Unix and Operating Systems
(Part I)

Devin J. Pohly <djpohly@cse.psu.edu>
Origins of Unix

• Developed in 1969 at Bell Labs
  ‣ Intended as a development environment for multi-platform code
• Use grew quickly
• Architectural advantages were embraced by academia and industry
  ‣ Dominates in “big iron” industrial environments
    • About 2/3 of servers run some Unix-like system (Dec. 2013)
Main attributes of Unix

• **multiuser** – supports multiple users on the same system at the same time, each working at their own terminal

• **multitasking** – supports running multiple programs at a time

• **portable** – when changing hardware, only the lowest software layers need to be reimplemented
Unix flavors over the years

• AT&T: the original Unix on research systems, 1969
  ‣ Ken Thompson, Dennis Ritchie, Brian Kernighan, Doug McIlroy, et al.
  ‣ System III, 1981
  ‣ System V (SysV), 1983; System V Release 4 (SVR4), 1988–95
• University of California, Berkeley: BSD series, 1977–1995
  ‣ Popular derivatives: FreeBSD, NetBSD, OpenBSD
• Microsoft: Xenix, 1980
  ‣ Abandoned (“vaporware”), ported to 386 by SCO in 1987, succeeded by SCO OpenServer
• Sun Microsystems: SunOS, 1982 (later Solaris, OpenSolaris, acquired by Oracle in 2010)
• Hewlett-Packard: HP-UX, 1984
• Digital Equipment Corp. (DEC)/Compaq/HP, Ultrix, 1984 (later Digital Unix, Tru64 Unix)
• IBM: AIX, 1986; z/OS (for mainframes), 2001
• Silicon Graphics Inc. (SGI): IRIX, 1988
• Open Software Foundation (DEC/IBM/HP/others): OSF/1, 1992 (based on Mach)
• Apple: Mac OS X, 2001 (derived from BSD and Mach)
Linux

- Linux kernel, 1991
  - Open-source Unix-like kernel that has seen broad adoption in both academia and industry
- GNU/Linux distributions
  - Red Hat, SUSE (Novell), Caldera (SCO, defunct), Debian, Mandrake/Mandriva, Slackware, Gentoo, **Ubuntu**, Knoppix, Fedora, Arch, and hundreds more
- Android, since 2003
  - Linux kernel
  - Open Handset Alliance, Android Open Source Project
Open source

- Many Unix-like systems in use today are distributed as "open source"
  - Open source software is distributed with a license where the copyright allows the user of the source to review, modify, and distribute with no cost to anyone.
  - Variants of this arrangement allow a person (a) to derive software from the distribution and charge or (b) never charge anyone for derivative works.
- Aside: “free” as in free beer (gratis) vs. free speech (libre)
Unix architecture

• Software layers
  ‣ *OS kernel* – direct interaction with hardware
  ‣ *system calls* – interface to the kernel
  ‣ *system libraries* – wrappers around system calls
  ‣ *programming language libraries* – extend system libraries
  ‣ *system utilities* – application-independent tools
    ▪ fsck, fdisk, ip, mknod, mount, nfsd
  ‣ *command interpreter* or *shell* – text-based user interface
  ‣ *application libraries* – application-specific tools
  ‣ *applications* – complete programs for ordinary users
    ▪ some applications have their own command shells and programming-
      language facilities (e.g., Perl, Python)
What’s an OS?

• Software that:
  ‣ Directly interacts with hardware
    ▪ OS is trusted to do so; user-level processes are not
    ▪ OS must be ported to new HW; user-level programs are portable
  ‣ Manages (allocates, schedules, protects) hardware resources
    ▪ decides which programs can access which files, memory locations, pixels on the screen, etc., and when
  ‣ Abstracts away messy hardware devices
    ▪ provides high-level, convenient, portable abstractions
    ▪ example: files vs. disk blocks

• Unix is a classic example of an OS
OS as abstraction provider

• The OS is the “layer below”
  ▸ a module that your program can call (with system calls)
  ▸ a powerful API
    ▸ Library functions in addition to system calls
    ▸ Unix API is standardized as POSIX, from “portable operating system” + (Un)ix
OS as abstraction provider

- API supports useful abstractions
  - filesystem: open, read, write, close
  - network stack: connect, listen, read, write
  - virtual memory: mmap, brk, shm_open
  - process management: fork, wait, nice
OS as protection system

- OS isolates processes from each other
  - permits controlled sharing via shared namespaces (e.g., filesystem names)
- OS isolates itself from processes
  - therefore must prevent processes from accessing hardware directly
- OS is allowed to access the hardware
  - user-level processes run with the CPU in an unprivileged mode
  - when the OS is running, the CPU is set to a privileged mode
  - user-level processes invoke a system call to safely enter the OS
Hardware privilege modes

- A **privilege mode** is a CPU state that restricts the kinds of actions that code may perform
  - e.g., prevents direct access to hardware, process controls, and privileged/sensitive instructions
- There are two modes we are principally concerned about in this class: **user** and **kernel** modes
  - **user mode** has low privilege and is used for normal programs (and some system services) running in “userspace”
  - **kernel mode** has full privilege and is used by the operating system in “kernelspace”
A CPU (thread of execution) is running userspace code in process A; that CPU is set to *user mode*.
Code in process A invokes a system call instruction or trap; the hardware then sets the CPU to *kernel mode* and gives control to the OS, which invokes the appropriate system call handler.
Since the CPU executing the OS code is in kernel mode, it is able to use privileged instructions that interact directly with hardware devices such as disks.
Once the OS has finished handling the system call (which can involve long waits as it interacts with hardware), it sets the CPU back to user mode and returns control from the system call handler to the userspace code in process A.
OS as protection system

The process continues executing whatever code is next after the system call invocation.
Device drivers

- A *device driver* is a software module (program) that implements the interface to a piece of hardware (often needs kernel mode privilege)
  - e.g., printers, monitors, graphics cards, USB devices
  - often provided by the manufacturer of the device
  - for performance reasons, the driver is commonly run in kernelspace
  - device drivers were often compiled into the kernel
    - required the administrator to re-compile the operating system whenever support for a new device type was needed
    - each system had its own custom kernel
Recompiling kernels?

- Recompilation of the kernel is problematic
  - takes a long time
  - requires sophistication
  - versioning problems

- Solution 1: userspace modules
  - Userspace programs that support the operating system
    - leverages protection
    - allows independent patching and upgrading
    - removes dependency on kernel version (mostly)
  - **Problem**: performance. For high-speed hardware, context switching is costly.
Recompiling kernels?

- **Solution 2: kernel modules**
  - aka loadable kernel modules (LKM)
  - software modules that run in kernel space that can be loaded/unloaded on a running system
    - can extend the kernel functionality without recompilation
    - kernel provides generic interfaces (APIs) that the module uses to integrate with kernel structures
    - used by almost every modern OS (OSX, Windows, Linux, etc.)
  - PROTIP: To see what modules are loaded on your Unix system, use the “`lsmod`” command.