Computer Systems
CSE 410 Spring 2012
20 – OS Introduction & Structure

Slides adapted from CSE 451 material by Gribble, Lazowska, Levy, and Zahorjan
What is an Operating System?

**Answers:**

- I don't know
- Nobody knows
- The book claims to know – read Chapter 1
- They’re programs – big hairy programs
  - The Linux source has over 1.7M lines of C
What is an Operating System?

Answers:

- I don't know
- Nobody knows
- The book claims to know – read Chapter 1
- They’re programs – big hairy programs
  - The Linux source has over 1.7M lines of C

Okay. What are some goals of an OS?
The traditional picture

“The OS is everything you don’t need to write in order to run your application”

This depiction invites you to think of the OS as a library

- In some ways, it is:
  - all operations on I/O devices require OS calls (syscalls)
- In other ways, it isn't:
  - you use the CPU/memory without OS calls
  - it intervenes without having been explicitly called
The OS and hardware

- An OS **mediates** programs’ access to hardware resources (*sharing and protection*)
  - computation (CPU)
  - volatile storage (memory) and persistent storage (disk, etc.)
  - network communications (TCP/IP stacks, Ethernet cards, etc.)
  - input/output devices (keyboard, display, sound card, etc.)

- The OS **abstracts** hardware into logical resources and well-defined interfaces to those resources (*ease of use*)
  - processes (CPU, memory)
  - files (disk)
  - programs (sequences of instructions)
  - sockets (network)
Why bother with an OS?

- **Application benefits**
  - programming *simplicity*
    - see high-level abstractions (files) instead of low-level hardware details (device registers)
    - abstractions are *reusable* across many programs
  - *portability* (across machine configurations or architectures)
    - device independence: 3com card or Intel card?

- **User benefits**
  - safety
    - program “sees” its own virtual machine, thinks it “owns” the computer
    - OS *protects* programs from each other
    - OS fairly *multiplexes* resources across programs
  - efficiency (cost and speed)
    - share one computer across many users
    - concurrent execution of multiple programs
The major OS issues

- structure: how is the OS organized?
- sharing: how are resources shared across users?
- naming: how are resources named (by users or programs)?
- security: how is the integrity of the OS and its resources ensured?
- protection: how is one user/program protected from another?
- performance: how do we make it all go fast?
- reliability: what happens if something goes wrong (either with hardware or with a program)?
- extensibility: can we add new features?
- communication: how do programs exchange information, including across a network?
More OS issues...

- concurrency: how are parallel activities (computation and I/O) created and controlled?
- scale: what happens as demands or resources increase?
- persistence: how do you make data last longer than program executions?
- distribution: how do multiple computers interact with each other?
- accounting: how do we keep track of resource usage, and perhaps charge for it?

There are tradeoffs, not right and wrong!
Architectural features affecting OS’s

- These features were built primarily to support OS’s:
  - timer (clock) operation
  - synchronization instructions (e.g., atomic test-and-set)
  - memory protection
  - I/O control operations
  - interrupts and exceptions
  - protected modes of execution (kernel vs. user)
  - privileged instructions
  - system calls (and software interrupts)
  - virtualization architectures
Privileged instructions

- some instructions are restricted to the OS
  - known as privileged instructions
- e.g., only the OS can:
  - directly access I/O devices (disks, network cards)
    - why?
  - manipulate memory state management
    - page table pointers, TLB loads, etc.
    - why?
  - manipulate special ‘mode bits’
    - interrupt priority level
    - why?
OS protection

- So how does the processor know if a privileged instruction should be executed?
  - the architecture must support at least two modes of operation: kernel mode and user mode
    - VAX, x86 support 4 protection modes
  - mode is set by status bit in a protected processor register
    - user programs execute in user mode
    - OS executes in kernel (privileged) mode (OS == kernel)

- Privileged instructions can only be executed in kernel (privileged) mode
  - what happens if code running in user mode attempts to execute a privileged instruction?
Crossing protection boundaries

- So how do user programs do something privileged?
  - e.g., how can you write to a disk if you can’t execute an I/O instructions?

