Operating Systems Notes - Chapter 2

June 9, 2025

Contents

2.1 Operating-system services

Introduction

An operating system (OS) provides the environment for program execution. OS designs vary, but defining system goals is crucial before design begins. We can view an OS by its services, user/programmer interface, or internal components. This chapter explores these aspects, covering OS services, their provision, debugging, design methodologies, creation, and booting process.

Chapter objectives

- Identify services provided by an operating system.
- Illustrate how system calls are used to provide operating system services.
- Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing operating systems.
- Illustrate the process for booting an operating system.
- Apply tools for monitoring operating system performance.
- Design and implement kernel modules for interacting with a Linux kernel.

Operating-system services

An OS provides an environment for program execution, offering services to programs and users. While specific services vary, common classes exist. These services also simplify programming tasks.

OS services helpful to the user:

- **User interface (UI):** Provides interaction via GUI, touch-screen, or CLI.
- **Program execution:** Loads, runs, and terminates programs.
- **I/O operations:** Manages program access to files and I/O devices for efficiency and protection.
- **File-system manipulation:** Handles file/directory operations (read, write, create, delete, search, list) and permissions.
- **Communications:** Enables inter-process communication (IPC) via shared memory or message passing, locally or across networks.
- **Error detection:** Constantly detects and corrects errors in hardware, I/O devices, and user programs, taking appropriate action (e.g., halting, terminating process, returning error code).

OS functions ensuring efficient system operation by sharing computer resources among multiple processes:

- **Resource allocation:** Manages allocation of CPU cycles, memory, file storage, and I/O devices to multiple running processes, using scheduling routines.
- **Logging:** Tracks resource usage for accounting or accumulating statistics to improve computing services.
- **Protection and security:** Controls access to system resources and defends against external/internal attacks (e.g., viruses, DoS). Requires user authentication and safeguards for I/O devices.

Section glossary

Term	Definition
user interface (UI)	A method by which a user interacts with a computer.
graphical user interface (GUI)	A computer interface comprising a window system with a pointing device to direct I/O, choose from menus, and make selections and, usually, a keyboard to enter text.
touch-screen interface	A user interface in which touching a screen allows the user to interact with the computer.
command-line interface (CLI)	A method of giving commands to a computer based on a text input device (such as a keyboard).
shared memory	In interprocess communication, a section of memory shared by multiple processes and used for message passing.
message passing	In interprocess communication, a method of sharing data in which messages are sent and received by processes. Packets of information in predefined formats are moved between processes or between computers.

2.2 User and operating-system interface

Users interface with the OS via three fundamental approaches: command-line interface (CLI) or **command interpreter**, and two forms of graphical user interface (GUI).

Command interpreters

- Most OS (Linux, UNIX, Windows) treat the command interpreter as a special program run at process initiation or user login.
- Systems with multiple command interpreters are known as **shells** (e.g., C shell, Bourne-Again shell, Korn shell on UNIX/Linux).
- Shells provide similar functionality; choice is personal preference.
- Main function: get and execute user commands, often manipulating files (create, delete, list, print, copy, execute).
- Commands can be implemented:
  - **Interpreter contains code:** Interpreter holds execution code, increasing its size.
  - **System programs:** Most commands are separate system programs (e.g., UNIX 'rm file.txt' executes the 'rm' program). Allows easy addition of new commands without changing the interpreter.

Graphical user interface

- A second interface is the **graphical user interface (GUI)**.
- Users interact with a mouse-based window-and-menu system using a **desktop** metaphor.
- **Icons** represent programs, files, and functions; clicking them or selecting from menus invokes actions.
- GUIs originated from Xerox PARC (1973, Xerox Alto) and became widespread with Apple Macintosh (1980s).
- macOS adopted the Aqua interface.
- Microsoft Windows (Version 1.0) added a GUI to MS-DOS, with later versions enhancing functionality.
- Traditionally CLI-dominated UNIX systems now offer GUIs like KDE and GNOME, available under open-source licenses.

Touch-screen interface

- Smartphones and tablets typically use a touch-screen interface, as CLI/mouse systems are impractical.
- Users interact via **gestures** (e.g., pressing, swiping).
- Most mobile devices simulate keyboards on the touch screen.
- Apple iPhone/iPad use the **Springboard** touch-screen interface.

