Chapter 02 Operating-System Structures

The basic structures in operating systems

Outline

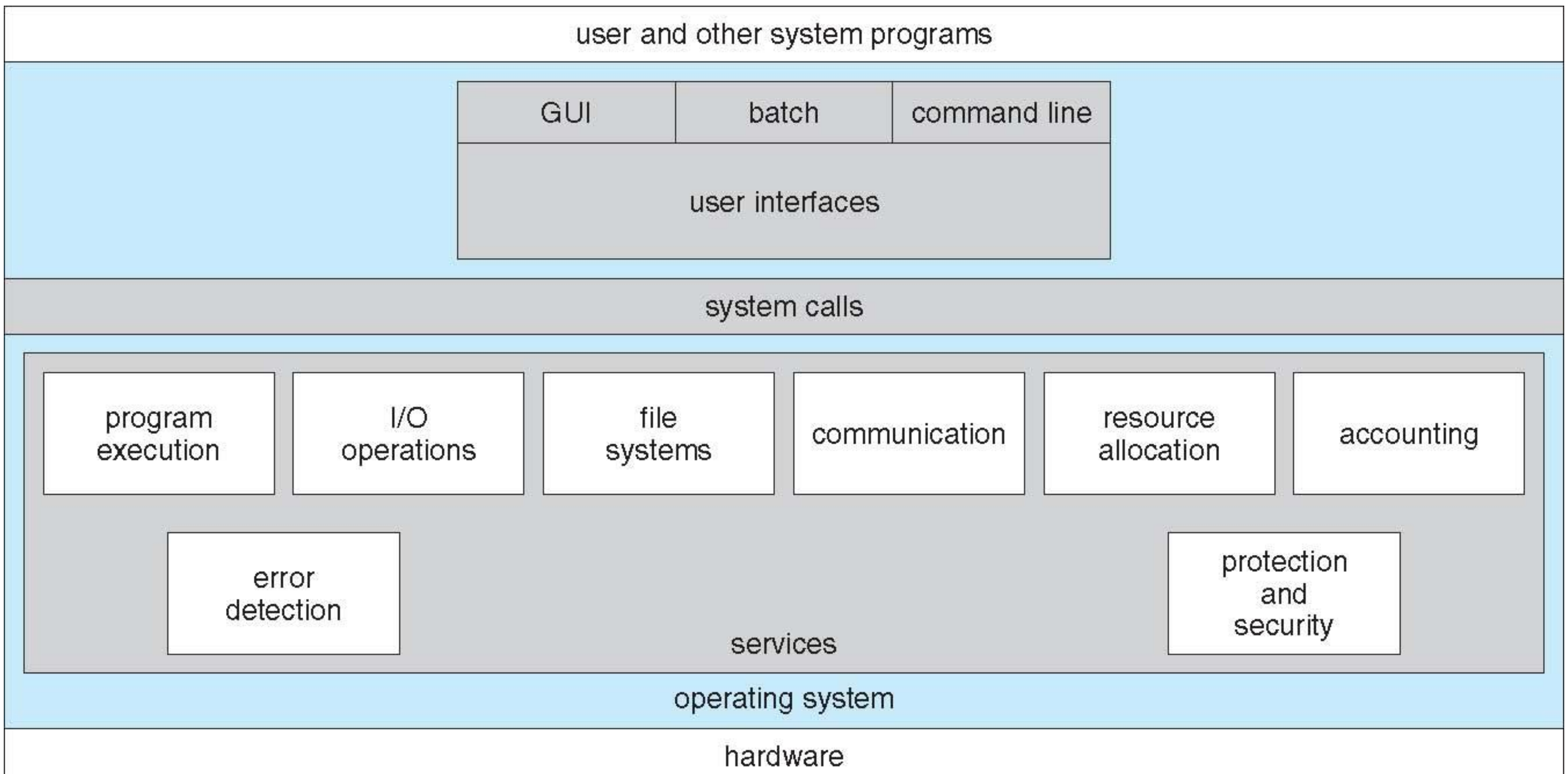
- Operating-System Services
- User Operating-System Interface
- System Calls
- System Services
- Linkers and Loaders
- Why Applications Are Operating-System Specific
- Operating-System Design and Implementation
- Operating-System Structure
- Building and Booting an Operating System
- Operating-System Debugging

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.

Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users



Operating System Services

- One set of operating-system services provides functions that are helpful to the user:
  1. **User interface** – Differs in their forms
     - Graphics User Interface (GUI): window system with mouse and keyboard
     - Touch-Screen Interface: used in mobile system
     - Command-Line (CLI): text commands and keyboard
  2. **Program execution** - The system must be able to **load** a program into memory and to **run** that program, **end** execution, either normally or abnormally (indicating error)
  3. **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
     - Provide means to do I/O

Operating System Services (Cont.)

4. **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
5. **Communications** – Processes may exchange information, on the same computer or between computers over a network
   - Communications may be via shared memory or through message passing (packets moved by the OS)
6. **Error detection** – OS needs to be constantly aware of possible errors
   - May occur in the CPU and memory hardware, in I/O devices, in user program
   - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
   - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

Operating System Services (Cont.)