- User programs must call an OS procedure – that is, get the OS to do it for them
  - OS defines a set of system calls
  - User-mode program executes system call instruction (int on x86)

- Syscall (int) instruction
  - Like a protected procedure call
  - We’ve seen this earlier, but a few more details...
System calls

- The syscall instruction atomically:
  - Saves the current PC
  - Sets the execution mode to privileged
  - Sets the PC to a handler address

- With that, it’s a lot like a local procedure call:
  - Caller puts arguments in a place callee expects (registers or stack)
    - One of the args is a syscall number, indicating which OS function to invoke
  - Callee (OS) saves caller’s state (registers, other control state) so it can use the CPU
  - OS function code runs
    - **OS must verify caller’s arguments** (e.g., pointers)
  - OS returns using a special instruction
    - Automatically sets PC to return address and sets execution mode to user
Firefox: `read(int fileDescriptor, void *buffer, int numBytes)`

**user mode**
- Save user PC
- PC = trap handler address
- Enter kernel mode

**kernel mode**
- **trap handler**
  - Save app state
  - Verify syscall number
  - Find sys_read() handler in vector table
- **sys_read() kernel routine**
  - Verify args
  - Initiate read
  - Choose next process to run
  - Setup return values
  - Restore app state

**ERET instruction**
- PC = saved PC
- Enter user mode

Diagram illustrates the process of transitioning from user mode to kernel mode and back, highlighting key steps in the system call handling process.
OS structure

- The OS sits between application programs and the hardware
  - it mediates access and abstracts away ugliness
  - programs request services via traps or exceptions
  - devices request attention via interrupts
The Classic Diagram...

User Apps
- Firefox
- Photoshop
- Acrobat
- Java

Operating System
- Application Interface (API)
  - File Systems
  - Memory Manager
  - Process Manager
  - Network Support
  - Device Drivers
  - Interrupt Handlers
  - Boot & Init

Hardware Abstraction Layer

Hardware (CPU, devices)
But reality isn’t always that simple…
Major OS components

- processes
- memory
- I/O
- secondary storage
- file systems
- protection
- shells (command interpreter, or OS UI)
- GUI
- networking
Process management

- An OS executes many kinds of activities:
  - users’ programs
  - batch jobs or scripts
  - system programs
    - print spoolers, name servers, file servers, network daemons, ...

- Each of these activities is encapsulated in a process
  - a process includes the execution context
    - PC, registers, VM, OS resources (e.g., open files), etc...
    - plus the program itself (code and data)
  - the OS’s process module manages these processes
    - creation, destruction, scheduling, ...
Program/processor/process

- Note that a program is totally passive
  - just bytes on a disk that encode instructions to be run

- A process is an instance of a program being executed by a (real or virtual) processor
  - at any instant, there may be many processes running copies of the same program (e.g., an editor); each process is separate and (usually) independent
  - Linux: `ps -auwwx` to list all processes
States of a user process

- Running
  - Dispatch
  - Interrupt

- Ready
  - Interrupt

- Blocked
  - Trap or exception
Process operations

The OS provides the following kinds operations on processes (i.e., the process abstraction interface):

- create a process
- delete a process
- suspend a process
- resume a process
- clone a process
- inter-process communication
- inter-process synchronization
- create/delete a child process (subprocess)
Memory management

- The primary memory is the directly accessed storage for the CPU
  - programs must be stored in memory to execute
  - memory access is fast
  - but memory doesn’t survive power failures

- OS must:
  - allocate memory space for programs (explicitly and implicitly)
  - deallocate space when needed by rest of system
  - maintain mappings from physical to virtual memory
    - through page tables
  - decide how much memory to allocate to each process
    - a policy decision
  - decide when to remove a process from memory
    - also policy
I/O

- A big chunk of the OS kernel deals with I/O
  - hundreds of thousands of lines in NT (Windows)
- The OS provides a standard interface between programs (user or system) and devices
  - file system (disk), sockets (network), frame buffer (video)
- **Device drivers** are the routines that interact with specific device types
  - **encapsulates** device-specific knowledge
    - e.g., how to initialize a device, how to request I/O, how to handle interrupts or errors
    - examples: SCSI device drivers, Ethernet card drivers, video card drivers, sound card drivers, ...
- **Note:** Windows has ~35,000 device drivers!
Secondary storage

- Secondary storage (disk, tape) is persistent memory
  - often magnetic media, survives power failures (hopefully)

- Routines that interact with disks are typically at a very low level in the OS
  - used by many components (file system, VM, ...)
  - handle scheduling of disk operations, head movement, error handling, and often management of space on disks

- Usually independent of file system
  - although there may be cooperation
  - file system knowledge of device details can help optimize performance
    - e.g., place related files close together on disk
File systems

- Secondary storage devices are crude and awkward
  - e.g., “write 4096 byte block to sector 12”

- File system: a convenient abstraction
  - defines logical objects like files and directories
    - hides details about where on disk files live
  - as well as operations on objects like read and write
    - read/write byte ranges instead of blocks

- A file is the basic unit of long-term storage
  - file = named collection of persistent information

- A directory is just a special kind of file
  - directory = named file that contains names of other files and metadata about those files (e.g., file size)