Choice of interface

- Choice between CLI and GUI is often personal preference.
- **System administrators** and **power users** prefer CLI for efficiency and faster access, especially for repetitive tasks via **shell scripts**.
- Some systems offer only a subset of functions via GUI, reserving less common tasks for CLI.
- Most Windows users prefer the GUI; recent Windows versions offer both standard GUI and touch-screen interfaces.
- macOS (based on a UNIX kernel) now provides both the Aqua GUI and a command-line interface.
- Mobile systems (iOS, Android) predominantly use touch-screen interfaces.
- The user interface is typically separate from the core OS structure; this book focuses on providing adequate service to user programs, not UI design.

Section glossary

Term	Definition
command interpreter	OS component interpreting user commands.
shell	A command interpreter on systems with multiple choices.
desktop	GUI workspace on screen.
icons	Images representing objects in GUI.
folder	File system component for grouping files.
gestures	Motions causing computer actions (e.g., "pinching").
Springboard	The iOS touch-screen interface.
system administrators	Users who configure, monitor, and manage systems.
power users	Users with deep system knowledge.
shell script	File containing a series of commands for a specific shell.

2.3 System calls

Example

- System calls provide an interface to OS services, typically as C/C++ functions, sometimes assembly.
- Even simple programs (e.g., file copy) make extensive use of system calls.
- A file copy program (`cp in.txt out.txt`) involves:
  - Obtaining file names (command line, user input, GUI).
  - Opening input file, creating/opening output file (with error handling for non-existence, protection, or existing files).
  - Looping to read from input and write to output (with error handling for EOF, hardware failure, disk space).
  - Closing files, displaying messages, and terminating the program.
- System calls are fundamental for program interaction with the OS.

Application programming interface

- Application developers typically use an **application programming interface** (*API*) rather than direct system calls.
- An API specifies functions, parameters, and return values available to programmers (e.g., Windows API, POSIX API, Java API).
- APIs are accessed via libraries (e.g., **libc** for C programs on UNIX/Linux).
- API functions invoke actual system calls in the kernel (e.g., `CreateProcess()` calls `NTCreateProcess()`).
- Benefits of using APIs:
  - **Portability:** Programs can run on any system supporting the same API.
  - **Simplicity:** APIs are often less detailed and easier to use than raw system calls.
- The **run-time environment** (*RTE*) provides a **system-call interface** that links API calls to OS system calls.
- The system-call interface uses a table indexed by system call numbers to invoke the correct kernel function.
- Details of system call implementation are hidden from the programmer by the API and managed by the RTE.
- Parameters to system calls can be passed via registers, memory blocks, or a **stack** (using **push** and **pop** operations).

Types of system calls

System calls are broadly categorized into six types:

- **Process control:**
  - Terminate program normally (`end()`) or abnormally (`abort()`).
  - Debugging tools (**debugger**) can examine memory dumps from abnormal terminations.
  - Load and execute other programs (`load()`, `execute()`).
  - Create new processes (`create_process()`, `fork()`).
  - Control process attributes (priority, execution time) (`get_process_attributes()`, `set_process_attributes()`).
  - Terminate created processes (`terminate_process()`).
  - Wait for processes to finish (`wait_time()`, `wait_event()`) or signal events (`signal_event()`).
  - **Lock** shared data for integrity (`acquire_lock()`, `release_lock()`).
  - Examples: Arduino (single-tasking, **sketch**, **boot loader**); FreeBSD (multitasking, `fork()`, `exec()`, `exit()`).
- **File management:**
  - Basic operations: `create()`, `delete()`, `open()`, `read()`, `write()`, `reposition()`, `close()` files.
  - Similar operations apply to directories.
  - Get/set file/directory attributes (`get_file_attributes()`, `set_file_attributes()`).
- **Device management:**
  - Request (`request()`) and release (`release()`) devices for exclusive use.
  - Read, write, and reposition devices, similar to files.
  - Many OS (e.g., UNIX) merge files and devices into a combined structure.
- **Information maintenance:**
  - Transfer information between user program and OS.
  - Examples: current `time()` and `date()`, OS version, free memory/disk space.
  - Debugging aids: `dump()` memory, `strace` (Linux), **single step** CPU mode.
  - Get/set process information (`get_process_attributes()`, `set_process_attributes()`).
- **Communications:**

