1. USER INTERFACES

1.1 Command-Line Interface (CLI)

What is it?

Text-based interface where you type commands to interact with the OS

Key Features:

- Direct command execution
- Powerful for automation
- Fast for repetitive tasks
- Steep learning curve but efficient once mastered

Examples:

- **Unix/Linux:** bash, zsh, csh
- **Windows:** Command Prompt, PowerShell
- **macOS:** Terminal

When to use:

- Automating tasks with scripts
- Remote server administration
- Quick file operations
- System configuration

1.2 Graphical User Interface (GUI)

What is it?

Visual interface using windows, icons, menus, and mouse (WIMP)

Key Features:

- Point-and-click interaction
- Visual metaphors (desktop, folders)
- Easy for beginners
- Intuitive discovery of features

Examples:

- Windows Explorer
- macOS Finder
- GNOME/KDE on Linux

When to use:

- General everyday computing
- Visual tasks (graphics, video)
- Learning new software
- Multitasking with multiple windows

---

2. SHELL COMMANDS AND SCRIPTS

2.1 The Shell

Simple Definition:

The shell is your command interpreter - it's the program that reads your commands and makes the computer execute them.

What it does:

- Interprets commands you type
- Launches programs
- Manages processes
- Provides scripting capabilities

Popular Shells:

- **bash** - Most common on Linux
- **zsh** - Default on modern macOS
- **PowerShell** - Windows scripting

2.2 Shell Scripts

What are they?

Text files containing a list of commands that run automatically

Script File Extensions:

- Linux: `.sh`

- Windows: `.bat` or `.cmd`

The Shebang Line:

```
bash#!/bin/bash
```

- First line of a script
- Tells system which interpreter to use
- Makes script executable

Example Script:

```
bash#!/bin/bash for i in *.c do echo $i done
```

This script lists all `.c` files in the current directory.

Why use scripts?

- Automate repetitive tasks
- Save time
- Reduce errors
- Create reusable workflows

---

3. OPERATING SYSTEM SERVICES

3.1 Services for Users and Programs

1. Program Execution

What it does:

- Loads programs into memory
- Runs your programs
- Ends programs (normally or when they crash)

Example: When you double-click an app, the OS loads and runs it.

2. I/O Operations

What it does:

- Controls access to input/output devices
- Provides safe interface to hardware

Why needed:

- You can't directly access hardware (for security)
- OS manages device sharing between programs

Examples:

- Reading from keyboard
- Writing to screen
- Saving files to disk

3. File System Manipulation

What it does:

- Creates/deletes files and folders
- Reads and writes file contents
- Manages file permissions
- Searches for files

Common operations:

- Create a document
- Delete old files
- Copy files between folders
- Set who can access files

4. Communications

Two ways processes can communicate:

Message Passing:

- Processes send explicit messages to each other
- Like passing notes
- Good for processes on different computers
- Examples: pipes, sockets

Shared Memory:

- Processes share a memory area
- Like sharing a whiteboard

- Faster but needs coordination
- Good for processes on same computer

5. Error Detection

What it does:

- Constantly checks for errors
- Takes corrective action
- Logs problems

Error types:

- **Hardware errors:** CPU problems, memory failures
- **I/O errors:** Disk read failures, network problems
- **Program errors:** Division by zero, illegal memory access

3.2 Services for System Efficiency

6. Resource Allocation

What it manages:

- CPU time (which program runs when)
- Memory (how much RAM each program gets)
- Disk space
- I/O devices

Goal: Fair sharing among all users and programs

7. Accounting

What it tracks:

- Which users use the system
- How much resources they consume
- What types of resources are used

Used for:

- Billing (in commercial systems)
- Statistics and performance analysis
- Capacity planning

8. Protection and Security

Protection:

- Controls who can access what
- Prevents unauthorized resource use
- Isolates programs from each other

Security:

- User authentication (passwords, biometrics)
- Defends against hackers
- Detects suspicious activity
- Logs security events

---

4. SYSTEM CALLS

4.1 What are System Calls?

Simple explanation:

System calls are the way programs ask the operating system to do something for them.

