Overview of the OS

CS 450: Operating Systems
Sean Wallace <swallac6@iit.edu>
Agenda

• What is an operating system?
  • What are its main responsibilities?
    • How does it achieve them?
  • How is an operating system organized?
    • What is an operating system kernel?
What is an OS?
operating system

noun

the software that supports a computer's basic functions, such as scheduling tasks, executing applications, and controlling peripherals.

—New Oxford American Dictionary
tasks & applications = running programs

= Processes

peripherals = I/O devices
• OS duties revolve around aiding and abetting user processes

• Setting up a consistent view of the system (e.g., virtual memory)

• Simplifying access to disparate devices (e.g., open/close/read/write API)
• Problem: there’s never enough hardware to go around
  • OS *multiplexes* hardware (time/space)
  • Must also *isolate* processes from each other (and the OS itself)
Primary OS services:

- *Isolation, hardware abstraction, and concurrency*

  (with another arising from the first: *interaction*)
• How to enforce isolation?
• Two routes: software / hardware
• Is isolation possible solely via software?

• That is, can you write a program (the OS) to execute other (user) programs, and guarantee separation & robustness without hardware support?
Some software attack vectors:

- Address fabrication (e.g., integer-to-address cast for cross-space pointers)
- Buffer overruns (e.g., on syscalls)
- Run-time errors (e.g., intentional/accidental stack overflows)
• Software prevention mechanisms:
  • Static verification (e.g., type-checking)
    • Programs must “pass” to be run
  • Run-time tools (e.g., garbage collection, exception handling, etc.)
• Is isolation possible solely via software?

• Maybe…but difficult/impractical!

• The popular approach (all commercial OS’s) is to rely on hardware support.
• e.g., Intel x86 architecture provides a 2-bit *current privilege level* (CPL) flag

• Implements 4 *protection ring* levels
• CPL=3 → “user” mode

• CPL=0 → “supervisor/kernel” mode
  • Access to special instructions & hardware
How to modify CPL?

Q: Ok to allow user to directly modify CPL before invoking OS?

A: No! User can set CPL=0 and run arbitrary code before calling OS.
How to modify CPL?

Q: What about combining CPL “set” instruction with “jump” instruction to force instruction pointer (eip) change?

A: Nope! User can set CPL=0 and jump to user code to masquerade as OS.
How to modify CPL?

Q: What about combining CPL “set” instruction with “jump” instruction that must target OS codespace?

A: Close, but not good enough. User code may jump to “malicious” location in OS.
Solution: x86 provides `int` instruction:

- Sets CPL=0

- Loads a pre-defined OS entry point from interrupt descriptor table (IDT)

- IDT base address can only be set when CPL=0 (by privileged `lidt` instruction)
• Privileged instruction & hardware access prevented, but how is memory protected?
  
  • Each segment/page of memory in x86 is associated with a minimum CPL
  
  • Only permit current process to access its own segments/pages
• Finally, how can OS regain control from rebellious user process? (E.g., running in tight loop, never executing `int`)

• Hardware sends periodic *clock interrupt*

• *Preempts* user and summons OS
• *Isolation* accomplished.

• Related questions, how to achieve *hardware abstraction & concurrency*?
• Hardware abstraction
  • User traps to OS (via int) with service request
  • OS carries out task and returns result—“syscall”
• Therefore, hardware (e.g., NIC) is exposed as a software stack (e.g., TCP/IP)
• Concurrency

• Clock interrupt drives context switches and hardware multiplexing, carried out by OS scheduler (and others)

• Enables multitasking on limited hardware (compared to parallelism)
Different approaches to multitasking:

- *Cooperative*: process voluntarily control
- *Preemptive*: OS periodically interrupts
- *Real-time*: more stringent requirements
How is an OS organized?
That is, what are the *top-level modules* of an OS, and which must run in privileged mode (i.e., CPL=0)?
• Some modules:

  • Virtual memory
  • Scheduler
  • Device drivers
  • File system
  • IPC
Privileged modules constitute the “core” of the operating system—the OS kernel
• Traditional approach: all are privileged

• That is, entire “OS” runs in kernel mode
  • Known as *monolithic* kernel

• Pros/cons?
Alternative approach: *minimum* privileged

- That is, have a “*microkernel*” with minimal set of privileged services.
  - Everything else runs in user mode
    - Microkernel relays requests

- Pros/cons?
Monolithic Kernel based Operating System

Application

System Call

VFS

IPC, File System

Scheduler, Virtual Memory

Device Drivers, Dispatcher, ...

Hardware

Microkernel based Operating System

Application IPC

UNIX Server

Device Driver

File Server

Basic IPC, Virtual Memory, Scheduling

Hardware

user mode

kernel mode

 Courtesy of Wikimedia Commons
...suffice it to say that among the people who actually design operating systems, the debate is essentially over. *Microkernels have won.*

—Andrew Tanenbaum

(noted OS researcher)
The whole “microkernels are simpler” argument is just bull, and it is clearly shown to be bull by the fact that whenever you compare the speed of development of a microkernel and a traditional kernel, the traditional kernel wins. By a huge amount, too.

–Linus Torvalds
(chief architect, Linux)
• Your opinion?

• Assignment 1 (Paper)
• Yet another route: why not just implement the OS as a low-level library?

• Loss of isolation, but big efficiency gain (and flexibility in using hardware directly)

• Used by many embedded systems
• Finally, what about hosting multiple OS’s on a single machine?

• Why?

  • Useful on large, multi-core machines

• Hypervisors provide low-level virtual machines to guest OS’s

• Yet another layer of isolation!