Intro to Operating Systems

CSE 410, Spring 2009
Computer Systems

http://www.cs.washington.edu/410
Readings and References

• Reading
  » Operating System Concepts, Silberschatz, Galvin, and Gagne
    • Ch. 1 Introduction & Ch. 2 OS Structures for background
    • Most useful for us: Sec. 1.1, 1.4-1.9, 2.1, 2.3-2.4, 2.6-2.7

  » Slide credits: largely taken from CSE451, courtesy of Hank Levy.
What is an Operating System?

• An operating system (OS) is:
  » a software layer to abstract away and manage details of hardware resources
  » a set of utilities to simplify application development
  » “all the code you didn’t write” in order to implement your application

• Key idea: virtualization of resources
The OS and hardware

• An OS mediates programs’ access to hardware resources
  » Computation (CPU)
  » Volatile storage (memory) and persistent storage (disk, etc.)
  » Network communications (TCP/IP stacks, ethernet cards, etc.)
  » Input/output devices (keyboard, mouse, display, sound card, ..)

• The OS abstracts hardware into logical resources and well-defined interfaces to those resources
  » 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” own virtual machine, thinks it owns computer
    - OS protects programs from each other (what if one crashes?)
    - 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 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 and growth**: 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? how do we make distribution invisible?
- **accounting**: how do we keep track of resource usage, and perhaps charge for it?

There are a huge number of engineering tradeoffs in dealing with these issues!
Hardware/Software Changes with Time

- 1960s: mainframe computers (IBM)
- 1970s: minicomputers (DEC)
- 1980s: microprocessors and workstations (SUN)
- 1990s: PCs (rise of Microsoft, Intel, then Dell)
- 2000: Internet Services / Clusters (Amazon)
- 2006: General Cloud Computing (Google, Amazon)
- ......
- 2020: it’s up to you!!
OS history

• In the very beginning…
  » OS was just a library of code that you linked into your program; programs were loaded in their entirety into memory, and executed
  » interfaces were literally switches and blinking lights

• And then came batch systems
  » OS was stored in a portion of primary memory
  » OS loaded the next job into memory from the card reader
    • job gets executed
    • output is printed, including a dump of memory (why?)
    • repeat…
  » card readers and line printers were very slow
    • so CPU was idle much of the time (wastes $$)
Spooling

- Disks were much faster than card readers and printers
- Spool (Simultaneous Peripheral Operations On-Line)
  » while one job is executing, spool next job from card reader onto disk
    • slow card reader I/O is overlapped with CPU
  » can even spool multiple programs onto disk
    • OS must choose which to run next
    • job scheduling
  » but, CPU still idle when a program interacts with a peripheral during execution
  » buffering, double-buffering
Multiprogramming

- To increase system utilization, **multiprogramming** OSs were invented
  - keeps multiple runnable jobs loaded in memory at once
  - overlaps I/O of a job with computing of another
    - while one job waits for I/O completion, OS runs instructions from another job
  - to benefit, need **asynchronous** I/O devices
    - need some way to know when devices are done
      - interrupts
      - polling
  - goal: optimize system throughput
    - perhaps at the cost of response time…
Timesharing

• To support interactive use, create a timesharing OS:
  » multiple terminals into one machine
  » each user has illusion of entire machine to him/herself
  » optimize response time, perhaps at the cost of throughput

• Timeslicing
  » divide CPU equally among the users
  » if job is truly interactive (e.g. editor), then can jump between programs and users faster than users can generate load
  » permits users to interactively view, edit, debug running programs (why does this matter?)

• MIT Multics system (mid-1960’s) was the first large timeshared system
  » nearly all OS concepts can be traced back to Multics
Timesharing

• In early 1980s, a single timeshared VAX/780 (like the one in the Allen Center atrium) ran computing for the entire CSE department.

• A typical VAX/780 was 1 MIPS (1 MHz) and had 16MB of RAM and 100MB of disk.

• An iPhone is 400 MIPS, has 128MB of RAM (way too little though) and 8GB of disk.
Parallel systems

• Some applications can be written as multiple parallel threads or processes
  » can speed up the execution by running multiple threads/processes simultaneously on multiple CPUs [Burroughs D825, 1962]
    • true multiprocesssing (not just multiprogramming)
  » need OS and language primitives for dividing program into multiple parallel activities
  » need OS primitives for fast communication among activities
    • degree of speedup dictated by communication/computation ratio
  » many flavors of parallel computers today
    • SMPs (symmetric multi-processors, multi-core)
    • SMT (simultaneous multithreading [“hyperthreading”])
    • MPPs (massively parallel processors)
    • NOWs (networks of workstations) [clusters]
    • computational grid (SETI @home)
Personal computing

- Primary goal was to enable new kinds of interactive applications
- Bit-mapped display [Xerox Alto, 1973]
  - New graphic/visual apps
  - New input device (the mouse)
- Move computing near the display
  - Why?
- Window systems
  - The display as a managed resource
- Local area networks [Ethernet]
  - Why?
- Effect on OS?
Embedded OS

• Pervasive computing
  » cheap processors embedded everywhere
  » how many are on your body now? in your car?
  » cell phones, PDAs, games, iPod, network computers, …

• Typically very constrained hardware resources
  » slow processors
  » small amount of memory
  » no disk or tiny disk
  » typically only one dedicated application
  » limited power