Think of it like:

- You (the program) can't directly cook in a restaurant kitchen (access hardware)
- You must ask the waiter (system call) to place your order
- The chef (OS kernel) prepares your food (performs the operation)

4.2 APIs (Application Programming Interfaces)

What are APIs?

Higher-level interfaces that make system calls easier to use

Common APIs:

- **POSIX** - Used by Unix/Linux/macOS
- **Win32** - Used by Windows
- **Java API** - Used by Java programs

Why use APIs instead of direct system calls?

- **Portability:** Same code works on different systems
- **Easier:** Don't need to know low-level details
- **Stable:** Changes to OS don't break your program

4.3 How System Calls Work

The Process:

1. Your program calls an API function
2. API function invokes the system call
3. System call goes to OS kernel
4. Kernel performs the operation
5. Result comes back to your program

Important: You don't need to know HOW it works, just WHAT it does

4.4 Passing Parameters to System Calls

Three methods:

1. Registers (simplest):

- Store values in CPU registers
- Fast but limited space

2. Memory block:

- Store many parameters in memory
- Pass memory address in register
- Used by Linux

3. Stack:

- Push parameters onto stack
- OS pops them off
- No limit on number of parameters

5. TYPES OF SYSTEM CALLS

5.1 Process Control

What they do: Manage programs

Common calls:

- `create process` - Start new program
- `terminate process` - End program
- `wait` - Wait for process to finish
- `allocate memory` - Get more RAM
- `free memory` - Release RAM

Examples:

- Unix: `fork()`, `exec()`, `wait()`, `exit()`
- Windows: `CreateProcess()`, `ExitProcess()`

5.2 File Management

What they do: Work with files

Common calls:

- `create` / `delete` - Make/remove files
- `open` / `close` - Start/stop using file
- `read` / `write` - Get/put data
- `seek` - Move to position in file

Examples:

- Unix: `open()`, `read()`, `write()`, `close()`
- Windows: `CreateFile()`, `ReadFile()`, `WriteFile()`

5.3 Device Management

What they do: Control hardware devices

Common calls:

- `request device` - Get exclusive access
- `release device` - Free device
- `read` / `write` - Transfer data
- `get/set attributes` - Configure device

Philosophy: Treat devices like files (Unix approach)

5.4 Information Maintenance

What they do: Get/set system information

Common calls:

- `get time/date` - Check system clock
- `get system data` - CPU, memory info
- `get/set process attributes` - Process details

Examples:

- `time()`, `getpid()`, `stat()`

5.5 Communications

What they do: Enable inter-process communication

Common calls:

- `create connection` - Set up communication
- `send` / `receive` - Exchange messages
- `attach/detach devices` - Network connections

Examples:

- `socket()`, `connect()`, `send()`, `recv()`

5.6 Protection

What they do: Manage security

Common calls:

- `get/set file permissions` - Control access
- Authentication functions
- Access control

Examples:

- Unix: `chmod()`, `chown()`
- Windows: `SetFileSecurity()`

---

6. SYSTEM PROGRAMS

6.1 What are System Programs?

Simple definition:

Utility programs that make the OS easier to use

Think of them as: The tools that come with your OS

6.2 Categories of System Programs

1. File Management

What they do: Manipulate files and directories

Examples:

- **Linux:** `cp` (copy), `mv` (move), `rm` (delete), `ls` (list)
- **Windows:** `copy`, `move`, `del`, `dir`

2. Status Information

What they do: Show system status

Information shown:

- Date and time
- Available memory
- Disk space
- Running processes

Examples:

- **Linux:** `date`, `df`, `free`, `top`, `ps`
- **Windows:** `date`, `time`, `mem`, `tasklist`

3. File Modification

What they do: Edit and search files

Examples:

- **Editors:** `vi`, `emacs`, `nano`, Notepad
- **Search:** `grep`, `find`
- **Transform:** `sed`, `awk`, `sort`

4. Programming Language Support

What they include:

- **Compilers:** Convert code to executable (GCC, Java)
- **Interpreters:** Run code directly (Python, Ruby)
- **Debuggers:** Find bugs (`gdb`)

