Chapter I: Overview

1.3. Operating-System Design and Implementation

Outline

- Design and Implementation
- OS Structure
- Building and Booting an OS
- OS Debugging

Objectives

- Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing OSs
- Illustrate the process for booting an OS
- Apply tools for monitoring OS performance
- Design and implement kernel modules for interacting with a Linux kernel

Design and Implementation

- Design and Implementation of OS is not "solvable", but some approaches have proven successful
- Internal structure of different OSs can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- **User** goals and **System** goals
  - User goals – OS should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – OS should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Specifying and designing an OS is highly creative task of **software engineering**

Policy and Mechanism

- **Policy**: **What** needs to be done?
  - Example: Interrupt after every 100 seconds
- **Mechanism**: **How** to do something?
  - Example: timer
- Important principle: separate policy from mechanism
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later.
  - Example: change 100 to 200

Implementation

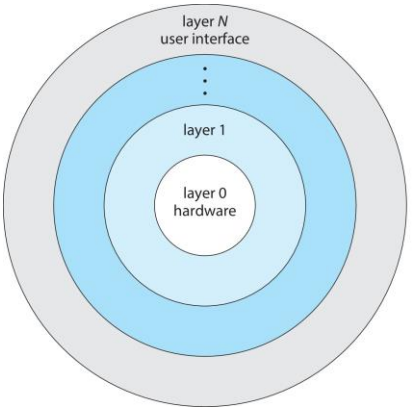
- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually, usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to **port** to other hardware
  - But slower
- **Emulation** can allow an OS to run on non-native hardware

OS Structure

- General-purpose OS is very large program
- Various ways to structure ones
  - Layered – an abstraction
  - Simple structure – MS-DOS
  - More complex – UNIX
  - Microkernel – Mach

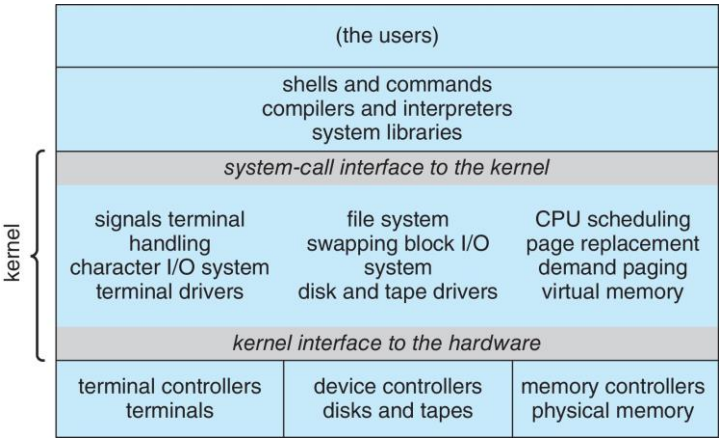
Layered Approach

- The OS 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

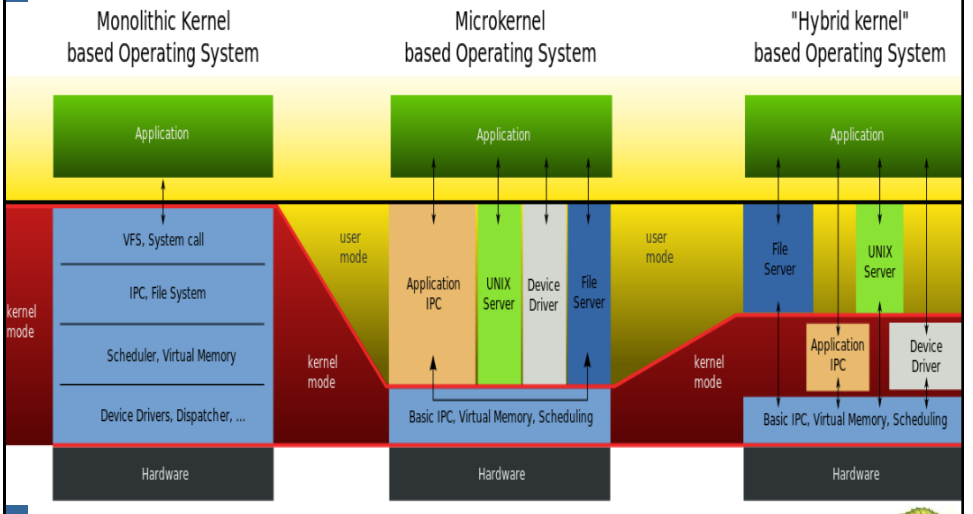


Traditional UNIX System Structure

Beyond simple but not fully layered



Comparison kernel types

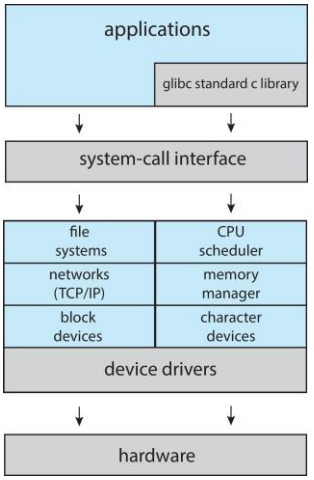


Monolithic Structure – Original UNIX

- UNIX – limited by hardware functionality, the original UNIX OS had limited structuring.
- The UNIX OS consists of two separable parts
  - Systems 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 System Structure