  - **Message-passing model:** Processes exchange messages directly or via mailboxes.
    - Requires opening connections, knowing **host name** and **process name**.
    - **Daemons** (**server**) wait for connections from **clients**.
    - Use `read_message()` and `write_message()` for exchange.
  - **Shared-memory model:** Processes access shared memory regions.
    - Processes agree to remove memory access restrictions.
    - Exchange information by reading/writing shared data.
    - Faster communication, but requires protection and synchronization.
  - Most systems implement both models.
- **Protection:**
  - Control access to computer system resources.
  - System calls: `set_permission()`, `get_permission()`, `allow_user()`, `deny_user()`.
  - Essential for multiprogrammed, networked, and mobile systems.

Term	Definition
system call	Software-triggered interrupt allowing a process to request a kernel service.
application programming interface (API)	A set of commands, functions, and other tools that can be used by a programmer in developing a program.
C library (libc)	The standard UNIX/Linux system API for programs written in the C programming language.
run-time environment (RTE)	The full suite of software needed to execute applications written in a given programming language, including its compilers, libraries, and loaders.
system-call interface	An interface that serves as the link to system calls made available by the operating system and that is called by processes to invoke system calls.
push	The action of placing a value on a stack data structure.
stack	A sequentially ordered data structure that uses the last-in, first-out (LIFO) principle for adding and removing items; the last item placed onto a stack is the first item removed.
process control	A category of system calls.
information maintenance	A category of system calls.
communications	A category of system calls.
protection	A category of system calls. Any mechanism for controlling the access of processes or users to the resources defined by a computer system.
debugger	A system program designed to aid programmers in finding and correcting errors.
bug	An error in computer software or hardware.
lock	A mechanism that restricts access by processes or subroutines to ensure integrity of shared data.
sketch	An Arduino program.
single step	A CPU mode in which a trap is executed by the CPU after every instruction (to allow examination of the system state after every instruction); useful in debugging.
message-passing model	A method of interprocess communication in which messages are exchanged.
host name	A human-readable name for a computer.
process name	A human-readable name for a process.
daemon	A service that is provided outside of the kernel by system programs that are loaded into memory at boot time and run continuously.
client	A computer that uses services from other computers (such as a web client). The source of a communication.
server	In general, any computer, no matter the size, that provides resources to other computers.
shared-memory model	An interprocess communication method in which multiple processes share memory and use that memory for message passing.

2.4 System services

System services (or **system utilities**) provide a convenient environment for program development and execution, categorized as:

- **File management:** Create, delete, copy, rename, print, list, manipulate files/directories.
- **Status information:** Query system for date, time, memory/disk, user count; performance, logging, debugging. **Registry** for configuration.
- **File modification:** Text editors for content; search/transform commands.
- **Programming-language support:** Compilers, assemblers, debuggers, interpreters (C, C++, Java, Python) often OS-provided.
- **Program loading and execution:** Loaders (absolute, relocatable, linkage editors, overlay); debugging for high-level/machine languages.
- **Communications:** Virtual connections among processes, users, systems. Messages, web browsing, e-mail, remote login, file transfer.
- **Background services:** System programs launched at boot. Constantly running: **services**, **subsystems**, **daemons** (e.g., network, schedulers, error monitoring, print servers).

Application programs (e.g., web browsers, word processors) supplied. User's OS view shaped by these programs, not system calls; enables diverse interfaces (GUI, CLI) or dual-booting.

Section glossary

Term	Definition
system service	A collection of applications included with or added to an operating system to provide services beyond those provided by the kernel.
system utility	A collection of applications included with or added to an operating system to provide services beyond what are provided by the kernel.
registry	A file, set of files, or service used to store and retrieve configuration information. In Windows, the manager of hives of data.
service	A software entity running on one or more machines and providing a particular type of function to calling clients. In Android, an application component with no user interface; it runs in the background while executing long-running operations or performing work for remote processes.
subsystem	A subset of an operating system responsible for a specific function (e.g., memory management).
application program	A Program designed for end-user execution, such as a word processor, spreadsheet, compiler, or Web browser.

2.5 Linkers and loaders

A program typically resides on disk as a binary executable file (e.g., `a.out`, `prog.exe`). To execute, it must be loaded into memory within a process's context. This section details the compilation-to-execution procedure, involving compilers, linkers, and loaders.

