktion working(x) beendet haben, folgt Prozess P2, etc.
Schreiben Sie Pseudocode mit maximal 3 Semaphoren S0, S1 und S2, der garantiert, dass die oben skizzierte Reihenfolge eingehalten wird. Verwenden Sie dazu ausschliesslich Befehle der Form up(S0) und down(S0), etc. Geben Sie an, wie die Semaphore initialisiert werden müssen.

P0	P1	P2
Sem S0: 1 Sem S1: 1 Sem S2: 0		

while(1) {
down(S0);
working(0);
up(S2);
}
	while(1) {
	down(S1);
	working(1);
	up(S2);
	}
		while(1) {
		down(S2);
		working(2);
		up(S0);
		up(S1);
		}

Initialisierung:

- S0 = 1 (P0 kann starten)
- S1 = 1 (P1 kann starten)
- S2 = 0 (P2 muss warten - Synchronisation zwischen P0/P1 und P2)

Ablauf:

- P0 und P1 starten parallel
- Beide signalisieren mit up(S2) wenn fertig
- P2 wartet mit down(S2) bis beide P0 und P1 fertig sind
- P2 gibt mit up(S0) und up(S1) die nächste Runde frei

Producer-Consumer Problem

Producer-Consumer pattern

- Identify shared resource (buffer)
- Determine capacity constraints
- Create semaphores for available/used slots
- Producer: wait(empty), produce, signal(full)
- Consumer: wait(full), consume, signal(empty)

Producer-Consumer Problem

Implement a semaphore-based solution for the producer-consumer problem:

```
1  const int N = 4;        // Maximum buffer size
2  int sem_write = N;      // Semaphore for write
       slots
3  int sem_read = 0;       // Semaphore for read
       slots
4
5  // Producer process
6  while(1) {
7      wait(sem_write);    // Wait for empty slot
8      write(value);       // Write to buffer
9      signal(sem_read);   // Signal data available
10 }
11
12 // Consumer process
13 while(1) {
14     wait(sem_read);     // Wait for data
15     read(value);        // Read from buffer
16     signal(sem_write);  // Signal slot available
17 }
```

Key points:

- sem_write tracks empty buffer slots
- sem_read tracks filled buffer slots
- Producer waits for empty slots, signals filled slots
- Consumer waits for filled slots, signals empty slots

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

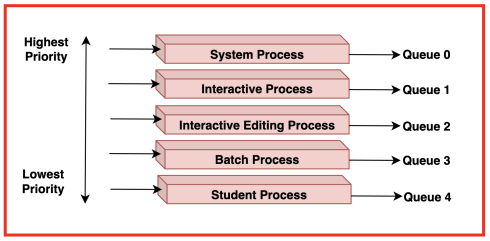
Simple Schedulers

- **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



Fair Share Scheduling

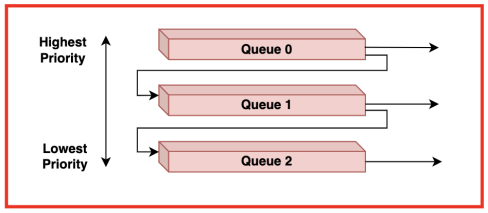
aim for equal CPU time:

- 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

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

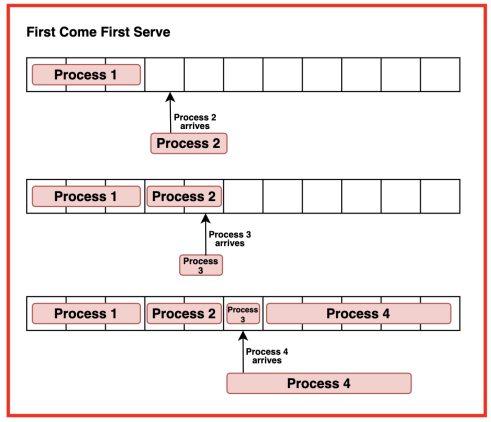


First-In-First-Out (FIFO/FCFS)

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)

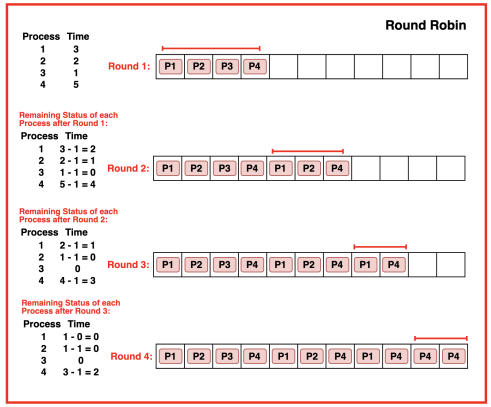
→ also known as First-Come-First-Served (FCFS)



Round Robin (RR)

time-sharing algorithm:

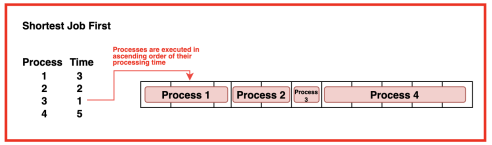
- Each process runs for a fixed time slice (quantum)
- After quantum expires, process is moved to the end of the queue
- Preemptive, as running tasks are interrupted after quantum
- Arriving tasks have priority over adjourned tasks (by convention)
- No starvation, as all processes eventually get CPU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



Shortest Job First (SJF)

non-preemptive scheduling algorithm:

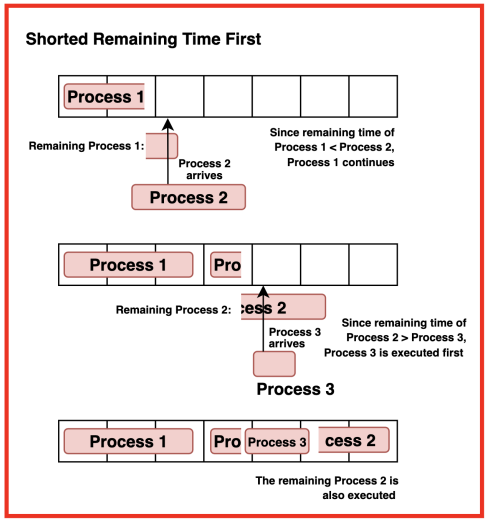
- Selects the process with the shortest burst time
- Minimizes average waiting time
- Non-preemptive, runs to completion
- Can lead to starvation for longer processes
- Optimal for average turnaround time



Shortest Remaining Time (SRT)

preemptive version of SJF:

- At each time unit, selects the process with the shortest remaining time
- Preempts running processes if a new process arrives with shorter remaining time
- Can lead to high context switching overhead
- Optimal for minimizing average waiting time



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.

Scheduling Algorithm Analysis

Scheduling Algorithm Analysis

FIFO/FCFS approach

- Execute processes in arrival order
- Non-preemptive
- Simple but may cause long wait times

SJF approach

- At decision point, choose process with shortest total time
- Non-preemptive
- Optimal for average waiting time

SRT approach

- At each time unit, choose process with shortest remaining time
- Preemptive version of SJF
- May cause frequent context switches

Problem-solving steps

- Create timeline showing all events (arrivals, completions)
- For each algorithm, trace through decision points
- Calculate metrics: turnaround time, waiting time, response time

Round Robin Analysis

Quantum size considerations

- Too large: Approaches FIFO, poor response time
- Too small: High context switching overhead
- Optimal: Slightly larger than typical interaction time

Performance factors

- Context switch overhead
- Interactive response requirements
- CPU-bound vs. I/O-bound process mix
- System throughput vs. responsiveness trade-off

Round Robin Scheduling Why should the time quantum be larger than typical interaction time?
Reason: If the quantum is larger than interaction processing time, the entire user interaction can be completed within one quantum. This ensures the system responds quickly to user input without interrupting the interactive process.
If the quantum is too small, interactive processes get interrupted before completing their response, leading to poor user experience.