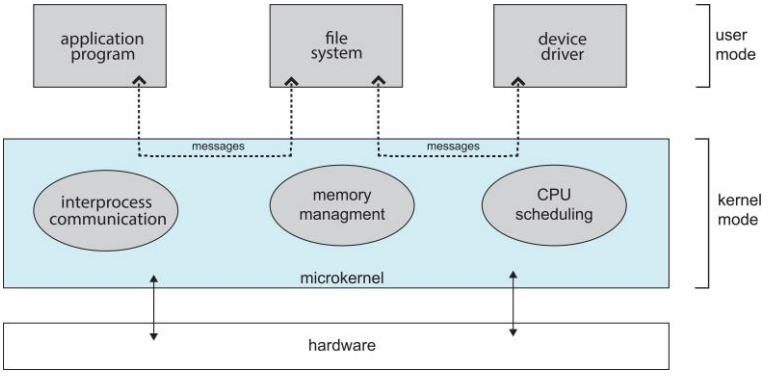
Monolithic plus modular design



Microkernels

- Moves as much from the kernel into user space
- **Mach** is an example of **microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the OS to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

Microkernel System Structure



Modules

- Many modern OSs 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
- Overall, similar to layers but with more flexible
  - Linux, Solaris, etc.

Hybrid Systems

- Most modern OSs are 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***
- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

macOS and iOS Structure

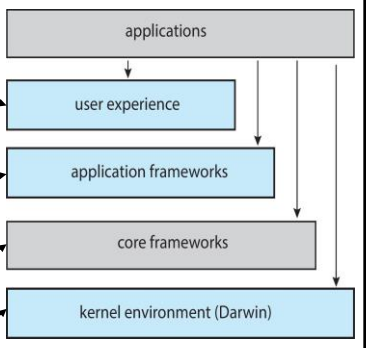
- Apple's macOS: is designed to run primarily on desktop and laptop
- iOS is a mobile operating system designed for the iPhone smartphone and iPad tablet computer.
- Architecturally, macOS and iOS have much in common

defines the software interface of users to devices.
- macOS uses the Aqua user interface – for amouse or trackpad,
- iOS uses the Springboard user interface - for touch devices.

provide an API for the Objective-C and Swift programming languages
- macOS use Cocoa
- iOS use Cocoa Touch to support for hardware features unique to mobile devices

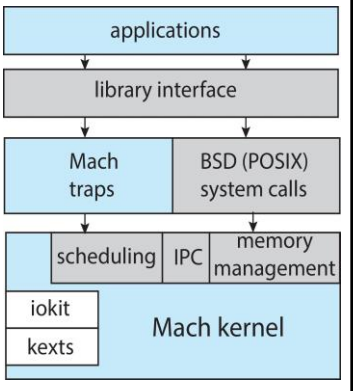
defines frameworks that support graphics and media

also known as **Darwin**,includes the Mach microkernel and the BSD UNIX kernel.



Darwin - a hybrid structure

- Darwin is a layered system that consists
  - the Mach microkernel
  - the BSD UNIX kernel.
- Darwin provides two system-call interfaces:
  - Mach system calls (known as traps):
    - provides fundamental OS
  - BSD system calls
    - provide POSIX functionality
  - I/O kit:
    - for development of device drivers and dynamically loadable modules

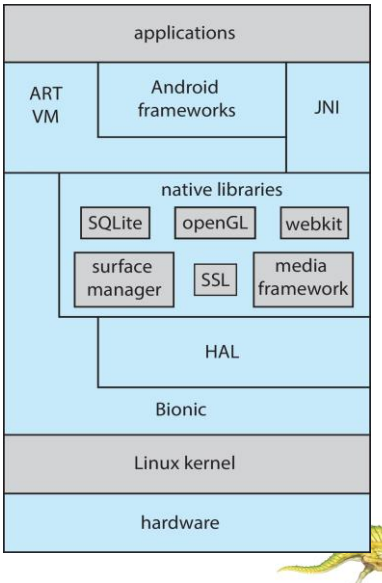


Android

- Developed by Open Handset Alliance (mostly Google)
  - Open Source
- Similar to IOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java plus Android API
    - Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc

Android Architecture

- Google has chosen to abstract the physical hardware through the hard-ware abstraction layer, or HAL.
  - By abstracting all hardware, such as the camera, GPS chip, and other sensors, the HAL provides applications with a consistent view independent of speciic hardware.
  - => allows developers to write programs that are portable across different hardware platforms.