5. Program Loading

What they do: Get programs ready to run

Components:

- **Loaders:** Load program into memory
- **Linkers:** Combine code modules
- **Debuggers:** Help find errors

6. Communications

What they enable:

- Sending messages between users
- Web browsing
- Email
- Remote login (`ssh`)
- File transfer (`ftp`, `scp`)

7. Background Services

What they are: Programs that run automatically

Also called: Services, daemons

Examples:

- Web server
- Print server
- Network services
- System monitoring

Note: Run in user space, not kernel

---

7. OPERATING SYSTEM STRUCTURES

7.1 Why Structure Matters

The Problem:

- Operating systems are HUGE (millions of lines of code)
- Need organization to manage complexity
- Must be maintainable and understandable

Goals:

- Easy to understand
- Easy to maintain
- Easy to modify
- Reliable
- Efficient

7.2 Simple Structure

Example: MS-DOS

Characteristics:

- Minimal structure
- Everything mixed together
- No clear layers

Problems:

- Applications can access hardware directly
- Security vulnerabilities
- One bad program can crash entire system

- Hard to maintain

Why this design?

- Limited by early PC hardware
- No memory protection available
- Focus was on fitting in small memory

7.3 Layered Approach

Concept: Build OS in layers, like a cake

Structure:

```
textLayer N: User Interface Layer N-1: User Programs ... Layer 2: I/O Management Layer 1: Device Drivers Layer 0: Hardware
```

Rules:

- Each layer uses only lower layers
- Each layer provides services to upper layers
- Clear interfaces between layers

Advantages:

- Easy to build and debug
- Build from bottom up
- Test each layer independently
- Changes to one layer don't affect others

Example: Traditional UNIX

Two main parts:

1. **System Programs** (user level)
2. **Kernel** (everything else)

Problem:

- Kernel has too much functionality in one place
- "Monolithic" kernel
- Hard to debug and maintain

7.4 Microkernel Approach

Philosophy: Keep kernel as small as possible

What stays in kernel:

- Basic process management
- Memory management
- Inter-process communication (IPC)

What moves to user space:

- File systems
- Device drivers
- Network protocols

Communication: Everything talks via message passing

Advantages:

✅ More reliable (less code in kernel)

✅ More secure (smaller attack surface)

✅ Easier to port to new hardware

✅ Can update services without rebooting

Disadvantages:

❌ Performance overhead (more message passing)

Examples:

- Mach (used in macOS)
- QNX
- MINIX

7.5 Modular Approach

Concept: Loadable kernel modules

How it works:

- Core kernel provides essential services
- Additional modules can be loaded/unloaded dynamically
- No reboot needed to add functionality

Advantages:

- Very flexible
- Better performance than microkernel
- Easy to add new features
- Update individual modules

Examples:

- Linux (loadable kernel modules)
- Solaris

Commands:

```
bashlsmod # List loaded modules insmod module # Load a module rmmod module # Remove a module
```

7.6 Hybrid Systems

Reality: Most modern OSes mix approaches

Why hybrid?

- Pure models have trade-offs
- Combine best features
- Meet practical needs

Example: Mac OS X

Structure:

- Mach microkernel (core)
- BSD Unix layer (services)
- Cocoa environment (user interface)

Benefits:

- Microkernel reliability
- Unix compatibility
- Good performance

Example: Android

Structure:

- Linux kernel (bottom)
- Libraries and runtime (middle)
- Application framework (top)

- Apps (user level)

Benefits:

- Linux stability
- Java app development
- Open source

---

8. KEY CONCEPTS TO REMEMBER

System Calls vs APIs vs System Programs

System Calls:

- Direct requests to OS kernel
- Low-level
- Examples: `read()`, `write()`, `fork()`

APIs:

- Higher-level wrapper around system calls
- Easier to use
- Examples: POSIX, Win32

System Programs:

- Utility programs
- Use APIs/system calls
- Examples: `ls`, `cp`, text editors

Communication Models

Message Passing:

- Explicit messages
- Good for distributed systems
- Examples: pipes, sockets

Shared Memory:

- Direct memory access
- Faster for local processes

- Needs synchronization