- Another set of operating-system functions for ensuring the efficient operation of the system

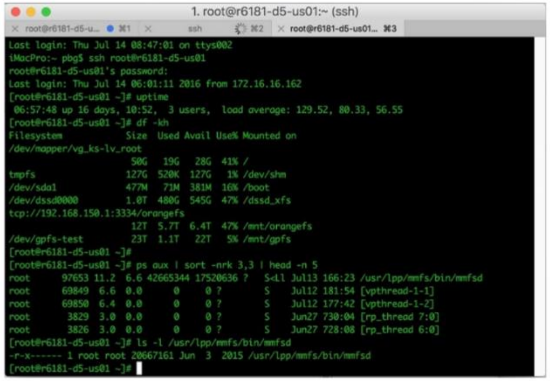
1. **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
   - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
2. **Logging** - To keep track of which users use how much and what kinds of computer resources
   - Accounting and accumulating usage statistics
3. **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
   - Protection involves ensuring that all access to system resources is controlled
   - Security of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

Exercises

- List five services provided by an operating system, and explain how each creates convenience for users. In which cases would it be impossible for user-level programs to provide these services? Explain your answer.

User and Operating-System Interface

- Command-Line (CLI) or command interpreter allows direct command entry
  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple flavors implemented – shells
  - Primarily fetches a command from user and executes it in two general ways:
    - The command interpreter itself contains the code to execute the command
      - Jump to a section of its code and make appropriate system call
    - The commands are implemented through system programs
      - E.g. UNIX “rm file.txt” command
      - Search file “rm”, load and execute with parameter “file.txt”

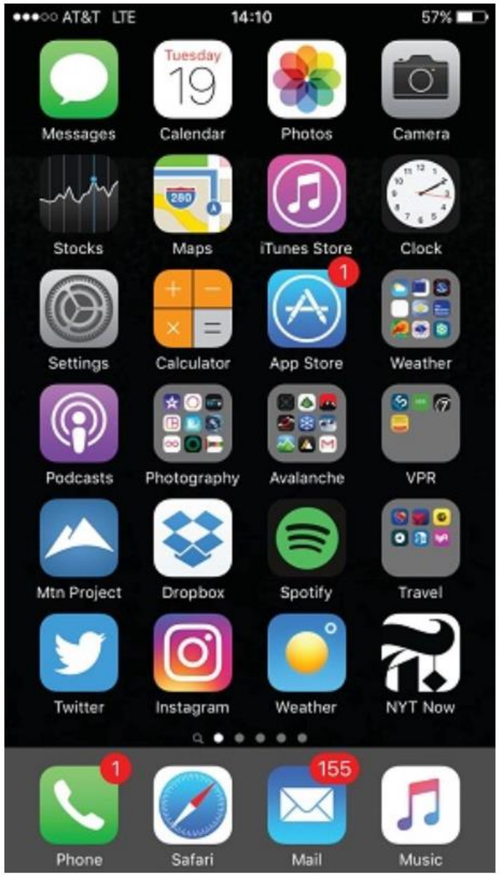


User and Operating-System Interface

- Graphical User Interface (GUI)
  - User-friendly desktop metaphor interface
    - Usually mouse, keyboard, and monitor
    - Icons represent files, programs, actions, etc
    - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)
    - Invented at Xerox PARC
  - Many systems now include both CLI and GUI interfaces
    - Microsoft Windows is GUI with CLI "command" shell
    - Apple Mac OS X is "Aqua" GUI interface with UNIX kernel underneath and shells available
    - Unix and Linux have CLI with optional GUI interfaces
      - E.g. *K Desktop Environment (or KDE)* and the *GNOME* desktop

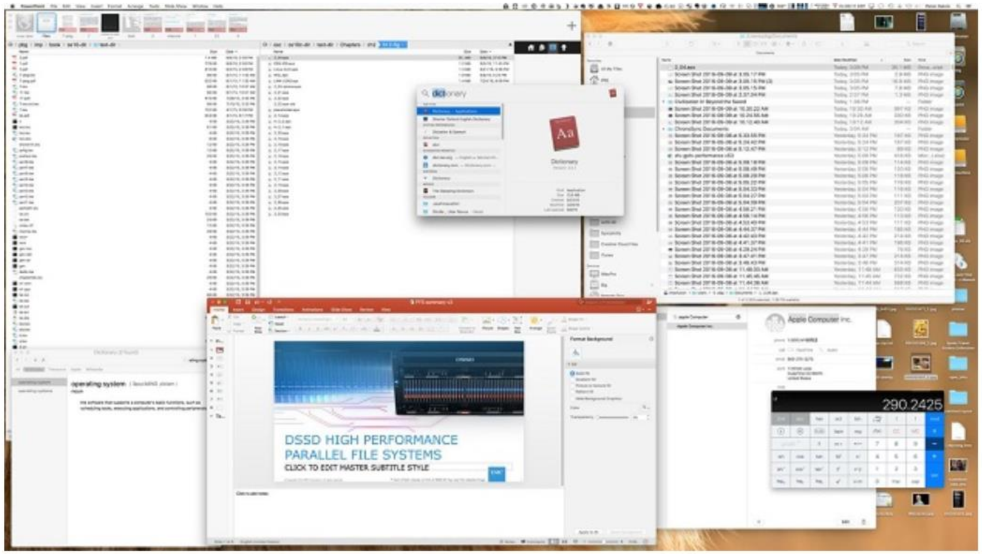
User and Operating-System Interface

- Touchscreen Interfaces
  - Touchscreen devices require new interfaces
    - Mouse not possible or not desired
    - Actions and selection based on gestures
    - Virtual keyboard for text entry



User and Operating-System Interface

- Choice of Interface
  - Personal preference
  - CLI
    - For system administrator and power users
    - Usually make repetitive tasks easier such as shell scripts
  - GUI
    - Most Windows users