2.5.1: The role of the linker and loader.

The process of preparing a program for execution involves several key steps:

- A source program (e.g., `main.c`) is processed by a **compiler** (e.g., `gcc`), producing **object code**.
- A **linker** combines these object files, along with any necessary libraries (e.g., standard C or math libraries), into a single **executable file**.
- At runtime, a **loader** brings the executable file into memory. This includes loading **dynamically linked libraries (DLLs)** as needed, and performing **relocation** to adjust addresses for proper execution.

Source files are compiled into **relocatable object files**, designed for loading into any physical memory location. The **linker** then combines these into a single binary **executable file**, potentially incorporating other object files or libraries.

Key concepts related to program loading and execution:

- **Loader:** Responsible for loading the executable file into memory.
- **Relocation:** A crucial activity during linking and loading that assigns final addresses to program parts and adjusts code/data to match.
- **Dynamic Linking:** Modern systems often use **dynamically linked libraries (DLLs)**, where libraries are linked and loaded only when required at runtime. This avoids loading unused libraries and allows multiple processes to share common libraries, saving memory.

On UNIX systems, invoking a program (e.g., `./main`) triggers the shell to create a new process via `fork()` and then use `exec()` to invoke the loader.

Standard formats for object and executable files:

- **ELF (Executable and Linkable Format):** The standard for UNIX/Linux systems, including compiled machine code and a symbol table. It contains the program's **entry point**.
- **Portable Executable (PE):** Used by Windows systems.
- **Mach-O:** Used by macOS.

ELF format

Linux provides various commands to identify and evaluate ELF files. For example, the `file` command determines a file type. If `main.o` is an object file, and `main` is an executable file, the command

```
file main.o
```

will report that `main.o` is an ELF relocatable file, while the command

```
file main
```

will report that `main` is an ELF executable. ELF files are divided into a number of sections and can be evaluated using the `readelf` command.

Section glossary

Term	Definition
relocatable object file	The output of a compiler in which the contents can be loaded into any location in physical memory.
linker	A system service that combines relocatable object files into a single binary executable file.
executable file	A file containing a program that is ready to be loaded into memory and executed.
loader	A system service that loads a binary executable file into memory, where it is eligible to run on a CPU core.
relocation	An activity associated with linking and loading that assigns final addresses to program parts and adjusts code and data in the program to match those addresses.
dynamically linked libraries (DLLs)	System libraries that are linked to user programs when the processes are run, with linking postponed until execution time.
executable and linkable format (ELF)	The UNIX standard format for relocatable and executable files.
portable executable (PE)	The Windows format for executable files.
Mach-O	The macOS format of executable files.

2.6 Why applications are operating-system specific

Applications compiled on one operating system are generally not executable on others due to unique system calls and other barriers. However, some applications can run on multiple operating systems through specific techniques.

2.6.1: Section review questions.

An application can be made available to run on multiple operating systems in one of three ways:

1. **Interpreted languages:** Applications written in languages like Python or Ruby use an interpreter available for multiple operating systems. The interpreter executes equivalent native instructions and calls native OS functions. Performance may suffer, and features might be limited.
2. **Virtual machines:** Languages like Java include a virtual machine as part of their runtime environment (RTE). The RTE, ported to various operating systems, loads the application into the virtual machine. This approach has similar disadvantages to interpreters.
3. **Standard language/API with porting:** Application developers use a standard language or API where the compiler generates machine- and OS-specific binaries. The application must be ported to each OS, which is time-consuming. POSIX API is an example for UNIX-like systems.

Despite these approaches, cross-platform application development remains challenging due to several architectural differences:

- **Binary formats:** Each OS has a unique binary format (header, instructions, variables layout) for applications, dictating how the OS loads and executes the file.
- **CPU instruction sets:** CPUs have varying instruction sets, requiring applications to contain appropriate instructions for correct execution.
- **System calls:** OS system calls vary significantly in operands, ordering, invocation methods, numbering, meanings, and result returns.

Approaches like Linux's adoption of the **ELF format** for binary executable files provide a common standard across Linux and UNIX systems, but do not guarantee cross-hardware platform compatibility.