- Note: Sequential byte stream is only one possibility!
File system operations

- The file system interface defines standard operations:
  - file (or directory) creation and deletion
  - manipulation of files and directories (read, write, extend, rename, protect)
  - copy
  - lock

- File systems also provide higher level services
  - accounting and quotas
  - backup (must be incremental and online!)
  - (sometimes) indexing or search
  - (sometimes) file versioning
Protection

- Protection is a general mechanism used throughout the OS
  - all resources needed to be protected
    - memory
    - processes
    - files
    - devices
    - CPU time
    - ...
  - protection mechanisms help to detect and contain unintentional errors, as well as preventing malicious destruction
Command interpreter (shell)

- A particular program that handles the interpretation of users’ commands and helps to manage processes
  - user input may be from keyboard (command-line interface), from script files, or from the mouse (GUIs)
  - allows users to launch and control new programs

- On some systems, command interpreter may be a standard part of the OS (mostly old/historical or tiny systems)

- On others, it’s just non-privileged code that provides an interface to the user
  - e.g., bash/csh/tcsh/zsh on UNIX

- On others, there may be no command language
  - e.g., classic MacOS (pre-OS X)
OS structure

- It’s not always clear how to stitch OS modules together:
OS structure

- An OS consists of all of these components, plus:
  - many other components
  - system programs (privileged and non-privileged)
    - e.g., bootstrap code, the init program, ...

- Major issue:
  - how do we organize all this?
  - what are all of the code modules, and where do they exist?
  - how do they cooperate?

- Massive software engineering and design problem
  - design a large, complex program that:
    - performs well, is reliable, is extensible, is backwards compatible, ...
  - we won’t be able to go into detail in the remaining few classes (alas...)
Early structure: Monolithic

Traditionally, OS’s (like UNIX) were built as a monolithic entity:

- **OS**
  - user programs
  - everything
  - hardware
Monolithic design

■ **Major advantage:**
  - cost of module interactions is low (procedure call)

■ **Disadvantages:**
  - hard to understand
  - hard to modify
  - unreliable (no isolation between system modules)
  - hard to maintain

■ **What is the alternative?**
  - find a way to organize the OS in order to simplify its design and implementation
Layering

- The traditional approach is layering
  - implement OS as a set of layers
  - each layer presents an enhanced ‘virtual machine’ to the layer above
- The first description of this approach was Dijkstra’s THE system
  - Layer 5: Job Managers
    - Execute users’ programs
  - Layer 4: Device Managers
    - Handle devices and provide buffering
  - Layer 3: Console Manager
    - Implements virtual consoles
  - Layer 2: Page Manager
    - Implements virtual memories for each process
  - Layer 1: Kernel
    - Implements a virtual processor for each process
  - Layer 0: Hardware
- Each layer can be tested and verified independently
Problems with layering

- **Imposes hierarchical structure**
  - but real systems are more complex:
    - file system requires VM services (buffers)
    - VM would like to use files for its backing store
  - strict layering isn’t flexible enough

- **Poor performance**
  - each layer crossing has overhead associated with it

- **Disjunction between model and reality**
  - systems modeled as layers, but not really built that way
Hardware Abstraction Layer

- An example of layering in modern operating systems
- Goal: separates hardware-specific routines from the “core” OS
  - Provides portability
  - Improves readability
Microkernels

- Popular in the late 80’s, early 90’s
  - recent resurgence of popularity

- Goal:
  - minimize what goes in kernel
  - organize rest of OS as user-level processes

- This results in:
  - better reliability (isolation between components)
  - ease of extension and customization
  - poor performance (user/kernel boundary crossings)

- First microkernel system was Hydra (CMU, 1970)
  - Follow-ons: Mach (CMU), Chorus (French UNIX-like OS), OS X (Apple), in some ways NT (Microsoft)
Microkernel structure illustrated

User processes:
- firefox
- powerpoint
- apache
- word
- itunes

System processes:
- file system
- network
- threads
- scheduling
- paging

Microkernel:
- communication
- protection
- processor control
- low-level VM

Hardware

User mode

Kernel mode
EXAMPLE: WINDOWS

Application program

User mode

Word
Excel
Photoshop

Kernel mode

Windows—including scheduling, memory management, process management, file system, device drivers (I/O) and much, much more
Summary & Next

• **Summary**
  – OS design has been an evolutionary process of trial and error. Probably more error than success
  – Successful OS designs have run the spectrum from monolithic, to layered, to micro kernels, to virtual machine monitors
  – The role and design of an OS are still evolving
  – It is impossible to pick one “correct” way to structure an OS

• **Next...**
  – Processes and threads, one of the most fundamental pieces in an OS
  – What these are, what do they do, and how do they do it