Exercises

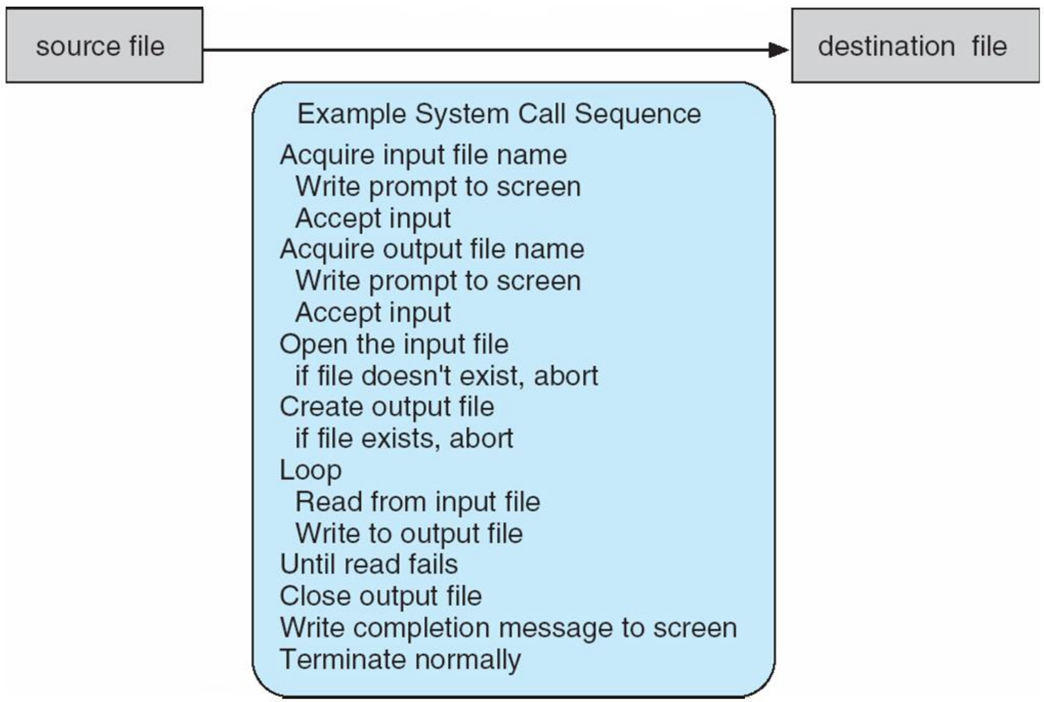
- What is the purpose of the command interpreter? Why is it usually separate from the kernel?
- Give three approaches for interacting with an operating system.

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Some written in assembly-language for accessing hardware

System Calls

- Example of System Calls
  - System call sequence to copy the contents of one file to another file
    - by the command “cp in.txt out.txt”
    - Ask user for the names in an interactive system

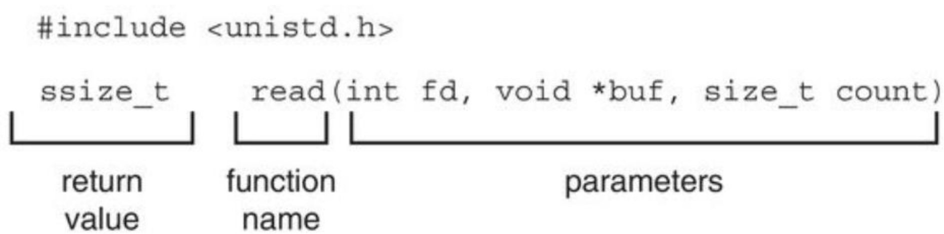


System Calls

- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use
- Three most common APIs are the Windows API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and MacOS), and Java API for the Java virtual machine (JVM)
  - Invoke actual system calls
  - E.g. CreateProcess() in Windows invokes NTCreateProcess() system call in the Windows kernel.

System Calls

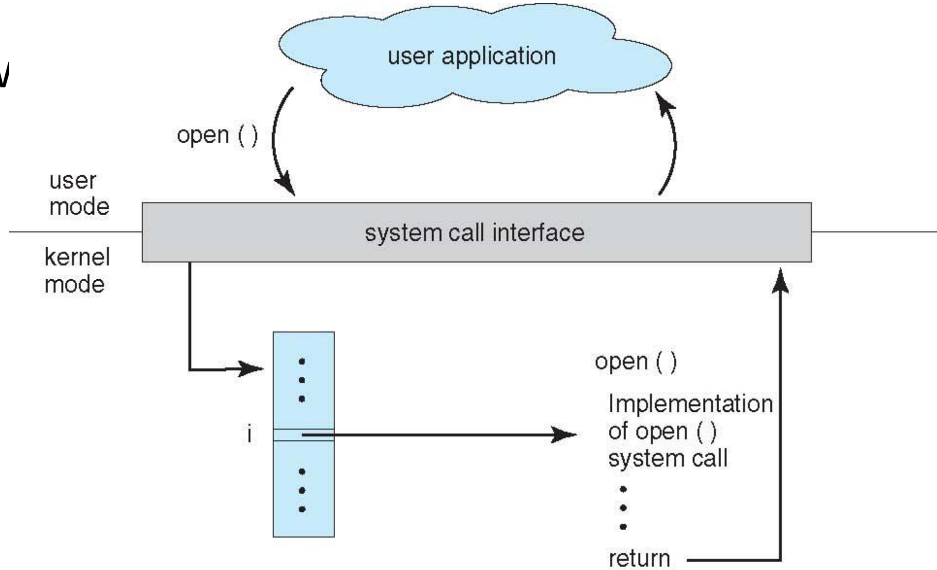
- Example of Standard API
  - Description obtain by “man read” command
  - Three parameters are fd, buf, count
  - A return value