An **application binary interface (ABI)** defines how different components of binary code interface for a given operating system on a specific architecture. ABIs specify low-level details such as address width, parameter passing methods for system calls, runtime stack organization, binary format of system libraries, and data type sizes. While an ABI ensures compatibility on systems supporting that ABI, it does not provide cross-platform compatibility as it is defined for a specific OS and architecture.

In summary, these differences necessitate that applications are written for and compiled on a specific operating system and CPU type (e.g., Intel x86 or ARM v8) unless an interpreter or RTE is used. This explains the extensive work required for programs like Firefox to run across various platforms (Windows, macOS, Linux, iOS, Android) and CPU architectures.

Section glossary

Term	Definition
port	To move code from its current platform to another platform (e.g., between operating systems or hardware systems).
application binary interface (ABI)	Defines how different components of binary code can interface for a given operating system on a given architecture.

2.7 Operating-system debugging

Debugging is a crucial task in operating system development, involving the detection and correction of errors. Errors can range from hardware failures to application bugs.

2.7.1: Section review questions.

Debugging is the process of finding and fixing errors, or **bugs**. Operating systems, being large and complex, are prone to bugs. Debugging can be challenging due to the difficulty of reproducing errors and the system's concurrent nature.

Common debugging tools and techniques:

- **Log files:** System-generated files that record events, errors, and warnings. Analyzing these logs helps identify the cause of issues.
- **Core dump:** A snapshot of the memory of a process at the time of a crash. It contains the state of the program, including register values, stack, and memory, useful for post-mortem analysis.
- **Crash dump (or system dump):** Similar to a core dump but for the entire operating system. It captures the system's state when a kernel panic or system crash occurs.
- **Debugger:** A software tool that allows developers to step through code, inspect variables, and set breakpoints to understand program execution flow and identify bugs.
- **Tracing:** A technique to record the sequence of events or function calls during program execution, providing insights into behavior.
- **Profiling:** Analyzing program performance to identify bottlenecks and areas for optimization.

Performance tuning is a related activity focused on improving system efficiency. Tools for performance tuning often include:

- **Profilers:** Identify which parts of a program consume the most resources (CPU, memory).
- **System monitors:** Track real-time system resource usage (CPU utilization, memory usage, disk I/O, network activity).
- **Benchmarking tools:** Measure system performance under specific workloads to compare against baselines or other systems.

Section glossary

Term	Definition
debugging	The process of finding and fixing errors.
bug	An error in a program.
core dump	A file containing the state of a program when it crashed.
crash dump	A file containing the state of the operating system when it crashed.
system dump	A file containing the state of the operating system when it crashed.
performance tuning	An activity that improves the performance of a system.

2.8 Operating-system structure

Operating systems are complex and structured into components or modules for proper function and easy modification. This section discusses common kernel structures.

Monolithic structure

All kernel functionality in a single, static binary file in one address space.

- **Example:** Original UNIX OS.
- **Linux:** Monolithic kernel, but with modular design for runtime modification.
- **Advantages:** High performance (minimal overhead, fast internal communication).
- **Disadvantages:** Difficult to implement and extend.

Layered approach

Kernel divided into separate, smaller components (**loosely coupled**) vs. **tightly coupled** monolithic.

- **Concept:** OS broken into layers (Layer 0: hardware, Layer N: user interface). Each layer uses functions only from lower layers.
- **Advantages:** Simplicity in construction and debugging.
- **Disadvantages:** Challenges in defining layers, potential poor performance (multiple layer traversals).
- **Current use:** Pure layered systems rare; some layering with fewer, more functional layers is common.

Microkernels

Removes nonessential components from kernel, implementing them as user-level programs in separate address spaces (smaller kernel).

- **Core functions:** Minimal process/memory management, communication facility.
- **Communication:** Indirect via message passing through microkernel.
- **Benefits:** Easier OS extension/modification, easier portability, enhanced security/reliability.
- **Examples: Mach** (kernel of **Darwin**), QNX.
- **Disadvantages:** Performance can suffer (message copying, process switching overhead).

Modules

Loadable kernel modules (LKMs) allow dynamic linking of additional services.

- **Design:** Kernel provides core services; others are dynamic. Avoids kernel recompilation for changes.
- **Comparison:** More flexible than layered; more efficient than microkernels.
- **Linux:** Uses LKMs for device drivers, file systems (dynamic insertion/removal).

Hybrid systems

Combine different structures to balance performance, security, usability.