FIFO, SJF, and SRT Scheduling Given processes with arrival and burst times, determine execution order:

Process	Arrival T.	Burst T.
P	0	10
Q	4	5
R	5	10
S	6	1

Zeit	0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9 10	0 1 2 3 4 5 6 7 20
FCFS	P P P P P P P P P P	Q Q Q Q Q R R R R R	R R R R S
SJF	P P P P P P P P P P	S Q Q Q Q Q R R R R	R R R R R R
SRT	P P P P Q Q S Q Q Q	P P P P P P R R R R	R R R R R R

Multi-level Scheduling

Priority-Based Multi-Level Scheduling

Organize by priority levels

- Group processes by priority (higher number = higher priority)
- Maintain separate ready queue for each priority level
- Always schedule from highest priority non-empty queue

Apply Round Robin within priority

- Use time quantum for processes at same priority level
- Move process to end of same priority queue when quantum expires
- New arrivals join appropriate priority queue

Handle preemption

- Higher priority process always preempts lower priority
- Preemption occurs immediately when higher priority task arrives
- Preempted task returns to head of its priority queue

Priority-Based Scheduling Issues

- Priority inversion
- Starvation of low-priority processes
- Priority assignment difficulties
- Real-time deadline misses

Solution approaches:

- Priority inheritance protocols
- Aging mechanisms
- Fair-share scheduling
- Deadline-based scheduling

Priority inversion occurs when:

- High-priority process waits for low-priority process
- Medium-priority processes preempt low-priority process
- High-priority process effectively blocked by medium-priority processes

Prevention methods:

- **Priority inheritance**: Low-priority process temporarily inherits high priority
- **Priority ceiling**: Resources assigned maximum priority of potential users
- **Dynamic priority adjustment**: Increase priority of waiting processes over time

Multi-Level Round Robin Scheduling

Multi-level principles

- Separate queues for different priority levels
- Higher priority queues are served first
- Round robin within each priority level
- Lower priority tasks only run when higher queues are empty

Timeline construction

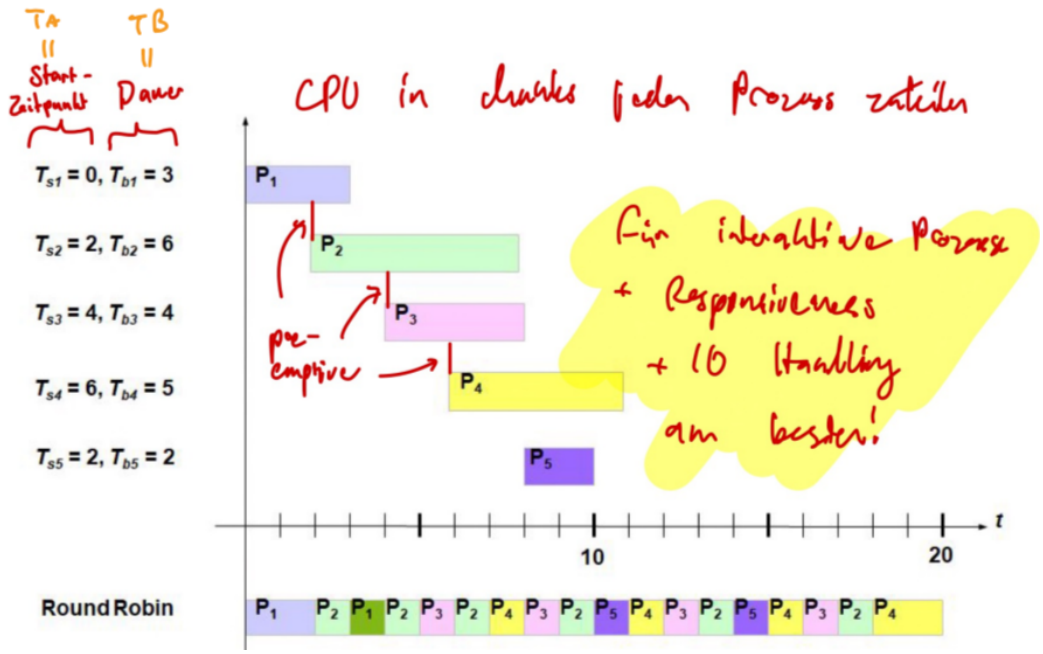
- Mark process arrivals on timeline
- At each time unit, determine which process should run
- Apply round robin within the highest non-empty priority queue
- Show context switches and queue changes

Scheduling algorithm

- Group processes by priority level
- Always check highest priority queue first
- Use round robin with specified time quantum within each queue
- Preemption occurs when higher priority process arrives
- Track remaining execution time for each process

Multi-Level Scheduling with RR Five processes with time quantum = 1:

Process	Priority	Arrival	Execution
P1	0	0	4
P2	1	1	3
P3	1	2	2
P4	0	5	3
P5	0	7	2



Execution timeline:

- t=0-1: P1 (only process available)
- t=1-2: P2 (higher priority, preempts P1)
- t=2-3: P3 (same priority as P2, P2's quantum expired)
- t=3-4: P2 (continues RR at priority 1)
- t=4-5: P3 (completes)
- t=5-6: P2 (completes), then P4 starts
- t=6-7: P1 (RR at priority 0)
- t=7-8: P4 (continues RR), P5 arrives
- And so on with RR between P1, P4, P5 at priority 0...

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

Rate Monotonic Scheduling Given tasks with the following characteristics:

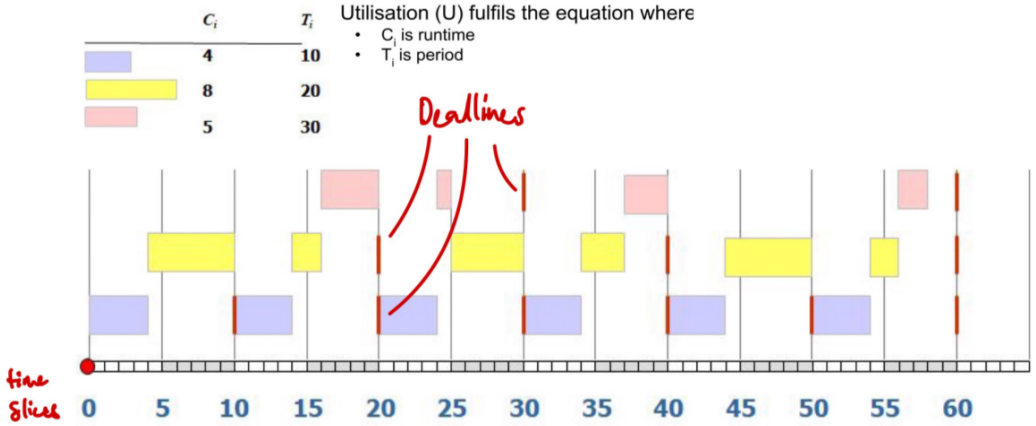
Task	WCET (C)	Period (T)
T1	10	20
T2	10	50
T3	5	30

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.