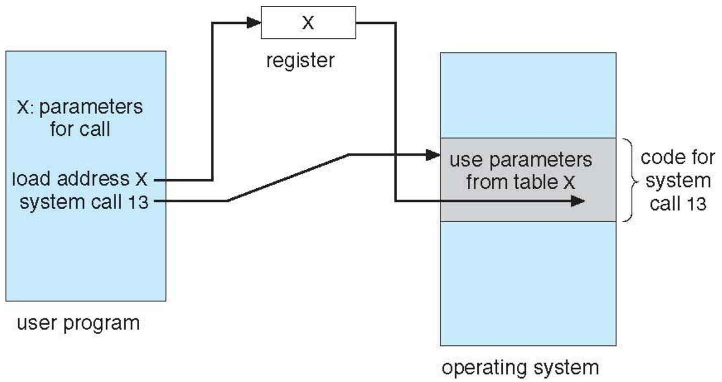
- Why programmer favors API rather than system call?
  - Portability
  - System call often more detailed and difficult to work

System Calls

- Run-time Environment (RTE)
  - Full suit of software including compilers or interpreters as well as other software, such as libraries and loaders
  - Provide system-call interface
    - A link to the system call
    - A number associated with each system call maintains in a table
  - The caller need know nothing about how the system call is implemented
    - Just obey the API and know the result
    - Most details of the operating-system interface are hidden



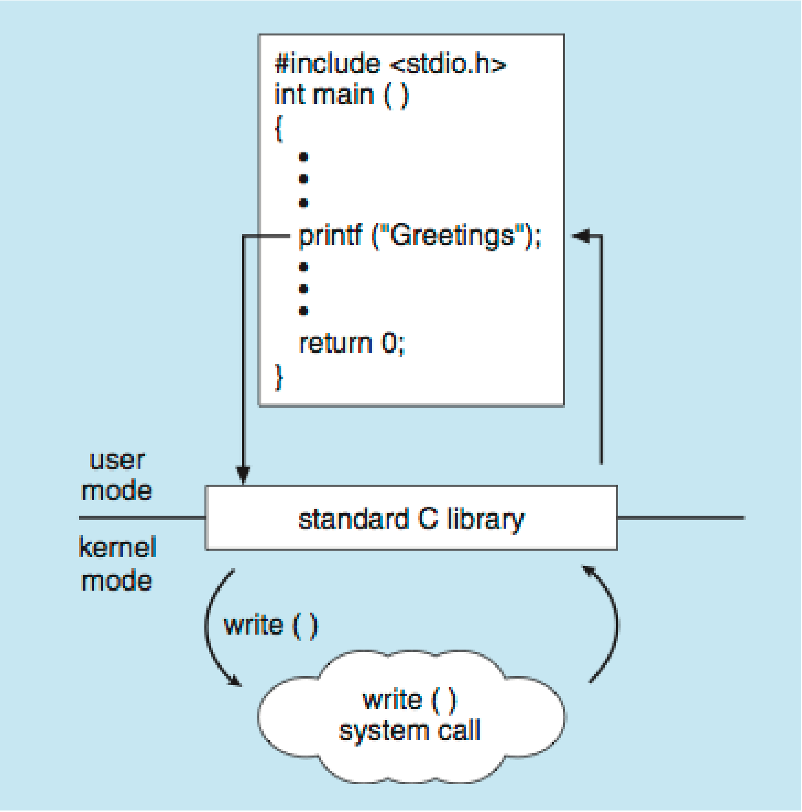
System Calls



- Parameter Passing
  - Often, more information is required han simply identity of desired system call
    - Exact type and amount of information vary according to OS and call
  - Three general methods used to pass parameters to the OS
    - Simplest: pass the parameters in registers
      - In some cases, may be more parameters than registers
    - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
      - This approach taken by Linux and Solaris
    - Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
    - Block and stack methods do not limit the number or length of parameters being passed

System Calls

- Standard C Library Example - C program invoking printf() library call, which calls write() system call



System Calls

- Types of System Calls
  - Process control
    - **create process, terminate process**
    - load, execute
    - get process attributes, set process attributes
    - wait event, signal event
    - allocate and free memory

System Calls

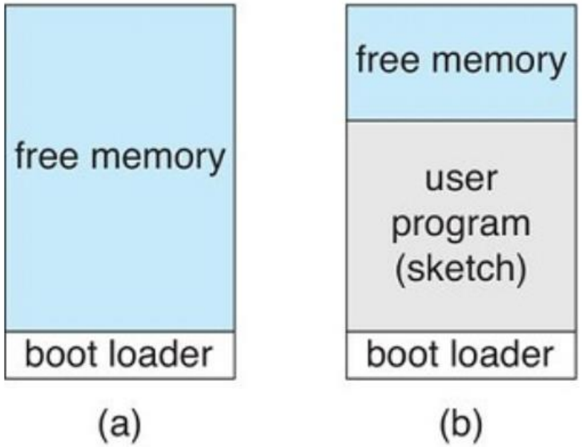
- Types of System Calls
  - Process control
    - create process, terminate process
    - **load, execute**
    - **get process attributes, set process attributes**
    - wait event, signal event
    - allocate and free memory

System Calls

- Types of System Calls
  - Process control
    - create process, terminate process
    - load, execute
    - get process attributes, set process attributes
    - **wait event, signal event**
    - **allocate and free memory**

System Calls

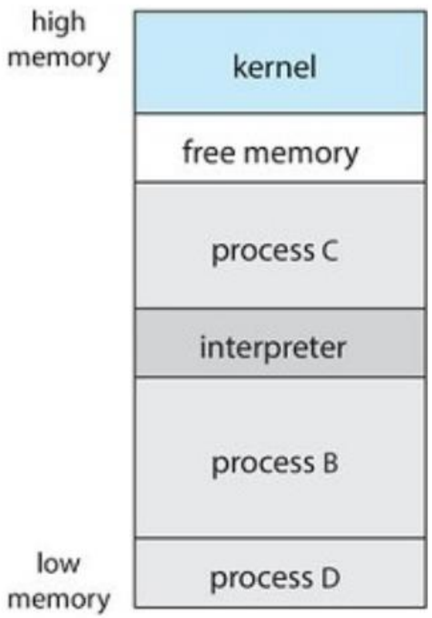
- Process Control Example
  - Write a program for Arduino
    - Write program on PC
    - Upload compiled program known as sketch
    - Boot loader in Arudino loads the sketch to memory



Arduino execution. (a) At system startup. (b) Running a sketch.