- **Linux:** Monolithic + Modular.
- **Windows:** Largely monolithic + Microkernel-like behavior (user-mode subsystems/OS **personalities**) + Dynamically loadable kernel modules.

macOS and iOS

Apple's OSes share a layered structure:

- **User experience layer:** User interaction (**Aqua**/macOS, **Springboard**/iOS).
- **Application frameworks layer: Cocoa** (macOS), **Cocoa Touch** (iOS) for Objective-C/Swift APIs.
- **Core frameworks:** Graphics/media (Quicktime, OpenGL).
- **Kernel environment: Darwin** (Mach microkernel + BSD UNIX kernel).

Darwin hybrid structure:

- **System-call interfaces:** Two interfaces: Mach (**traps**), BSD (POSIX).
- **Mach services:** Memory management, CPU scheduling, IPC (**kernel abstractions**: tasks, threads, memory objects, ports).
- **Kernel environment additions:** I/O kit, dynamic modules (**kernel extensions**/**kexts**).
- **Performance mitigation:** Combines components in single address space to reduce message copying.

Android

Open-source mobile OS (Google-led). Layered software stack.

- **Application development:** Java with Android API. Compiled to '.dex' for Android RunTime (ART) VM.
- **ART compilation: Ahead-of-time (AOT) compilation** to native code on installation.
- **JNI:** Java Native Interface for direct hardware access (less portable).
- **Native libraries:** webkit, SQLite, SSLs.
- **Hardware abstraction layer (HAL):** Abstracts hardware for portability.
- **C library: Bionic** (smaller, efficient on mobile CPUs).
- **Linux kernel:** Modified for mobile needs (power/memory management, **Binder** IPC).

Windows subsystem for linux

Windows hybrid architecture with subsystems. **WSL** allows native Linux apps (ELF binaries) on Windows 10.

- **Operation:** 'bash.exe' starts **Linux instance** (init, bash shell) in Windows **Pico** process.
- **Pico processes:** Load Linux binaries.
- **System call translation:** LXCore/LXSS translate Linux calls (provide functionality or combine with Windows calls).

Section glossary

Term	Definition
monolithic	Kernel without structure.
tightly coupled systems	Processors in close communication, sharing resources.
loosely coupled	Kernel composed of components with specific, limited functions.
layered approach	OS separated into layers (hardware to user interface).
Mach	Microkernel OS with threading (Carnegie Mellon).
microkernel	Removes nonessential components from kernel, implements as user-level programs.
loadable kernel module (LKM)	Kernel links additional services dynamically.
user experience layer	Software interface for user interaction (macOS/iOS).
application frameworks layer	Includes Cocoa/Cocoa Touch (macOS/iOS) for APIs.
core frameworks	Graphics/media support (macOS/iOS).
kernel environment	Darwin layer (Mach microkernel + BSD UNIX kernel).
Darwin	Apple's open-source kernel.
trap	Software interrupt (error or OS service request).
kernel abstractions	Components adding functionality to Mach (tasks, threads, memory objects, ports).
kernel extensions (kexts)	Third-party components added to kernel (macOS).
kexts	Third-party components added to kernel (macOS).
ahead-of-time (AOT) compilation	ART compiles Java apps to native code on installation.
Bionic	Android standard C library (smaller, efficient on mobile CPUs).
Windows Subsystem for Linux (WSL)	Windows 10 component for running native Linux apps.
Linux instance	Set of Pico processes in Windows created by WSL.
Pico	WSL process translating Linux system calls.

2.9 Building and booting an operating system

Operating systems are commonly designed to run on a class of machines with various peripheral configurations, rather than a single specific machine.

Operating-system generation

When generating (or building) an operating system from scratch, the following steps are typically involved:

1. Write or obtain the OS source code.
2. Configure the OS for the target system.
3. Compile the OS.
4. Install the OS.
5. Boot the computer with the new OS.

Configuration involves specifying features, usually stored in a configuration file.

- **System build:** Modifying source code and recompiling the entire OS for a tailored version.
- **Precompiled modules:** Selecting and linking precompiled object modules from a library. Faster, but less tailored.
- **Completely modular:** Selection occurs at execution time by setting parameters.

Most modern OSes for desktops, laptops, and mobile devices use the second approach, combining specific hardware generation with modular support (e.g., loadable kernel modules) for dynamic changes.

