OS09
K. Gopinath, IISc

Slides are from
Tanenbaum, provided as part of his book
Updated Tanenbaum slides from
by Darrell Long/Ethan Miller at UCSC
Mine own (K. Gopinath)
Some papers/books too numerous to list
Hence PL. DO NOT CIRCULATE
Overview: Chapter 1

- What is an operating system, anyway?
- Operating systems history
- The zoo of modern operating systems
- Review of computer hardware
- Operating system concepts
- Operating system structure
  - User interface to the operating system
  - Anatomy of a system call
What *is* an operating system?

* It’s a program that runs on the “raw” hardware
  - Acts as an intermediary between computer and users
  - Standardizes the interface to the user across different types of hardware: extended machine
    - Hides the messy details which must be performed
    - Presents user with a virtual machine, easier to use

* It’s a resource manager
  - Each program gets time with the resource
  - Each program gets space on the resource

* May have potentially conflicting goals:
  - Use hardware efficiently
  - Give maximum performance to each user
A computer system consists of
- hardware
- system programs
- application programs

<table>
<thead>
<tr>
<th></th>
<th>Application programs</th>
<th>System programs</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banking system</td>
<td>Airline reservation</td>
<td>Command interpreter</td>
<td>Operating system</td>
</tr>
<tr>
<td>Compilers</td>
<td>Editors</td>
<td></td>
<td>Machine language</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Microarchitecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Physical devices</td>
</tr>
</tbody>
</table>
Operating system timeline

First generation: 1945 – 1955
• Vacuum tubes
• Plug boards

Second generation: 1955 – 1965
• Transistors
• Batch systems

Third generation: 1965 – 1980
• Integrated circuits
• Multiprogramming

Fourth generation: 1980 – present
• Large scale integration
• Personal computers

Fifth generation: ??? (maybe 2001–?)
• Systems connected by high-speed networks?
• Wide area resource management?
• Peer-to-peer systems?
First generation: direct input

- Run one job at a time
  - Enter it into the computer (might require rewiring!)
  - Run it
  - Record the results
- Problem: lots of wasted computer time!
  - Computer was idle during first and last steps
  - Computers were very expensive!
- Goal: make better use of an expensive commodity: computer time
Second generation: batch systems

- Bring cards to 1401
- Read cards onto input tape
- Put input tape on 7094
- Perform the computation, writing results to output tape
- Put output tape on 1401, which prints output
Structure of a typical 2nd generation job

- **Data for the program**
- **FORTRAN program code**
- **Job header**

```plaintext
$JOB, 10, 6610802, ETHAN MILLER
$FORTRAN
$LOAD
$RUN
$END
```
Spooling

- Original batch systems used tape drives
- Later batch systems used disks for buffering
  - Operator read cards onto disk attached to the computer
  - Computer read jobs from disk
  - Computer wrote job results to disk
  - Operator directed that job results be printed from disk
- Disks enabled simultaneous peripheral operation on-line (spooling)
  - Computer overlapped I/O of one job with execution of another
  - Better utilization of the expensive CPU
  - Still only one job active at any given time
Third generation: multiprogramming

- Multiple jobs in memory
  - Protected from one another
- Operating system protected from each job as well
- Resources (time, hardware) split between jobs
- Still not interactive
  - User submits job
  - Computer runs it
  - User gets results minutes (hours, days) later
Timesharing

- Multiprogramming allowed several jobs to be active at one time
  - Initially used for batch systems
  - Cheaper hardware terminals ⇒ interactive use

- Computer use got much cheaper and easier
  - No more “priesthood”
  - Quick turnaround meant quick fixes for problems
Types of modern operating systems

- Mainframe operating systems: MVS
- Server operating systems: FreeBSD, Solaris, Linux
- Multiprocessor operating systems: Cellular IRIX
- Personal computer operating systems: MacOS X, Windows Vista, Linux
- Real-time operating systems: VxWorks
- Embedded operating systems
- Smart card operating systems

Some operating systems can fit into more than one category
Components of a simple PC

- Video controller
- Hard drive controller
- USB controller
- Network controller

Outside world

Computer internals (inside the “box”)

CPU

Memory
Computer Hardware

Structure of a large PC
CPU internals

Pipelined CPU

Superscalar CPU
Goal: really large memory with very low latency
• Latencies are smaller at the top of the hierarchy
• Capacities are larger at the bottom of the hierarchy
Solution: move data between levels to create illusion of large memory with low latency
Disk drive structure