System Calls

- Process Control Example
  - FreeBSD as an example of multitasking system
    - Run selected shell
    - Interpreter continue running
    - A new process can be started by fork()
    - Selected program will be loaded by exec()
      - Runs in the foreground or background
    - Terminate by exit()



System Calls

- Types of System Calls
  - File management
    - create file, delete file
    - open, close
    - read, write, reposition
    - get file attributes, set file attributes

System Calls

- Types of System Calls
  - Device management
    - request device, release device
    - read, write, reposition
    - get device attributes, set device attributes
    - logically attach or detach devices

System Calls

- Types of System Calls
  - Information maintenance
    - get time or date, set time or date
    - get system data, set system data
    - get process, file, or device attributes
    - set process, file, or device attributes

System Calls

- Types of System Calls
  - Communications
    - create, delete communication connection
    - send, receive messages
    - transfer status information
    - attach and detach remote devices
  - Two models for inter-process communication
    - Message-passing model - the communicating processes exchange messages with one another to transfer information.
      - Build connection via host name / process name
      - Useful for small amount of data
      - Easier to intercomputer communication
    - Shared-memory model - use shared memory to transfer information
      - Maximum speed due to memory transfer
      - Convenience of communication within a computer

System Calls

- Types of System Calls
  - Protection
    - get file permissions
    - set file permissions

Exercises

- What is the purpose of system calls? Give six major categories of system calls.

- What system calls have to be executed by a command interpreter or shell in order to start a new process on a UNIX system?

System Services

- A.k.a. system utilities
- Provide a convenient environment for program development and execution.
- They can be divided into:
  - File management
  - Status information
  - File modification
  - Programming-language support
  - Program loading and execution
  - Communications
  - Background services
  - Application program

System Services

- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a registry - used to store and retrieve configuration information

System Services

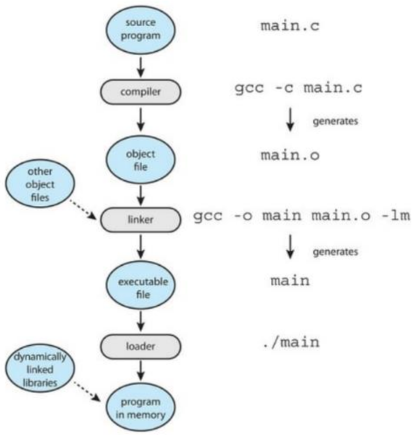
- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another′ s screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

System Services

- **Background Services**
  - Launch at boot time
    - Some for system startup, then terminate
    - Some from system boot to shutdown
  - Provide facilities like disk checking, process scheduling, error logging, printing
  - Run in user context not kernel context
  - Known as services, subsystems, daemons
- **Application programs**
  - Don't pertain to system
  - Run by users
  - Not typically considered part of OS
  - Launched by command line, mouse click, finger poke

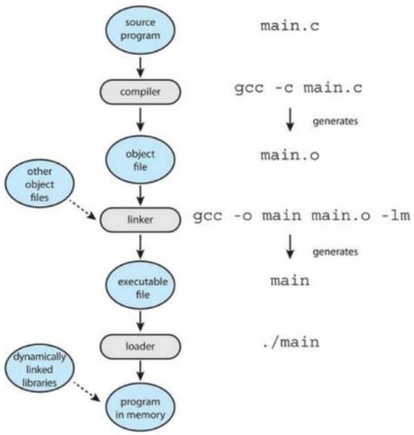
The view of the operating system seen by most users is defined by the application and system programs, rather than by the actual system calls.

Linker and Loaders



- A program resides on disk as a binary executable file
- Source files are compiled into object files which a format known as an relocatable object file
- Linker combines relocatable object files into a binary executable file
- Loader loads binary executable file into memory
  - Relocation assigns final addresses to the program parts and adjusts code and data in the program to match those addresses

Linker and Loaders



- Dynamically Linked Libraries (DLLs)
  - Avoid link and load libraries not used
  - Link and load during run time
  - Linker inserts relocation information for allowing dynamically linked and loaded
  - Saving memory use
- Format of Executable
  - Object files contain compiled machine code and a symbol table containing metadata about functions and variables that are referenced
  - Unix's Executable and Linkable Format (ELF), Windows' Portable Executable (PE) format, and macOS's Mach-O format.

Why Applications Are Operating-System Specific?

- Applications compiled on one operating system are not executable on other operating systems.
  - Unique set of system calls
- An application can be made available to run on multiple operating systems in one of three ways:
  - Application written in an interpreted language
  - Application written in a language that includes a virtual machine containing the running application
  - Application developer uses a standard language or API in which the compiler generates binaries in a machine- and operating-system-specific language.

Why Applications Are Operating-System Specific?

- However, the general lack of application mobility has several causes.
  - Application level - the libraries provided with the operating system contain APIs to provide features but these will not work on an operating system that does not provide those APIs
  - Lower levels
    - Each operating system has a binary format for applications
    - CPUs have varying instruction set
    - Different system calls
- A solution is the ELF format for binary executable files.

Exercises

- Why applications are operating-system specific? Give three reasons.