Building a Linux system from scratch typically involves:

1. Downloading Linux source code (e.g., from `http://www.kernel.org`).
2. Configuring the kernel using `make menuconfig` to generate a `.config` file.
3. Compiling the main kernel using `make`, producing the `vmlinuz` kernel image.
4. Compiling kernel modules using `make modules`.
5. Installing kernel modules into `vmlinuz` using `make modules_install`.
6. Installing the new kernel using `make install`.

The system will run the new OS upon reboot.

Alternatively, Linux can be installed as a virtual machine (VM):

- **Building from scratch:** Similar to building Linux, but without OS compilation.
- **VM appliance:** Downloading a pre-built and configured OS appliance and installing it with virtualization software (e.g., VirtualBox, VMware).

Example for building a VM: download Ubuntu ISO, use VM software to boot from ISO, answer installation questions, install and boot.

System boot

The process of starting a computer by loading the kernel is known as **booting**.

1. A small piece of code (**bootstrap program** or **boot loader**) locates the kernel.
2. The kernel is loaded into memory and started.
3. The kernel initializes hardware.
4. The root file system is mounted.

Multistage boot process:

- Initial boot loader in nonvolatile firmware (**BIOS**) loads a second boot loader from a **boot block** on disk.
- **UEFI** (Unified Extensible Firmware Interface) is a modern replacement for BIOS, offering better 64-bit/large disk support and faster single-stage booting.

The bootstrap program performs diagnostics, initializes system aspects (CPU registers, device controllers, memory), starts the OS, and mounts the root file system. The system is then considered **running**.

GRUB is an open-source bootstrap program for Linux/UNIX.

- Boot parameters are set in a GRUB configuration file.
- Allows runtime changes (kernel parameters, selecting different kernels).
- Linux kernel image is compressed; extracted after loading.
- Creates a temporary RAM file system (`initramfs`) for necessary drivers/modules.
- Switches to the real root file system after drivers are installed.
- Creates the `systemd` process (initial process), then starts other services.

Booting mechanisms are boot loader-dependent (e.g., specific GRUB versions for BIOS/UEFI).

Mobile system boot (e.g., Android):

- Different from PCs; vendors provide boot loaders (e.g., LK for Android).

- Uses compressed kernel image and initial RAM file system.
- Android maintains `initramfs` as the root file system (unlike Linux, which discards it).
- Starts the `init` process and creates services before displaying the home screen.

Most OS boot loaders (Windows, Linux, macOS, iOS, Android) provide **recovery mode** or **single-user mode** for diagnostics and repairs.

Section glossary

Term	Definition
system build	Creation of an operating-system build and configuration for a specific computer site.
booting	The procedure of starting a computer by loading the kernel.
bootstrap program	The program that allows the computer to start running by initializing hardware and loading the kernel.
BIOS	Code stored in firmware and run at boot time to start system operation.
boot block	A block of code stored in a specific location on disk with the instructions to boot the kernel stored on that disk.
unified extensible firmware interface (UEFI)	The modern replacement for BIOS containing a complete boot manager.
running	The state of the operating system after boot when all kernel initialization has completed and system services have started.
GRUB	A common open-source bootstrap loader that allows selection of boot partitions and options to be passed to the selected kernel.
recovery mode	A system boot state providing limited services and designed to enable the system admin to repair system problems and debug system startup.
single-user mode	A system boot state providing limited services and designed to enable the system admin to repair system problems and debug system startup.

2.10 Operating-system debugging

Debugging is the activity of finding and fixing errors (bugs) in a system, including hardware and software. It can also involve **performance tuning** to remove processing **bottlenecks**. This section focuses on debugging process and kernel errors, and performance problems.

Failure analysis

When a process fails, operating systems typically:

- Write error information to a **log file**.
- Create a **core dump** (a capture of the process's memory) for later analysis.
- Debuggers can then probe running programs and core dumps.

Kernel debugging is more complex due to its size, control of hardware, and lack of user-level tools.

- A kernel failure is a **crash**.
- Error information is saved to a log file, and memory state to a **crash dump**.
- Kernel memory state is often saved to a dedicated disk section without a file system to ensure data integrity during a crash.

Performance monitoring and tuning

Performance tuning aims to improve system efficiency by removing bottlenecks. This requires monitoring system behavior, using either **counters** or **tracing**.