- Data stored on surfaces
  - Up to two surfaces per platter
  - One or more platters per disk
- Data in concentric tracks
  - Tracks broken into sectors
    - 256B–1KB per sector
  - Cylinder: corresponding tracks on all surfaces
- Data read and written by heads
  - Actuator moves heads
  - Heads move in unison
Memory

- Single base/limit pair: set for each process
- Two base/limit registers: one for program, one for data
Anatomy of a device request

Left: sequence as seen by hardware
- Request sent to controller, then to disk
- Disk responds, signals disk controller which tells interrupt controller
- Interrupt controller notifies CPU

Right: interrupt handling (software point of view)
1: Interrupt
2: Process interrupt
3: Return
Operating systems concepts

- Many of these should be familiar to Unix users…
  - Processes (and trees of processes)
  - Deadlock
  - File systems & directory trees
  - Pipes
  - We'll cover all of these in more depth later on, but it's useful to have some basic definitions now
Processes

- **Process**: program in execution
  - Address space (memory) the program can use
  - State (registers, including program counter & stack pointer)

- OS keeps track of all processes in a process table

- Processes can create other processes
  - Process tree tracks these relationships
  - A is the root of the tree
  - A created three child processes: B, C, and D
  - C created two child processes: E and F
  - D created one child process: G
Inside a (Unix) process

Processes have three segments
- Text: program code
- Data: program data
  - Statically declared variables
  - Areas allocated by malloc() or new
- Stack
  - Automatic variables
  - Procedure call information

Address space growth
- Text: doesn't grow
- Data: grows “up”
- Stack: grows “down”
Deadlock

Potential deadlock

Actual
Hierarchical file systems
Interprocess communication

- Processes want to exchange information with each other
- Many ways to do this, including
  - Network
  - Pipe (special file): A writes into pipe, and B reads from it
System calls

- OS runs in privileged mode
  - Some operations are permitted only in privileged (also called supervisor or system) mode
    - Example: access a device like a disk or network card
    - Example: change allocation of memory to processes
  - User programs run in user mode and can’t do the operations

- Programs want the OS to perform a service
  - Access a file
  - Create a process
  - Others…

- Accomplished by system call
How system calls work

- User program enters supervisor mode
  - Must enter via well-defined entry point
- Program passes relevant information to OS
- OS performs the service if
  - The OS is able to do so
  - The service is permitted for this program at this time
- OS checks information passed to make sure it's OK
  - Don't want programs reading data into other programs’ memory!
- OS needs to be paranoid!
  - Users do the darnedest things…
Making a system call

- System call: `read(fd, buffer, length)`
- Program pushes arguments, calls library
- Library sets up trap, calls OS
- OS handles system call
- Control returns to library
- Library returns to user program
# System calls for files & directories

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd = open(name, how)</code></td>
<td>Open a file for reading and/or writing</td>
</tr>
<tr>
<td><code>s = close(fd)</code></td>
<td>Close an open file</td>
</tr>
<tr>
<td><code>n = read(fd, buffer, size)</code></td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td><code>n = write(fd, buffer, size)</code></td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td><code>s = lseek(fd, offset, whence)</code></td>
<td>Move the “current” pointer for a file</td>
</tr>
<tr>
<td><code>s = stat(name, &amp;buffer)</code></td>
<td>Get a file’s status information (in <code>buffer</code>)</td>
</tr>
<tr>
<td><code>s = mkdir(name, mode)</code></td>
<td>Create a new directory</td>
</tr>
<tr>
<td><code>s = rmdir(name)</code></td>
<td>Remove a directory (must be empty)</td>
</tr>
<tr>
<td><code>s = link(name1, name2)</code></td>
<td>Create a new entry (<code>name2</code>) that points to the same object as <code>name1</code></td>
</tr>
<tr>
<td><code>s = unlink(name)</code></td>
<td>Remove <code>name</code> as a link to an object (deletes the object if <code>name</code> was the only link to it)</td>
</tr>
</tbody>
</table>
## More system calls

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid = fork()</td>
<td>Create a child process identical to the parent</td>
</tr>
<tr>
<td>pid=waitpid(pid,&amp;statloc,options)</td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td>s = execve(name,argv,environp)</td>
<td>Replace a process’ core image</td>
</tr>
<tr>
<td>exit(status)</td>
<td>Terminate process execution and return status</td>
</tr>
<tr>
<td>s = chdir(dirname)</td>
<td>Change the working directory</td>
</tr>
<tr>
<td>s = chmod(name,mode)</td>
<td>Change a file’s protection bits</td>
</tr>
<tr>
<td>s = kill(pid,signal)</td>
<td>Send a signal to a process</td>
</tr>
<tr>
<td>seconds = time(&amp;seconds)</td>
<td>Get the current time</td>
</tr>
</tbody>
</table>
A simple shell

while (TRUE) {
    /* repeat forever */
    print_prompt( ); /* display prompt */
    read_command (command, parameters) /* input from terminal */

    if (fork() != 0) { /* fork off child process */
        /* Parent code */
        waitpid(-1, &status, 0); /* wait for child to exit */
    } else {
        /* Child code */
        execve (command, parameters, 0); /* execute command */
    }
}
Operating system structure

- OS is composed of lots of pieces
  - Memory management
  - Process management
  - Device drivers
  - File system

- How do the pieces of the operating system fit together and communicate with each other?