Operating System Design and Implementation

- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- User goals and System goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- No unique solution
  - Different requirements to different solutions
  - Follow general principles in software engineering

Operating System Design and Implementation

- Important principle to separate
  Policy: What will be done?
  Mechanism: How to do it?
- Mechanisms determine how to do something, policies decide what will be done
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)

Implementation

- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to port to other hardware
  - Slower but not a major issue
  - Major performance improvements may come from better data structures and algorithms

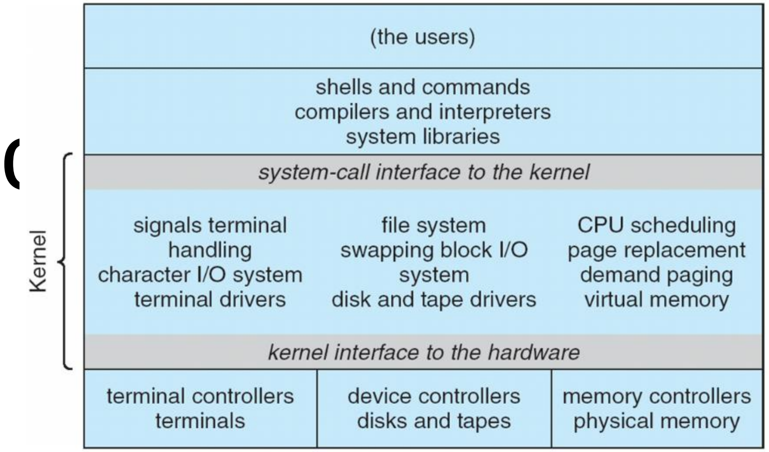
Exercise

- Briefly describe the relations of operating system goals, policies, and mechanism.

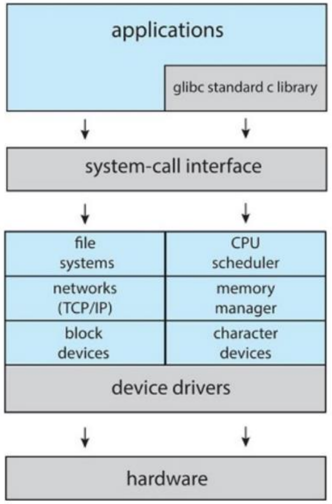
Operating System Structure

- General-purpose OS is very large program
- Various ways to structure ones
  - Monolithic Structure - UNIX
  - Layered Approach – an abstraction
  - Microkernel –Mach

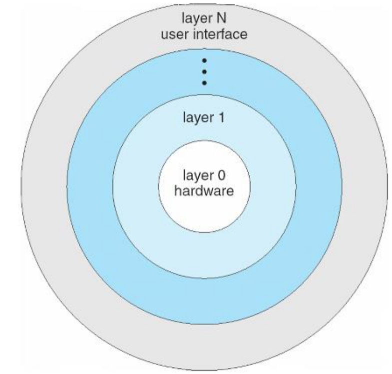
Operating System Struc



- **Monolithic Structure - UNIX**
  - No structure at all
  - Original UNIX operating system includes two parts:
    - System programs
    - The kernel
      - Consists of everything below the system-call interface and above the physical hardware
      - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level
  - Linux has similar structures
  - Difficult to implement and extend
  - Little overhead in the system-call interface
  - Tightly coupled
    - Small change causes wide-ranging effect

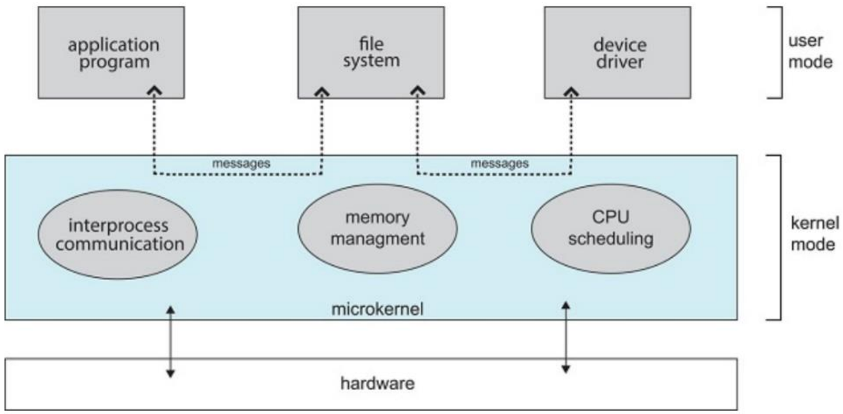


Operating System Structure

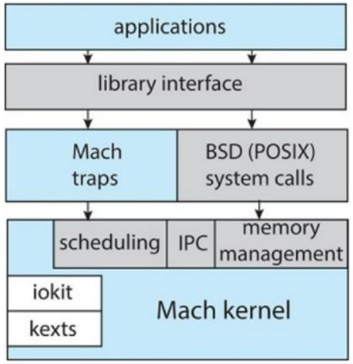


- Layered Approach
  - Loosely coupled system
    - Divided into small components with limited functionality
    - These components are composed of the kernel
  - The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
  - With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers
  - Simplicity of construction and debugging
    - Within layer debugging
    - Taking advantage of abstraction
  - Few OSes use pure layered approach
    - Hard to define the functionality of each layer
    - Pass multiple layers for OS service leads poor performance
    - A well-known example is TCP/IP