Counters

Operating systems track activity via counters (e.g., system calls, network/disk operations).

- **Per-process Linux tools:**
  - `ps`: Reports information for processes.
  - `top`: Reports real-time statistics for current processes.
- **System-wide Linux tools:**
  - `vmstat`: Reports memory-usage statistics.
  - `netstat`: Reports network interface statistics.
  - `iostat`: Reports disk I/O usage.
- Most Linux counter tools read statistics from the `/proc` pseudo file system (kernel memory).
- **Windows: Windows Task Manager** provides information on applications, processes, CPU/memory usage, and networking statistics.

Tracing

Tracing tools collect data for specific events (e.g., system-call invocation steps).

- **Per-process Linux tools:**
  - `strace`: Traces system calls invoked by a process.
  - `gdb`: A source-level debugger.
- **System-wide Linux tools:**
  - `perf`: Collection of Linux performance tools.
  - `tcpdump`: Collects network packets.

BCC

BCC (BPF Compiler Collection) is a toolkit for dynamic kernel tracing in Linux, providing a dynamic, secure, low-impact debugging environment.

- **Functionality:** Front-end interface to eBPF (extended Berkeley Packet Filter) tool.
- **eBPF:** Programs written in a subset of C, compiled into eBPF instructions, dynamically inserted into a running Linux system to capture events or monitor performance.
- **Verifier:** Checks eBPF instructions for system performance/security impact before insertion.
- **BCC tools:** Written in Python, embedding C code for eBPF instrumentation. Compiles C to eBPF instructions and inserts using probes or tracepoints.
- **Example:** `disksnoop.py` traces disk I/O activity (timestamp, R/W, bytes, latency).
- **Power:** Can be used on live production systems without harm, useful for system administrators to identify bottlenecks or security exploits.

Section glossary

Term	Definition
debugging	The activity of finding and removing errors.
performance tuning	The activity of improving performance by removing bottlenecks.
bottleneck	A performance-limiting aspect of computing.
log file	A file containing error or ”logging” information.
core dump	A copy of the state of a process written to disk when it crashes.
crash	Termination of execution due to a problem (kernel or process).
crash dump	A copy of the state of the kernel written to disk during a crash.
BPF compiler collection (BCC)	A rich toolkit for tracing system activity on Linux for debugging and performance-tuning.

2.11 Summary

- An operating system provides an environment for the execution of programs by providing services to users and programs.
- The three primary approaches for interacting with an operating system are (1) command interpreters, (2) graphical user interfaces, and (3) touch-screen interfaces.
- System calls provide an interface to the services made available by an operating system. Programmers use a system call's application programming interface (API) for accessing system-call services.
- System calls can be divided into six major categories: (1) process control, (2) file management, (3) device management, (4) information maintenance, (5) communications, and (6) protection.
- The standard C library provides the system-call interface for UNIX and Linux systems.
- Operating systems also include a collection of system programs that provide utilities to users.
- A **linker** combines several relocatable object modules into a single binary executable file. A **loader** loads the executable file into memory, where it becomes eligible to run on an available CPU.
- There are several reasons why applications are operating-system specific. These include different binary formats for program executables, different instruction sets for different CPUs, and system calls that vary from one operating system to another.
- An operating system is designed with specific goals in mind. These goals ultimately determine the operating system's policies. An operating system implements these policies through specific mechanisms.
- A **monolithic** operating system has no structure; all functionality is provided in a single, static binary file that runs in a single address space. Although such systems are difficult to modify, their primary benefit is efficiency.
- A **layered** operating system is divided into a number of discrete layers, where the bottom layer is the hardware interface and the highest layer is the user interface. Although layered software systems have had some success, this approach is generally not ideal for designing operating systems due to performance problems.
- The **microkernel** approach for designing operating systems uses a minimal kernel; most services run as user-level applications. Communication takes place via message passing.
- A **modular** approach for designing operating systems provides operating-system services through modules that can be loaded and removed during run time. Many contemporary operating systems are constructed as **hybrid systems** using a combination of a monolithic kernel and modules.
- A **boot loader** loads an operating system into memory, performs initialization, and begins system execution.
- The performance of an operating system can be monitored using either **counters** or **tracing**. Counters are a collection of system-wide or per-process statistics, while tracing follows the execution of a program through the operating system.