Rate Monotonic Scheduling Example



Earliest Deadline First Scheduling

EDF Algorithm principles

- Always schedule the task with the earliest deadline
- Preemption occurs when a task with earlier deadline arrives
- Re-scheduling happens at specified intervals or task completion

Scheduling steps

- Calculate all task deadlines for the scheduling period
- At each scheduling point, identify available tasks
- Select task with earliest deadline
- Handle ties with specified priority rules (e.g., shorter period)
- Mark execution periods in timeline diagram

Timeline construction

- Create timeline with scheduling intervals marked
- For each interval, determine which task should run
- Show task execution as filled blocks
- Verify all deadlines are met

EDF Details

Calculating absolute deadlines

- For each task instance: Deadline = Arrival time + Period
- Track all active tasks (arrived but not completed)
- Update deadlines when new instances arrive

Scheduling by earliest deadline

- At each scheduling point, select task with earliest absolute deadline
- Preempt running task if new arrival has earlier deadline
- Use period as tiebreaker for equal deadlines (shorter period wins)

Handling rescheduling points

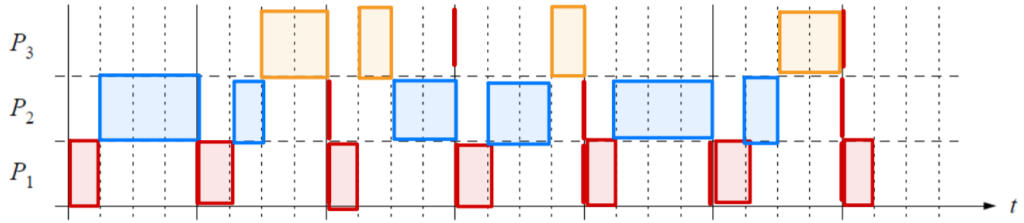
- Reschedule when task completes or suspends
- Reschedule at fixed intervals if specified
- Reschedule when new task arrives (if preemptive)

EDF Scheduling Three periodic tasks with rescheduling every 4ms or when a task is suspended:

Task	Period	Execution Time
T1	4ms	1ms
T2	8ms	4ms
T3	12ms	3ms

Priority: in case of equal deadlines, shorter period wins.

Preemptive Diagram:



Timeline analysis (first 24ms):

- t=0: T1(dl=4), T2(dl=8), T3(dl=12) arrive → Schedule T1
- t=1: T1 completes → Schedule T2
- t=4: T1(dl=8) arrives, T2 continues (dl=8, but T2 started first)
- t=5: T1 executes (same deadline, shorter period wins)
- t=6: T1 completes → Schedule T2
- t=8: T1(dl=12), T2(dl=16) arrive, T3 continues
- And so on...

The schedule ensures all deadlines are met with this task set.

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

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]`

Complexity Analysis

Algorithm Complexity Analysis

Time complexity evaluation

- O(1): Constant time, independent of input size
- O(log n): Logarithmic time, scales well
- O(n): Linear time, proportional to input size
- Identify bottleneck operations

Scheduler efficiency factors

- Process selection time
- Context switch overhead
- Load balancing cost
- Priority calculation complexity

Linux O(1) Scheduler

Explain why the Linux O(1) scheduler is called O(1):

O(1) refers to constant time complexity:
- Time to find next process is independent of total number of processes
- Uses active and expired arrays with 140 priority levels each
- Bitmap indicates which priority levels have waiting processes
- Simply finds highest priority bit set and takes first process from that queue

Algorithm:
- Check bitmap for highest priority with waiting processes
- Take first process from that priority queue
- Constant time regardless of system load

Resource Control in Linux

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:
- **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 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:

```
# 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

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:

```
# 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.

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
```

System Call Analysis

Key concepts

- fork() duplicates current process
- exec() family replaces current process image
- Code after exec() only runs if exec() fails
- Count processes by tracing fork() calls

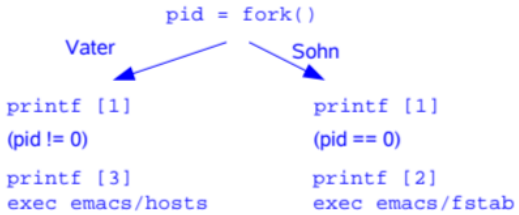
Analysis steps

- Draw execution tree showing parent/child branches
- Mark where each printf() executes
- Identify exec() calls that replace process image
- Determine unreachable code after successful exec()

System Call Analysis Analyze the output of a program using fork() and exec():