Operating System S

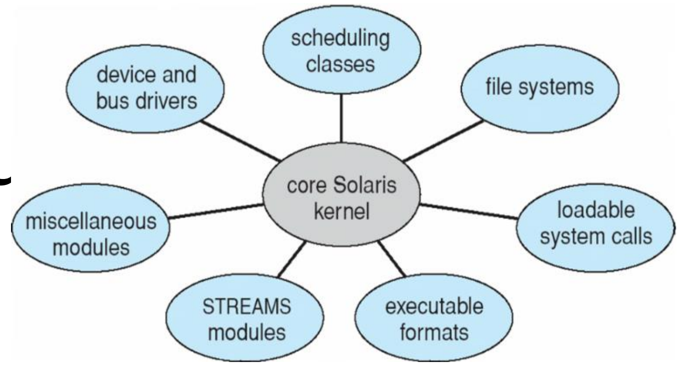


- Microkernel –Mach
  - Small kernel
    - Moves nonessential components as much from the kernel into user space
  - Which should reside in the kernel?
    - Process management, memory management, and communication facility
  - Communication takes place between user modules using message passing
  - Benefits:
    - Easier to extend a microkernel
    - Easier to port the operating system to new architectures
    - More secure and reliable (less code is running in kernel mode)
  - Detriment:
    - Performance overhead of user space to kernel space communication
    - Copying messages reside in separate address spaces
    - Switching between process
  - Examples:
    - Mach example of microkernel
    - Mac OS X kernel (Darwin) partly based on Mach
    - Windows NT to Windows NT 4.0



Darwin

Operating System Structu



- Modules
  - Many modern operating systems implement loadable kernel modules (LKMs)
    - Uses object-oriented approach
    - Each core component is separate
    - Each talks to the others over known interfaces
    - Each is loadable as needed within the kernel
  - Similar to layers but with more flexible
    - Module can call other modules
  - Similar to microkernel with more efficient
    - Communicate without message passing
  - Examples
    - Linux, macOS, Solaris, and Windows

Hybrid Systems

- Most modern operating systems are actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem ***personalities***

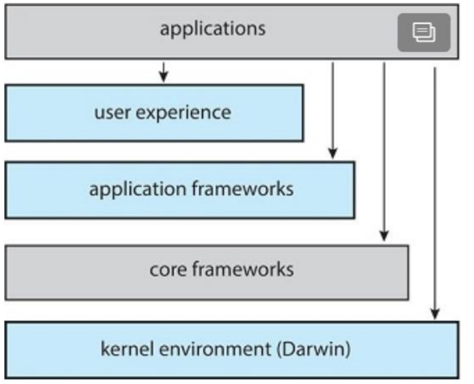
Exercise

- Briefly describe monolithic operating system. Give one pro and con for such system.

- Briefly describe layered operating system. Give one pro and con for such system.

- Give three characteristics for microkernel operating system.

macOS and iOS

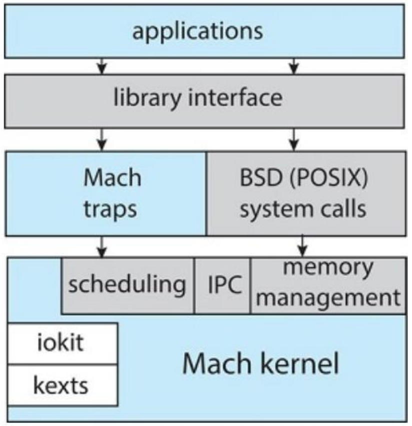


- General architecture of macOS and iOS
  - User experience layer - software interface allowing users to interact with the computing devices
    - macOS uses the Aqua user interface for a mouse or trackpad,
    - iOS uses the Springboard user interface for touch devices.
  - Application frameworks layer - provide an API for the Objective-C and Swift programming languages
    - Including the Cocoa and Cocoa Touch frameworks
    - The former for developing macOS applications, and the latter by iOS to support hardware features such as touch screens
  - Core frameworks - defining frameworks that support graphics and media including, Quicktime and OpenGL.
  - Kernel environment
    - Also known as Darwin
    - Including the Mach microkernel and the BSD UNIX kernel
  - Applications
    - Able to interact directly with all layers

macOS and iOS

- Differences between macOS and iOS
  - macOS for desktop and laptop computer
  - iOS for mobile devices
  - Architecture on which is compiled
    - macOS on Intel
    - iOS on ARM-based
  - iOS features the needs of mobile systems
    - Power management, aggressive memory management
    - More strict security settings
    - More restricted to developers