• But technology changes fast
  » embedded CPUs are getting faster
  » storage is growing rapidly
OS structure

• The OS sits between application programs and the hardware
  » it mediates access and abstracts away ugliness
  » programs request services via exceptions (traps or faults)
  » devices request attention via interrupts
Major OS components

- processes
- memory
- I/O
- secondary storage
- file systems
- protection
- accounting
- shells (command interpreter, or OS UI)
- GUI
- networking
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, …
Early structure: Monolithic

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

```
user programs

OS

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
Lower-level architecture and the OS

• Operating system functionality is dictated, at least in part, by the underlying hardware architecture
  » includes instruction set (synchronization, I/O, …)
  » also hardware components like MMU or DMA controllers

• Architectural support can vastly simplify (or complicate!) OS tasks
  » e.g.: early PC operating systems (DOS, MacOS) lacked support for virtual memory, in part because at that time PCs lacked necessary hardware support
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)
  - protected instructions
  - system calls (and software interrupts)
Protected instructions

• some instructions are restricted to the OS
  » known as protected or 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, user/kernel mode bit
    • why?
  » halt instruction
    • why?
OS protection

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

• Protected instructions can only be executed in the kernel mode
  » what happens if user mode executes a protected 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 do I/O instructions?

- User programs must call an OS procedure
  - OS defines a sequence of system calls
  - how does the user-mode to kernel-mode transition happen?

- There must be a system call instruction, which:
  - causes an exception (throws a software interrupt), which vectors to a kernel handler
  - passes a parameter indicating which system call to invoke
  - saves caller’s state (regs, mode bit) so they can be restored
  - OS must verify caller’s parameters (e.g., pointers)
  - must be a way to return to user mode once done
A kernel crossing illustrated

Firefox: read()

trap to kernel mode; save app state

trap handler

find read() handler in vector table

read() kernel routine

restore app state, return to user mode, resume
System call issues

- What would happen if kernel didn’t save state?
- Why must the kernel verify arguments?
- How can you reference kernel objects as arguments or results to/from system calls?
OS control flow

- after the OS has booted, all entry to the kernel happens as the result of an event
  - event immediately stops current execution
  - changes mode to kernel mode, event handler is called
- kernel defines handlers for each event type
  - specific types are defined by the architecture
    - e.g.: timer event, I/O interrupt, system call trap
  - when the processor receives an event of a given type, it
    - transfers control to handler within the OS
    - handler saves program state (PC, regs, etc.)
    - handler functionality is invoked
    - handler restores program state, returns to program
Interrupts and exceptions

• Two main types of events: interrupts and exceptions
  » exceptions are caused by software executing instructions
    • e.g., the x86 ‘int’ instruction, MIPS ‘syscall’ instruction
    • e.g., a page fault, write to a read-only page, divide by 0
    • an expected exception is a “trap”, unexpected is a “fault”
  » interrupts are caused by hardware devices
    • e.g., device finishes I/O
    • e.g., timer fires
I/O control

• Issues:
  » how does the kernel start an I/O?
    • special I/O instructions
    • memory-mapped I/O
  » how does the kernel notice an I/O has finished?
    • polling
    • interrupts
• Interrupts are basis for asynchronous I/O
  » device performs an operation asynch to CPU
  » device sends an interrupt signal on bus when done
  » in memory, a vector table contains list of addresses of kernel routines to handle various interrupt types
  » CPU switches to address indicated by vector specified by interrupt signal
Timers

• How can the OS prevent runaway user programs from hogging the CPU (infinite loops?)
  » use a hardware timer that generates a periodic interrupt
  » before it transfers to a user program, the OS loads the timer with a time to interrupt
    • “quantum”: how big should it be set?
  » when timer fires, an interrupt transfers control back to OS
    • at which point OS must decide which program to schedule next
    • very interesting policy question: we’ll dedicate a class to it

• Should the timer be privileged?
  » for reading or for writing?
Synchronization

• Interrupts cause a wrinkle:
  » may occur any time, causing code to execute that interferes with code that was interrupted
  » OS must be able to synchronize concurrent processes

• Synchronization:
  » guarantee that short instruction sequences (e.g., read-modify-write) execute atomically
  » one method: turn off interrupts before the sequence, execute it, then re-enable interrupts
    • architecture must support disabling interrupts
  » another method: have special complex atomic instructions
    • read-modify-write
    • test-and-set
    • load-linked store-conditional
“Concurrent programming”

- Management of concurrency and asynchronous events is biggest difference between “systems programming” and “traditional application programming”
  - modern “event-oriented” application programming is a middle ground
- Arises from the architecture
- Can be sugar-coated, but cannot be totally abstracted away
- Huge intellectual challenge
  - Unlike vulnerabilities due to buffer overruns, which are just sloppy programming
Architectures are still evolving

• New features are still being introduced to meet modern demands, e.g.:
  » Support for virtual machine monitors
  » Hardware transaction support (to simplify parallel programming)
  » Support for security (encryption, trusted modes)
  » Increasingly sophisticated video / graphics
  » Other stuff that hasn’t been invented yet…

• In current technology transistors are free – CPU makers are looking for new ways to use transistors to make their chips more desirable.

• Intel’s big challenge: finding applications that require new hardware support, so that you will want to upgrade to a new computer to run them.