Building and Booting an OS

- OSs generally designed to run on a class of systems with variety of perpherals
- Commonly, OS already installed on purchased computer
  - But can build and install some other OSs
  - If generating an OS from scratch
    - Write the OS source code
    - Configure the OS for the system on which it will run
    - Compile the OS
    - Install the OS
    - Boot the computer and its new OS

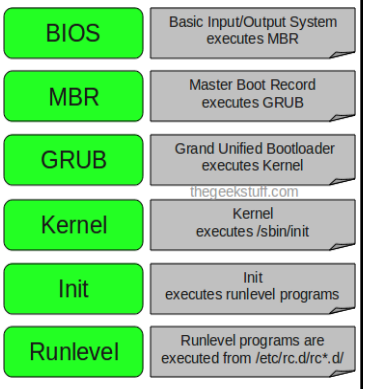
Building and Booting Linux

- Download Linux source code (http://www.kernel.org)
- Configure kernel via "`make menuconfig`"
- Compile the kernel using "`make`"
  - Produces `vmlinuz`, the kernel image
  - Compile kernel modules via "`make modules`"
  - Install kernel modules into `vmlinuz` via "`make modules_install`"
  - Install new kernel on the system via "`make install`"

System Boot

- Process
  - Small piece of code – **bootstrap loader**, **BIOS**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads bootstrap loader from disk
  - Modern systems replace BIOS with **Unified Extensible Firmware Interface** (**UEFI**)
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then **running**
- Boot loaders frequently allow various boot states, such as single user mode

Example: Linux boot



Operating-System Debugging

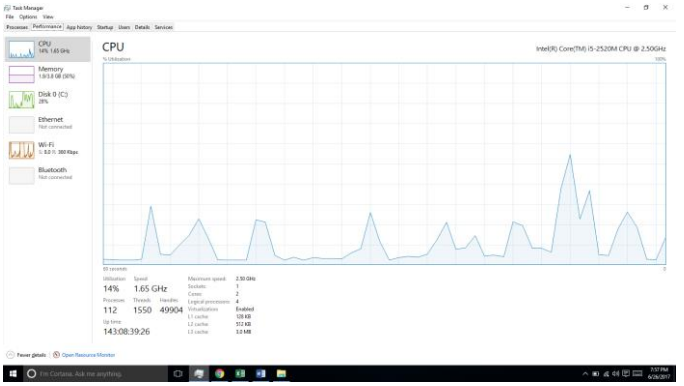
- **Debugging** is finding and fixing errors, or **bugs**
- OS generate **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- OS failure can generate **crash dump** file containing kernel memory

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."

Performance Tuning

- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- Sometimes using ***trace listings*** of activities, recorded for analysis
- **Profiling** is periodic sampling of instruction pointer to look for statistical trends
- Per-Process Tool
  - ps—reports information for a single process or selection of processes
  - top—reports real-time statistics for current processes
- System-Wide Tool
  - vmstat—reports memory-usage statistics
  - netstat—reports statistics for network interfaces
  - iostat—reports I/O usage for disks

Performance Tuning



Tracing

- Collects data for a specific event, such as steps involved in a system call invocation
- Tools include
  - strace – trace system calls invoked by a process
  - gdb – source-level debugger
  - perf – collection of Linux performance tools
  - tcpdump – collects network packets

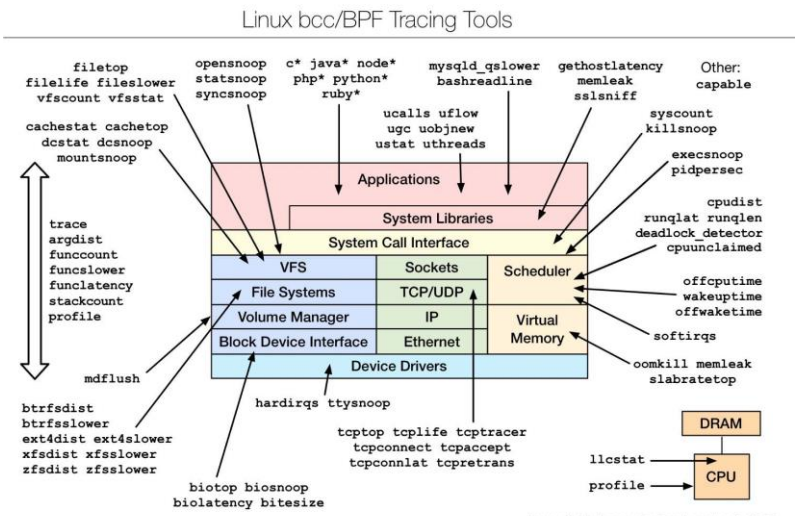
BCC

- Debugging interactions between user-level and kernel code nearly impossible without toolset that understands both and an instrument their actions
- BCC (BPF Compiler Collection) is a rich toolkit providing tracing features for Linux
  - See also the original DTrace
- For example, disksnoop.py traces disk I/O activity

```
TIME(s)          T    BYTES    LAT(ms)
1946.29186700    R    8            0.27
1946.33965000    R    8            0.26
1948.34585000    W    8192         0.96
1950.43251000    R    4096         0.56
1951.74121000    R    4096         0.35
```

- Many other tools (next slide)

Linux bcc/BPF Tracing Tools



End of Chapter 1.3