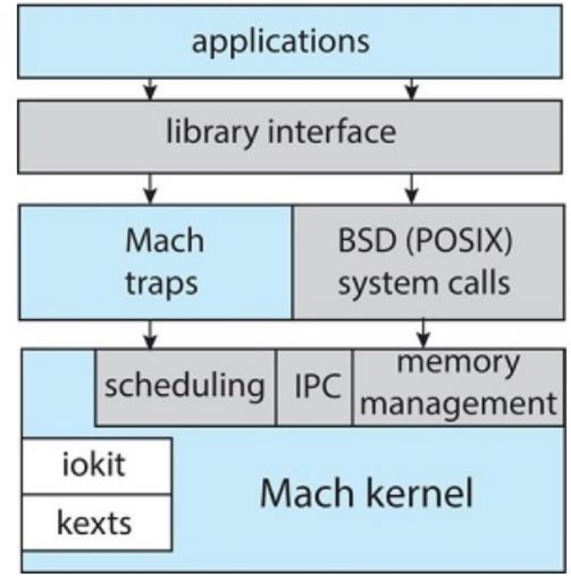
macOS and iOS



Darwin

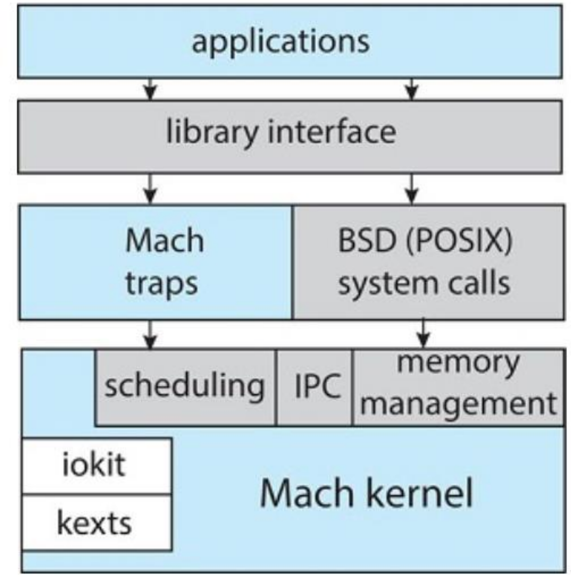
- Mac OS X kernel: Darwin
  - Two rather than one system-call interfaces
    - Mach traps
    - BSD system calls
    - Provide libraries for C, networking, security, and programming language support …
  - Beneath are OS services
    - Memory management, CPU scheduling, and interprocess communication (IPC)
  - Functionality provided by Mach through kernel abstraction
    - Include Mach process, threads, memory objects and ports
  - Kernel environments
    - Provide I/O kit and Kernel extensions (kexts)
    - For development of device drivers and dynamically loadable modules
  - Performance issue addressed
    - Combining Mach, BSD, the I/O kit, and any kernel extensions into a single address space
    - Message passing without copying

Android



- Developed by Open Handset Alliance (mostly Google)
  - Open Source
- Similar stack to IOS
- Runtime environment
  - Use Android API or Java native interface (JNI) for Java applications
  - Compiled to Android Runtime ART
    - Convert to java bytecode .class first
    - Then executable .dex (native machine code)
  - Ahead-of-time (AOT) compilation instead of just-in-time (JIT)
    - Efficient for power consumption

Android



- Libraries include frameworks for web browser (webkit), database (SQLite), network (secure sockets, SSLs)
- Hardware abstraction layer (HAL) for running on most devices
  - Provide consistent view of hardware
- Develop Bionic standard C library
  - Small memory footprint for slow CPUs
  - Bypass GPL licensing of the GNU C library (glibc)
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
  - New form of IPC called Binder

Exercise

- How are iOS and Android similar? How are they different?
- Explain why Java programs running on Android systems do not use the standard Java API and virtual machine.

Building and Booting an Operating System

- Operating-System Generation
  - Operating systems are designed to run on any of a class of machines
  - Build an operating system from the beginning
    1. Obtain the operating system source code
    2. Configure the operating system for the system where it will run
       - Parameter stored in configuration file
       - Can be used to modify OS source code
    3. Compile the operating system (system build)
    4. Install the operating system
    5. Boot the computer and its new operating system

Building and Booting an Operating System

- System Boot
  - Consist of following steps generally:
    1. A small piece of code known as the 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.
  - Some are multistage boot process
    - Small boot loader known as BIOS in nonvolatile firmware
    - BIOS loads the second boot loader from boot block
    - The second boot loader then loads the entire OS into memory
  - Recent systems replace BIOS with Unified Extensible Firmware Interface (UEFI)
    - Complete boot manager
    - Better support 64-bit systems and larger disks
    - Faster
  - After kernel loads, the system is running.

Building and Booting an Operating System

- Common bootstrap loader GRUB
  - Allows selection of kernel from multiple disks, versions, kernel options
  - Specific versions for BIOS and UEFI
    - Booting mechanism also depends on the boot loader
  - Linux create temporary RAM file system (**`initramfs`**)
    - For necessary kernel modules and drivers
    - To support the real root file system
    - Discard when finished, and create **`systemd`** process
- Android boot loader up to vendors
  - E.g. little kernel (LK)
  - Maintain **`initramfs`** as the root file system rather than discard
  - Start **`init`** process and create services
- Booting into recovery mode or single-user mode
  - For hardware diagnosing, corrupt file system fixing, OS reinstalling

Exercises

- Give the three tasks that a boot loader performs.
- Why do some systems store the operating system in firmware, while others store it on disk?
- How could a system be designed to allow a choice of operating systems from which to boot? What would the bootstrap program need to do?

Operating-System Debugging

- Debugging
  - Finding and fixing errors, or bugs
  - Also includes performance tuning by removing processing bottlenecks
- Failure Analysis
  - OS generate log files containing error information
  - Failure of an application can generate core dump file capturing memory of the process
  - Operating system failure can generate crash dump file containing kernel memory
  - Kernighan's Law:
    "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Operating-System Debugging

- Performance Monitoring and Tuning
  - To improve performance
  - Must be able to monitor system performance
    - Per-process
    - System-wide
  - Two approaches: counters and tracing

Operating-System Debugging

- Counters
  - Keep track of system activity through a series of counters
    - E.g. #system calls, #operations performed, …
  - Take Linux for example
    - Per-Process
      - ps—reports information for a single process or selection of processes
      - top—reports real-time statistics for current processes
    - System-Wide
      - vmstat—reports memory-usage statistics
      - netstat—reports statistics for network interfaces
      - iostat—reports I/O usage for disks
  - Windows Task Manager
    - Information includes applications, processes, CPU, memory, disk, network usage

Operating-System Debugging

- Tracing
  - Collect data for a specific event
  - To look for statistical trends, profiling provides periodic sampling of instruction pointer
  - Take Linux for example
    - Per-Process
      - **`strace`**—traces system calls invoked by a process
      - **`gdb`**—a source-level debugger
    - System-Wide
      - **`perf`**—a collection of Linux performance tools
      - **`tcpdump`**—collects network packets

Exercise

- Give two approach for monitoring an operating system, and briefly describe them.