- Different ways to structure an operating system
  - Monolithic
    - Modular is similar, but more extensible
  - Virtual machines
  - Microkernel
Monolithic OS structure

- All of the OS is one big “program”
  - Any piece can access any other piece
- Sometimes modular (as with Linux)
  - Extra pieces can be dynamically added
  - Extra pieces become part of the whole
- Easy to write, but harder to get right…
Virtual machines

- First widely used in VM/370 with CMS
- Available today in VMware (and Qemu, sort of)
  - Allows users to run any x86-based OS on top of Linux or NT
- “Guest” OS can crash without harming underlying OS
  - Only virtual machine fails—rest of underlying OS is fine
- “Guest” OS can even use raw hardware
  - Virtual machine keeps things separated
Microkernels (client-server)

- Processes (clients and OS servers) don't share memory
  - Communication via message-passing
  - Separation reduces risk of "byzantine" failures
- Examples include
  - Mach (used by MacOS X)
  - Minix
History

• UNIX: 1969 Thompson & Ritchie AT&T Bell Labs
• BSD: 1978 Berkeley Software Distribution
• Commercial Vendors: Sun, HP, IBM, SGI, DEC
• GNU: 1984 Richard Stallman, FSF
• POSIX: 1986 IEEE Portable Operating System UNIX
• Minix: 1987 Andy Tannenbaum
• SVR4: 1989 AT&T and Sun
• Linux: 1991 Linus Torvalds Intel 386 (i386)
• Open Source: GPL, LGPL, Cathedral and the Bazaar
GNU/Linux Features

• “UNIX-like”: Multi-user, multi-tasking, UNIX system

• Goals
  – Speed, efficiency
  – “aims at” standards compliance (e.g., POSIX)

• “all the features you would expect in a modern UNIX”
  – preemptive multitasking
  – virtual memory (protected memory, paging)
  – shared libraries
  – demand loading, dynamic kernel modules
  – shared copy-on-write executables
  – TCP/IP networking

• other features:
  – SMP support, large memory, large files
  – advanced networking, advanced filesystems
  – efficient, stable, highly portable, supports most device hardware
  – active development community, support, documentation, open source
  – GUIs, applications

• Components: kernel (VM, proc mgmt), libraries (syscalls, buf I/O), system utils (netw daemons)
What’s a Kernel?

(Also: executive, system monitor, nucleus)

• controls and mediates access to hardware
• implements and supports fundamental abstractions
  • processes, files, devices, users, net, etc.
• schedules “fair” sharing of system resources
  • memory, cpu, disk, descriptors, etc.
• enforces security and protection
• responds to user requests for service (system calls)
• performs routine maintenance, system checks, etc.

Highly concurrent!
Non-stop if possible! Even modifications on a live kernel
Highly extensible!
Needs to be designed to survive for a few decades (1 or 2) at least!
Software arch/engg critical to success
Kernel Design Goals

• performance: efficiency, speed
  – utilize resources to capacity, low overhead, code size
• stability: robustness, resilience
  – uptime, graceful degradation
• capability: features, flexibility, compatibility
• security, protection
  – protect users from each other, secure system from bad guys
• portability
• clarity
• extensibility
Design Tradeoffs

• Butler Lampson: ‘choose any three design goals”
• efficiency vs. protection
  – more checks, more overhead
• clarity vs. compatibility
  – ugly implementation of ‘broken”standards (e.g. signals)
• flexibility vs. security
  – the more you can do, the more potential security holes!
• not all are antagonistic
  – portability tends to enhance code clarity
Dependency Diagrams

Conceptual

Concrete
Vahalia’s Diagram

from *Unix Internals: The New Frontiers*
Uresh Vahalia / Prentice-Hall 1996
“Core” Kernel

Applications

System Libraries (libc)

System Call Interface

I/O Related
- File Systems
- Networking
- Device Drivers

Process Related
- Scheduler
- Memory Management
- IPC

Architecture-Dependent Code

Hardware
Inserting a wireless card

- i82365: PCMCIA controller driver