```c
int pid = fork();
printf("%s\n", "[1] Time for case distinction");
if (pid) {
    printf("%s\n", "[2] Starting emacs/fstab");
    execl("/bin/emacs", "/etc/fstab", (char *)NULL);
} else {
    printf("%s\n", "[3] Starting emacs/hosts");
    execl("/bin/emacs", "/etc/hosts", (char *)NULL);
}
printf("%s\n", "[4] Two editors started successfully");
// More code follows...
```

Analysis:
Programm startet mit 2 Editoren:



- Two editors are started
- Output: [1], [1], [2], [3]
- The code after execl() calls is never reached
- execl() replaces the process image, so [4] is never printed

Execution flow:

- fork() creates parent and child
- Both print [1]
- Parent (pid $\neq$ 0) prints [2], then execl() replaces it with emacs
- Child (pid == 0) prints [3], then execl() replaces it with emacs

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

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)

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

Free Space Management

Free space management methods

- Bitmap: Good for uniform access, fixed size
- Linked list: Simple but slow random access
- Grouping: Combines free blocks into groups
- Counting: Stores (address, count) pairs

Evaluation criteria

- Space efficiency
- Access speed
- Implementation complexity
- Fragmentation handling
- Main memory requirements

File System Free Space Management

Describe two efficient methods for tracking free blocks:

Method 1: Bitmap

- Each bit represents one block
- 0 = free, 1 = allocated (or vice versa)
- Fast scanning for free blocks
- Compact representation
- Requires main memory for efficiency

Method 2: Free block list with (start, length) tuples

- List of contiguous free block ranges
- Each entry: (starting block, number of blocks)
- Efficient for large contiguous areas
- Dynamic size based on fragmentation

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)

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

Buddy System Fragmentation

Determine buddy block sizes

- For each allocation request, find smallest power-of-2 block that fits
- Block size = $2^{\lceil \log_2(\text{request size}) \rceil}$

Calculate internal fragmentation

- Fragmentation per block = Block size - Requested size
- Total fragmentation = Sum of all individual fragmentations

Account for merging

- When blocks are freed, buddies may merge
- Track which blocks remain allocated vs. freed

Buddy System

Ein Betriebssystem-Kernel verwaltet seine Datenbuffer mit einem Buddy System, wobei insgesamt 8MByte Speicher zur Verfügung stehen. Zur Zeit sind folgende Buffer mit 62KByte, 34KByte und 9KByte alloziert worden. Wie viel Speicher geht dabei durch interne Fragmentierung insgesamt verloren, Angabe in KByte:

Buddy System alloziert in Potenzen von 2:

- 62 KByte → nächste 2er-Potenz: 64 KByte
- 34 KByte → nächste 2er-Potenz: 64 KByte
- 9 KByte → nächste 2er-Potenz: 16 KByte

Interne Fragmentierung = Alloziert - Angefordert:

- (64 - 62) = 2 KByte
- (64 - 34) = 30 KByte
- (16 - 9) = 7 KByte

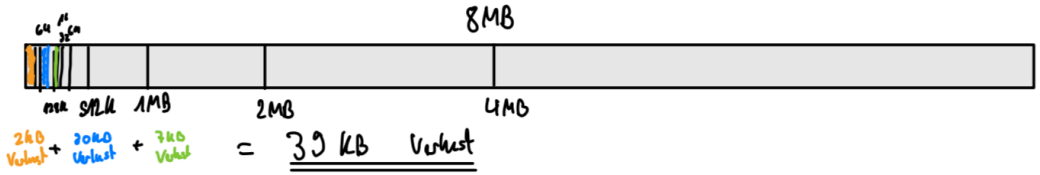
Gesamt: 2 + 30 + 7 = **39 KByte** interne Fragmentierung

Buddy System Analysis

8MB system allocates buffers: 62KB, 34KB, 9KB

Allocation analysis:

- 62KB → 64KB block (internal fragmentation: 2KB)
- 34KB → 64KB block (internal fragmentation: 30KB)
- 9KB → 16KB block (internal fragmentation: 7KB)
- Total internal fragmentation: 2 + 30 + 7 = 39KB



The remaining memory is still available for allocation but in specific block sizes according to the buddy system structure.

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

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

Address Translation logical ⇆ physical

Virtual memory requires translation between logical and pysical 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

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 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

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 Table Calculations

Extract address components

- Page Directory bits determine max number of page tables
- Page Number bits determine entries per page table
- Offset bits determine page size: $2^{\text{offset bits}}$

Calculate system limits

- Page size = $2^{\text{offset bits}}$ bytes
- Max page tables = $2^{\text{directory bits}}$
- Entries per page table = $2^{\text{page number bits}}$
- Max frames = Max page tables × Entries per table

Page Tables

Ein Prozessor besitzt eine Wortbreite von 32Bit. Pointer (Adressen) werden in 32Bit Worten gespeichert, aber nur die 24 tieferwertigen Bits werden für die Adressbildung verwendet (Bits 25-31sind auf 0 gesetzt).
Die logische Adresse ist wie folgt strukturiert:

6-Bit Page Directory	8-Bit Page Nummer	10-Bit Offset

a) Wie gross ist eine Page, Angabe in KBytes?
b) Wie viele Bytes enthält das Page Directory, wenn pro Eintrag ein Pointer (Adresse) auf eine Page Tabelle eingetragen wird, Angabe in KBytes?
c) Wie viele Page Tabellen kann ein Prozess maximal haben?
d) Wie viele Frames kann das System maximal haben?

Lösung:
a) Page-Grösse = 2^{10} Bit = 1024 Bytes = **1 KByte**
b) Page Directory:

- 6 Bit → 2^6 = 64 Einträge
- 4 Bytes pro Adresse
- 64 × 4 = 256 Bytes = **0.25 KBytes**

c) Page Tabellen: 6 Bit → 2^6 = **64 Page Tabellen**
d) Frames im System:

- 24 Bit physische Adresse
- 10 Bit Offset pro Frame
- Frame-Bits: 24 - 10 = 14 Bit
- Maximale Frames: 2^{14} = **16384 Frames**

Page Replacement

Page Replacement Algorithms

Least Recently Used (LRU)

- Ersetze die Page, die am längsten nicht verwendet wurde
- Gute Performance, aber aufwändig zu implementieren
- Benötigt Zeitstempel oder Zugriffs-Historie
- Approximation durch Clock Algorithm oder Second Chance

First-In-First-Out (FIFO)

- Ersetze die älteste Page (first loaded)
- Einfach zu implementieren mit Queue
- Kann zu Belady's Anomaly führen
- Nicht optimal, da alte Pages oft noch verwendet werden

Optimal (OPT)

- Ersetze Page, die am spätesten wieder verwendet wird
- Theoretisch optimal, praktisch nicht implementierbar
- Benötigt Zukunftswissen über Page-Zugriffe
- Wird als Benchmark für andere Algorithmen verwendet

Implementation Tips

- Page Fault tritt auf bei erstem Zugriff auf neü Page
- Bei mehreren Kandidaten: Wähle Frame mit kleinster Nummer
- Clock Algorithm: Circular list mit Reference Bit
- Working Set: Berücksichtige lokale vs. globale Ersetzung

Page Replacement Analysis

Algorithm comparison factors

- Page fault frequency
- Implementation complexity
- Hardware support requirements
- Performance under different workloads
- Memory overhead for bookkeeping

Common algorithms

- FIFO: Simple, poor performance
- LRU: Good performance, moderate complexity
- Clock: Approximates LRU, hardware efficient
- Optimal: Theoretical best, not implementable

Page Replacement Algorithms

Compare LRU and FIFO page replacement algorithms:

LRU (Least Recently Used) typically causes fewer page faults because:

- Exploits locality principle
- Recently accessed pages likely to be accessed again soon
- Keeps "hot"pages in memory longer
- Better prediction of future access patterns

FIFO (First In, First Out):

- Simple implementation
- No consideration of access patterns
- May replace frequently used pages
- Can suffer from Belady's anomaly

Optimal algorithm (theoretical):

- Replace page that will be accessed furthest in future
- Not implementable (requires future knowledge)
- Used as performance benchmark

LRU Page Replacement

Track page access order

- Maintain timestamp or access order for each page in memory
- On page fault, identify least recently used page
- Replace LRU page with new page

Handle page faults

- Mark page fault when referenced page not in memory
- Load new page into frame of LRU page
- Update access timestamps for all affected pages

Apply demand paging

- First access to any page is compulsory miss
- Subsequent accesses depend on memory capacity and access pattern

Least Recently Used (LRU)

Ein Prozess referenziert der Reihe nach folgende Pages:

8 7 5 8 7 3 5 1 3 4 2 1 8 3 1 2

Gehen Sie davon aus, dass zu Beginn keine Pages im Speicher stehen und dass auch das erstmalige Laden einer Page als Page Fault gezählt wird (demand paging). Pro Prozess stehen 4 Frames zur Verfügung.
Tragen Sie in unten stehender Tabelle die den Frames zugewiesenen Pages für den Least Recently Used Algorithmus. Das Page mit diesem Zugriff am weitesten zurückliegend wird dann mit ein neüs Page ersetzt.
Markieren Sie die Spalten mit einem Stern, wo ein Page Fault auftritt. Nehmen Sie an, dass ausschliesslich Demand Paging verwendet wird. Wenn mehrere Frames für das placement resp. replacement in Frage kommen, muss der Frame mit der kleinsten Nummer gewählt werden.

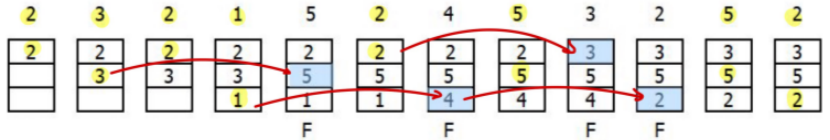
Lösung:

Referenzen	8	7	5	8	7	3	5	1	3	4	2	1	8	3	1	2
frame 1	8*	8	8	8	8	8	8	1*	1	1	1	1	1	1	1	1
frame 2		7*	7	7	7	7	7	7	7	4*	4	4	8*	8	8	8
frame 3			5*	5	5	5	5	5	5	5	2*	2	2	2	2	2
frame 4						3*	3	3	3	3	3	3	3	3	3	3
page fault	×	×	×			×		×		×	×		×			

Page Faults: 8 von 16 Zugriffen
Compulsory Page Faults: die ersten 4 (8, 7, 5, 3) sind unvermeidlich, da die Pages noch nicht im Speicher sind.

LRU-Logik:

- Bei jedem Zugriff wird die "letzte Verwendung"der Page aktualisiert
- Bei einem Page Fault wird die am längsten nicht verwendete Page ersetzt
- Zeitstempel oder Zugriffs-Historie bestimmen LRU-Page



Virtual Memory Addressing

Virtual Memory Address Analysis

Address structure calculation

- Page offset bits $= log_2$(page size in bytes)
- Remaining bits = total address bits - offset bits
- Bits per level = remaining bits $\div$ number of levels

Page table size calculation

- Entries per table $= 2^{\text{bits per level}}$
- Table size = entries $\times$ entry size in bytes
- Number of tables per level = cumulative from higher levels

Common mistakes to avoid

- Forgetting to account for page offset bits
- Mixing up table levels in size calculations
- Not considering that tables should fit in pages

Virtual Memory Address Translation Given: 16KB page size, 47-bit virtual addresses, 3-level paging, 8-byte page table entries.
How is a virtual address structured?

Calculation:

- Page size: 16KB $= 2^{14}$ bytes $\rightarrow$ 14 bits for page offset
- Remaining bits: 47 - 14 = 33 bits for page table indexing
- 3 levels: 33 $\div$ 3 = 11 bits per page table level

Address structure:

Bits 46-36	Bits 35-25	Bits 24-14	Bits 13-0
Level 1 (11 bit)	Level 2 (11 bit)	Level 3 (11 bit)	Offset (14 bit)

Page table sizes:

- Level 1: 1 table with $2^{11} = 2048$ entries
- Level 2: 2048 tables with 2048 entries each
- Level 3: $2048^2 = 4{,}194{,}304$ tables with 2048 entries each
- Each table: 2048 $\times$ 8 bytes = 16KB (exactly one page)

ADD EXERCISE 12 SEP07 SEGMENTATION

Memory Management Analysis

Buddy System

- Alloziert in Potenzen von 2 (1, 2, 4, 8, 16, ... KByte)
- Interne Fragmentierung = Alloziert - Angefordert
- Nächste grössere 2er-Potenz finden: $2^n \geq$ Anforderung
- Beispiel: 33 KByte $\rightarrow$ 64 KByte (2^6)

Page Table Berechnung

- Page-Grösse = $2^{\text{Offset-Bits}}$ Bytes
- Anzahl Pages = $2^{\text{Page-Number-Bits}}$
- Page Directory Grösse = $2^{\text{Directory-Bits}} \times$ Pointer-Grösse
- Maximale Frames = $2^{(\text{Physische-Adress-Bits - Offset-Bits})}$

Addressaufteilung

- Logische Adresse = Page Directory + Page Number + Offset
- Physische Adresse = Frame Number + Offset
- Page Table Entry enthält Frame Number
- Translation: Page Number $\rightarrow$ Frame Number

Memory Hierarchy

- Page Directory zeigt auf Page Tables
- Page Tables zeigen auf Frames
- Mehrstufige Page Tables reduzieren Speicherbedarf
- TLB cached häufig verwendete Translations

Memory Access Time Calculation

Average Memory Access Time

Calculation method

- Start from L1 cache (always accessed first)
- Apply formula from innermost level outward
- For each miss, calculate probability of accessing next level (e.g. L3 miss probability = $(1-h_1) \times (1-h_2) \times (1-h_3)$)
- Multiply access time by probability of reaching that level
- Sum all weighted access times

Formula

- $T_{avg} = T_{L1} + (1-h_1) \times T_{L2} + (1-h_1)(1-h_2) \times T_{L3} + \ldots$
- Convert clock cycles to nanoseconds: Time = Cycles $\times$ Clock period (or Cycle time)
- Clock period = 1 / Frequency

Memory Access Time Calculation 2 GHz processor (0.5 ns cycle), 90% hit rate on all levels:

L1 Cache	4 cycles = 2.0 ns
L2 Cache	10 cycles = 5.0 ns
L3 Cache	40 cycles = 20.0 ns
Main Memory	60 ns

Calculation:

- L1 access: $4 \times 0.5 = 2$ ns (always accessed)
- L2 access: $10 \times 0.5 = 5$ ns (10% probability)
- L3 access: $40 \times 0.5 = 20$ ns (1% probability)
- Main memory: 60 ns (0.1% probability)
- $T_{avg} = 2 + 0.1 \times 5 + 0.01 \times 20 + 0.001 \times 60$
- $T_{avg} = 2 + 0.5 + 0.2 + 0.06 = 2.76$ ns

Cache Performance Analysis

TODO: add theory and example!!

Cache Hit Rate Calculation

Problem identification

- Given: Array access pattern, processor architecture, cache block size
- Find: Cache hit rate during sequential array access

Given information analysis

- Identify data type size (e.g., int = 4 bytes on 32-bit system)
- Note cache line size (e.g., 64 bytes)
- Determine access pattern (sequential vs. random)

Solution approach

- Calculate element (data type) size based on processor architecture
- Determine how many array elements fit in one cache line
- For sequential access: First access to cache line = miss, rest = hits

Key formulas

- Elements per cache line = Cache line size / Element size
- Hit rate = Number of hits / Total accesses
- For 32-bit processor: int = 4 bytes
- For 64-bit processor: int = 4 bytes, long = 8 bytes

Cache Access Analysis Given code accessing an integer array sequentially:

```
#define N (10*1000*1000)
int arVL[N];
for (int i = 0; i < N; i++) {
    sum += arVL[i];
}
```

Calculate hit rate on a 32-bit processor with 64-byte cache lines.

Analysis:

- 32-bit system (processor): int = 4 bytes
- Cache line size: 64 bytes
- Elements per cache line: 64/4 = 16 integers
- Access pattern: Sequential $\rightarrow$ first access per line: cache miss
- For every 16 accesses: 1 miss + 15 hits
- Hit rate: $h_c = 15/16 = 0.9375 = 93.75\%$

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:

```
# 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

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

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 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`

I/O Principles

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

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

System Calls for I/O

I/O System Calls

System calls provide the interface between user space and kernel space for I/O operations:

- `open()`: Open a device file
- `read()`: Read data from a device
- `write()`: Write data to a device
- `ioctl()`: Device-specific control operations
- `close()`: Close a device file

These calls allow applications to interact with devices in a standardized way.

System Call Identification

Analyze operation type

- I/O operations: Screen output, file access, network communication
- Resource management: Memory allocation, time delays
- Process control: Process creation, inter-process communication

Distinguish user vs kernel operations

- User mode: Arithmetic, variable manipulation, function calls
- Kernel mode: Hardware access, system resource management
- Mode switch required for system calls

System Call Requirements Which operations require system calls?

- √ Screen output (I/O operation)
- X Variable comparison (CPU operation)
- √ Sleep for 1 second (timer/scheduler)
- X Integer increment (CPU operation)
- X Float increment (CPU operation)

Rule of thumb: Operations requiring hardware access, OS services, or resource management need system calls. Pure computational operations do not.

Interrupts and Interrupt Handling

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

Interrupt System Analysis

Interrupt vs. polling comparison

- Efficiency: Interrupts save CPU cycles
- Responsiveness: Interrupts provide better response time
- Complexity: Polling is simpler to implement
- Resource usage: Interrupts minimize waste

Interrupt handling strategies

- Nested vs. non-nested interrupts
- Priority schemes
- Top-half/bottom-half division (Linux)
- Maskable vs. non-maskable interrupts

Interrupt Handling Explain alternatives to interrupts and why interrupts improve performance:

Alternative: Polling

- OS continuously checks device status
- Wastes CPU time on unnecessary checks
- Creates busy-waiting loops

Interrupt advantages:

- CPU only responds when device needs attention
- Eliminates wasteful polling loops
- Allows CPU to focus on productive work
- Enables asynchronous I/O operations

Interrupt handling strategies:

- Non-nested: Disable interrupts during handling (simple but may miss urgent interrupts)
- Priority-based: Higher priority interrupts can preempt lower priority ones (complex but responsive)

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)

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

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

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

/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

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)
```

Linux Kernel and Device Drivers

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

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.)

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 adn 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)

Kernel Module Building Process

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

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`

Kernel Module Makefile

Module Makefile Example Makefile for a kernel module:

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

For multiple source files:

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

Makefile Dependency Analysis

Understanding make behavior

- Make compares timestamps of targets and dependencies
- Target is rebuilt if it's older than any dependency
- Missing targets are always built
- Implicit rules (.c.o:) handle automatic compilation

Analysis steps

- Identify target dependencies from makefile
- Check file timestamps in directory listing
- Determine which object files need recompilation
- Follow linking rules to determine final executable name

Common patterns

- .c.o: rule compiles .c files to .o files automatically
- $@ refers to the target name
- If .c file is newer than .o file, recompilation occurs
- Missing .o files are always created

Makefile Analysis Given Makefile:

```
CFL = -g
CMP = gcc $(CFL)
app: main1.o mythread.o scheduler.o queues.o mylist.o
 $(CMP) main1.o mythread.o scheduler.o queues.o mylist.o -o $@.e
.c.o:
 $(CMP) -c $<
```

Directory listing shows:

- mylist.c: Feb 24 09:31 (newer)
- mylist.o: Feb 24 09:21 (older)
- queues.c: Feb 24 09:25 (newer)
- queues.o: Feb 24 09:21 (older)
- All other .o files are newer than corresponding .c files

Solution:

- Files to be compiled: mylist.c, queues.c (because .c files are newer than .o files)
- Executable name: app.e (from $@.e where $@ = app)

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

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. 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

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
};
int init_module(void) { /* Initialization function */
    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: 'mknod /dev/%s c %d 0'\n", DEVICE_NAME, Major);
    return 0;
}
void cleanup_module(void) { /* Cleanup function */
    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--;
    module_put(THIS_MODULE);
    return 0;
}
static ssize_t device_read(struct file *filp, char *buffer, size_t length, loff_t
    *offset) {
    int bytes_read = 0;
    if (*msg_Ptr == 0) return 0; /* End of message */
    while (length && *msg_Ptr) { /* Transfer data to user space */
        put_user(*(msg_Ptr++), buffer++);
        length--;
        bytes_read++;
    }
    return bytes_read;
}
static ssize_t device_write(struct file *filp, const char *buff, size_t len, loff_t
    *off) {
    printk(KERN_ALERT "Operation not supported\n");
    return -EINVAL;
}
```

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
```

File Systems

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:

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

int main() {
    int fd;
    char buffer[100];
    ssize_t bytes_read;

    // Open file for reading
    fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
        perror("Error opening file");
        return 1;
    }

    // Read from file
    bytes_read = read(fd, buffer, sizeof(buffer) - 1);
    if (bytes_read > 0) {
        buffer[bytes_read] = '\0';
        printf("Read: %s\n", buffer);
    }

    // Close file
    close(fd);

    return 0;
}
```

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

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

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

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)

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`

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`

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`

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

example in chapter memory management

Build System Analysis

Make Dependency Analysis

Examine file timestamps

- List all source files (.c) and object files (.o)
- Compare timestamps: source newer than object = recompile needed
- Check dependency requirements in Makefile

Apply make rules

- Target depends on all listed prerequisites
- Suffix rule (.c.o:) applies to C source compilation
- Variable substitution: $@ = target name, $< = first prerequisite

Determine final executable name

- Follow target name and any modifications (e.g., $@.e)
- Check for explicit output naming in link command

Make Analysis Given Makefile and file listing:

```
# Makefile
CFL = -g
CMP = gcc $(CFL)
app: main1.o mythread.o scheduler.o queues.o mylist.o
    $(CMP) main1.o mythread.o scheduler.o queues.o mylist.o -o $@.e
.c.o:
    $(CMP) -c $<

# File listing (key timestamps)
# mylist.c    Feb 24 09:31  (source)
# mylist.o    Feb 24 09:21  (object)
# queues.c    Feb 24 09:25  (source)
# queues.o    Feb 24 09:21  (object)
```

Analysis:

- Files to recompile: mylist.c and queues.c (sources newer than objects)
- Final executable: app.e (from target äpp"+ ".eëxtension)
- Other .o files are up-to-date, no recompilation needed

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:

```
# Create a test file
echo "Hello, World!" > testfile.txt
# Check inode number
ls -i testfile.txt
# Create a hard link
ln testfile.txt hardlink.txt
# Check that both files have the same inode
ls -i testfile.txt hardlink.txt
# Check link count
ls -l testfile.txt
# Create a symbolic link
ln -s testfile.txt symlink.txt
# Check inode numbers (symlink has different inode)
ls -i testfile.txt hardlink.txt symlink.txt
# Check what symlink points to
ls -l symlink.txt
# Delete the original file
rm testfile.txt
# Hard link still works
cat hardlink.txt

# Symbolic link is broken
cat symlink.txt
# Output: cat: symlink.txt: No such file or directory

# Clean up
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:

```
# Show filesystem usage
df -h
# Show inode usage
df -i
# Display filesystem type
lsblk -f
# Check filesystem properties
tune2fs -l /dev/sda1 | head -20
# Find large files
find /home -type f -size +100M 2>/dev/null | head -10
# Check filesystem fragmentation (ext2/3/4)
e2fsck -fn /dev/sda1
# Monitor filesystem activity
iotop
# Check for filesystem errors in logs
journalctl -u systemd-fsck* --no-pager
```

Advanced Storage Management

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:

```
# Create a directory for tmpfs mount
mkdir -p /mnt/tmp

# Mount tmpfs with 2GB size limit
sudo mount -t tmpfs -o size=2G tmpfs /mnt/tmp

# Check filesystem usage
df -h /mnt/tmp

# Test performance
dd if=/dev/zero of=/mnt/tmp/speedtest bs=10M count=100

# Compare with disk-based filesystem
dd if=/dev/zero of=/tmp/speedtest bs=10M count=100

# Unmount when done
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:

```
# Create BtrFS filesystem on a single device
sudo mkfs.btrfs /dev/sdb1
# Mount the filesystem
sudo mount /dev/sdb1 /mnt/btrfs
# Create subvolumes
sudo btrfs subvolume create /mnt/btrfs/home
sudo btrfs subvolume create /mnt/btrfs/var
# List subvolumes
sudo btrfs subvolume list /mnt/btrfs
# Create a snapshot of a subvolume
sudo btrfs subvolume snapshot /mnt/btrfs/home /mnt/btrfs/home-snapshot
# Add another device to the filesystem
sudo btrfs device add /dev/sdc1 /mnt/btrfs
# Balance data across devices
sudo btrfs balance start -draid1 -mraid1 /mnt/btrfs
# Check filesystem status
sudo btrfs filesystem show /mnt/btrfs
sudo btrfs filesystem usage /mnt/btrfs
# Enable compression
sudo mount -o remount,compress=zlib /mnt/btrfs
# Defragment with compression
sudo btrfs filesystem defragment -czlib /mnt/btrfs/home
# Check for errors
sudo btrfs scrub start /mnt/btrfs
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 ßhadow"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:

```
# Create directory structure
mkdir -p /tmp/overlay/{lower,upper,work,merged}
# Populate lower directory (read-only base)
echo "Original file from lower" > /tmp/overlay/lower/file1.txt
echo "Lower layer file" > /tmp/overlay/lower/file2.txt
# Create the overlay mount
sudo mount -t overlay overlay \ -o
    lowerdir=/tmp/overlay/lower,upperdir=/tmp/overlay/upper,workdir=/tmp/overlay/work
    \ /tmp/overlay/merged

# View merged contents
ls -la /tmp/overlay/merged/
# Modify a file (goes to upper layer)
echo "Modified content" > /tmp/overlay/merged/file1.txt
# Create a new file (goes to upper layer)
echo "New file in upper" > /tmp/overlay/merged/file3.txt
# Check what's in each layer
echo "=== Lower layer ==="
ls -la /tmp/overlay/lower/
echo "=== Upper layer ==="
ls -la /tmp/overlay/upper/
echo "=== Merged view ==="
ls -la /tmp/overlay/merged/
# Delete a file from lower layer (creates whiteout)
rm /tmp/overlay/merged/file2.txt
# Check for whiteout file
ls -la /tmp/overlay/upper/
# Unmount overlay
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: `iotop -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:

```
# Create test partitions
sudo parted /dev/sdb mklabel gpt
sudo parted /dev/sdb mkpart primary 0% 33%
sudo parted /dev/sdb mkpart primary 33% 66%
sudo parted /dev/sdb mkpart primary 66% 100%
# Format with different filesystems
sudo mkfs.ext4 /dev/sdb1
sudo mkfs.xfs /dev/sdb2
sudo mkfs.btrfs -f /dev/sdb3
# Create mount points
sudo mkdir -p /mnt/{ext4,xfs,btrfs}
# Mount filesystems
sudo mount /dev/sdb1 /mnt/ext4
sudo mount /dev/sdb2 /mnt/xfs
sudo mount -o compress=zlib /dev/sdb3 /mnt/btrfs
# Test file creation performance
for fs in ext4 xfs btrfs; do
    echo "Testing $fs file creation..."
    time sh -c "for i in {1..1000}; do touch /mnt/$fs/file_$i; done"
done
# Test large file write performance
for fs in ext4 xfs btrfs; do
    echo "Testing $fs large file write..."
    time dd if=/dev/zero of=/mnt/$fs/bigfile bs=1M count=500 2>/dev/null
done
# Test compression ratio (BtrFS only)
echo "=== BtrFS compression analysis ==="
sudo btrfs filesystem usage /mnt/btrfs
# Clean up
sudo umount /mnt/{ext4,xfs,btrfs}
```

Container Storage with OverlayFS Demonstrating container-like storage layering:

```
# Create base image layer
mkdir -p /tmp/container/{base,app,runtime,merged,work}

# Base layer (OS files)
echo "#!/bin/bash" > /tmp/container/base/entrypoint.sh
echo "echo 'Hello from base layer'" >> /tmp/container/base/entrypoint.sh
chmod +x /tmp/container/base/entrypoint.sh

# Application layer
echo "config_file=app.conf" > /tmp/container/app/app.config
echo "#!/bin/bash" > /tmp/container/app/start-app.sh
echo "echo 'Starting application...'" >> /tmp/container/app/start-app.sh
chmod +x /tmp/container/app/start-app.sh

# Create overlay combining base and app layers
sudo mount -t overlay overlay \ -o
    lowerdir=/tmp/container/app:/tmp/container/base,upperdir=/tmp/container/runtime,workdir=/tmp/container/work
    \/tmp/container/merged

# View combined filesystem
ls -la /tmp/container/merged/

# Make runtime changes (goes to runtime layer)
echo "Runtime log entry" > /tmp/container/merged/runtime.log
echo "user_data=changed" > /tmp/container/merged/app.config

# Show layering
echo "=== Base layer ==="
ls -la /tmp/container/base/

echo "=== App layer ==="
ls -la /tmp/container/app/

echo "=== Runtime layer ==="
ls -la /tmp/container/runtime/

echo "=== Merged view ==="
ls -la /tmp/container/merged/

# Unmount
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

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

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

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

Basic TCP Socket Programming Example of TCP socket server and client:

```c
// TCP Server
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int server_fd, client_fd;
struct sockaddr_in address;
int addrlen = sizeof(address);

// Create socket
server_fd = socket(AF_INET, SOCK_STREAM, 0);

// Setup address
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(8080);

// Bind socket to address
bind(server_fd, (struct sockaddr *)&address, sizeof(address));

// Listen for connections
listen(server_fd, 3);

// Accept connection
client_fd = accept(server_fd, (struct sockaddr *)&address,
                   (socklen_t*)&addrlen);

// Read/write data
char buffer[1024] = {0};
read(client_fd, buffer, 1024);
send(client_fd, "Hello from server", 17, 0);

// TCP Client
int sock = 0;
struct sockaddr_in serv_addr;

// Create socket
sock = socket(AF_INET, SOCK_STREAM, 0);

// Setup server address
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(8080);
inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);

// Connect to server
connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

// Send/receive data
send(sock, "Hello from client", 17, 0);
read(sock, buffer, 1024);
```

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

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:

```
# Show current network configuration
ip addr show

# Configure a static IP address
sudo ip addr add 192.168.1.100/24 dev eth0
sudo ip link set eth0 up

# Add default gateway
sudo ip route add default via 192.168.1.1

# Configure DNS (temporary)
echo "nameserver 8.8.8.8" | sudo tee /etc/resolv.conf

# Test connectivity
ping -c 3 8.8.8.8
ping -c 3 google.com

# Show routing table
ip route show

# Show ARP table
ip neigh show

# Monitor network activity
ss -tuln | grep :80

# Configure persistent networking (systemd-networkd)
cat << EOF | sudo tee /etc/systemd/network/10-eth0.network
[Match]
Name=eth0

[Network]
Address=192.168.1.100/24
Gateway=192.168.1.1
DNS=8.8.8.8
DNS=8.8.4.4
EOF

# Enable systemd-networkd
sudo systemctl enable systemd-networkd
sudo systemctl restart systemd-networkd
```

Socket Programming Example

Simple client-server application:

```c
// Simple UDP Echo Server
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main() {
    int sockfd;
    struct sockaddr_in servaddr, cliaddr;
    char buffer[1024];
    socklen_t len = sizeof(cliaddr);

    // Create UDP socket
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Setup server address
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(8080);

    // Bind socket
    if (bind(sockfd, (const struct sockaddr *)&servaddr,
             sizeof(servaddr)) < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    printf("UDP Echo Server listening on port 8080...\n");

    // Echo loop
    while (1) {
        int n = recvfrom(sockfd, buffer, 1024, 0,
                         (struct sockaddr *)&cliaddr, &len);
        buffer[n] = '\0';
        printf("Received: %s\n", buffer);

        sendto(sockfd, buffer, strlen(buffer), 0,
               (const struct sockaddr *)&cliaddr, len);
    }

    return 0;
}

// Compile: gcc -o udp_server udp_server.c
// Test: echo "Hello" | nc -u localhost 8080
```

Network Namespace Isolation

Creating isolated network environments:

```bash
# Create a new network namespace
sudo ip netns add isolated

# List network namespaces
ip netns list

# Show interfaces in the new namespace (should be empty except lo)
sudo ip netns exec isolated ip link show

# Create a virtual ethernet pair
sudo ip link add veth0 type veth peer name veth1

# Move one end to the isolated namespace
sudo ip link set veth1 netns isolated

# Configure the host-side interface
sudo ip addr add 10.0.0.1/24 dev veth0
sudo ip link set veth0 up

# Configure the namespace-side interface
sudo ip netns exec isolated ip addr add 10.0.0.2/24 dev veth1
sudo ip netns exec isolated ip link set veth1 up
sudo ip netns exec isolated ip link set lo up

# Test connectivity
ping -c 3 10.0.0.2

# Run commands in the isolated namespace
sudo ip netns exec isolated ping -c 3 10.0.0.1

# Show routing in the namespace
sudo ip netns exec isolated ip route show

# Add default route in namespace (through host)
sudo ip netns exec isolated ip route add default via 10.0.0.1

# Enable IP forwarding on host for internet access
echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward

# Add NAT rule for namespace traffic
sudo iptables -t nat -A POSTROUTING -s 10.0.0.0/24 -j MASQUERADE

# Test internet connectivity from namespace
sudo ip netns exec isolated ping -c 3 8.8.8.8

# Clean up
sudo ip netns delete isolated
sudo ip link delete veth0
```

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`

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

Advanced Network Monitoring

Comprehensive network monitoring and analysis:

```
# Monitor real-time network connections
watch -n 1 'ss -tuln'

# Show detailed socket information
ss -tulpn | grep :80

# Monitor network traffic with detailed statistics
iftop -i eth0

# Capture and analyze packets
sudo tcpdump -i eth0 -w capture.pcap

# Analyze captured packets
tcpdump -r capture.pcap -n | head -20

# Monitor bandwidth usage per process
sudo nethogs eth0

# Show network interface statistics
cat /proc/net/dev

# Monitor netlink messages
ip monitor all

# Check connection tracking
sudo cat /proc/net/nf_conntrack | head -10

# Show detailed network stack statistics
ss --info --processes --numeric

# Monitor DNS queries
sudo tcpdump -i eth0 port 53

# Show kernel network parameters
sysctl -a | grep net.ipv4 | head -10

# Monitor network errors
netstat -i
```

Multiple Choice Questions

Operating System Core Concepts

Synchronisation Mechanisms

- Mutex: Binary lock (0/1) für kritische Bereiche
- Semaphore: Counting mechanism für Resource Management
- Monitor: High-level synchronization construct
- Condition Variables: Wait/Signal mechanism

Memory Management

- Pages: Logische Memory-Einheiten (Virtual Memory)
- Frames: Physische Memory-Einheiten (Physical Memory)
- MMU: Hardware für Address Translation
- TLB: Cache für Address Translation

Process States und Swapping

- Running: Prozess auf CPU
- Ready: Bereit zur Ausführung
- Blocked: Wartet auf I/O oder Resource
- Swapped: Auf Disk ausgelagert (nicht Ready/Running)

Memory Allocation

- Static Partitioning: Feste Grössen → Interne Fragmentierung
- Dynamic Partitioning: Variable Grössen → Externe Fragmentierung
- Paging: Feste Page/Frame Grösse → Nur interne Fragmentierung
- Compaction: Löst externe Fragmentierung durch Memory-Reorganisation

OS Concept Questions

System call identification

- System calls required for kernel mode operations
- I/O operations (screen output, file access) need system calls
- Time-related operations (sleep) need system calls
- Pure computation (arithmetic) doesn't need system calls
- Memory operations within process space don't need system calls

Thread and process concepts

- pthread_join() waits for thread completion
- Prevents main process from terminating before threads finish
- Software interrupts provide controlled kernel mode entry
- PCB (Process Control Block) stores OS process information

Process states

- Zombie: process finished but parent hasn't waited
- Daemon: background process without controlling terminal
- Orphan: process whose parent has terminated

Multiple Choice Questions

Pro Teilfrage können eine, mehrere oder keine Antworten zutreffen, kreuzen Sie die richtige(n) Antwort(en) an.

a) Welche der folgenden Aussagen treffen zu?
✓ Alle Mutexes können mit Semaphoren realisiert werden
× Semaphore können in allen Fällen durch Mutexes ersetzt werden
× Mutexes sollten immer anstelle von Semaphoren eingesetzt werden, weil es dann keine Deadlocks geben kann
✓ Mit Semaphoren lässt sich die Verarbeitungsreihenfolge von Prozessen und Threads erzwingen

b) Welcher der folgenden Aussagen treffen zu?
× Pages müssen grösser als Frames dimensioniert werden
× Es müssen mindestens so viele Pages wie Frames in einem System vorhanden sein
× Sowohl interne wie auch externe Fragmentierung treten bei Paging auf
✓ Pages und Frames müssen gleich gross dimensioniert werden

c) Welcher der folgenden Aussagen treffen zu?
✓ Ein MMU übersetzt Logischen Adressen zu Physikalische Adressen
× Swap in bedeutet dass ein Prozess auf die Hard-Disk verlagert wird
× Ein Prozess der auf der Harddisk verlagert worden ist kann sich im Zustand Running befinden

d) Welcher der folgenden Aussagen treffen zu?
× Bei Static Partitioning tritt External Fragmentation auf
✓ Bei Dynamic Partitioning tritt External Fragmentation auf
× Nur bei Best Fit Allocation wird kein Compaction benötigt

Erklärungen:

a) Semaphore vs. Mutexes:
✓ Mutexes sind Binary Semaphores (0/1) - können mit Semaphoren realisiert werden
× Counting Semaphores (>1) können nicht durch Mutexes ersetzt werden
× Auch mit Mutexes sind Deadlocks möglich
✓ Semaphore eignen sich gut für Prozess-Synchronisation

b) Paging:
× Pages und Frames sind gleich gross
× Anzahl ist unabhängig - Virtual Memory ermöglicht mehr Pages als Frames
× Nur interne Fragmentierung (innerhalb Pages)
✓ Pages (logisch) = Frames (physisch) in der Grösse

c) Memory Management:
✓ MMU (Memory Management Unit) macht Address Translation
× Swap in = von Disk in Memory (nicht umgekehrt)
× Ausgelagerte Prozesse sind nicht Running

d) Partitioning:
× Static: Interne Fragmentierung (feste Grössen)
✓ Dynamic: Externe Fragmentierung (variable Grössen)
× Alle Dynamic Allocation Methoden benötigen eventüll Compaction

Operating System Multiple Choice

Typical OS concept questions:

Which activities require system calls?
- √ Display variable value on screen (I/O operation)
- **X** Compare two variables (CPU operation)
- √ Sleep for 1 second (timer operation)
- **X** Increment integer variable (memory operation)
- **X** Increment float variable (CPU operation)

What is pthread_join() used for?
- **X** Terminates all user-level threads
- **X** Can be replaced by sleep(0)
- √ Prevents process termination before threads finish
- **X** Blocks CPU until all threads terminate

Why use software interrupts for system calls?
- **X** They are faster than procedure calls
- √ Procedure calls cannot switch to system mode
- √ Applications don't know system call routine addresses