Very Early Operating Systems

- The first computers had no operating systems
- ☞ The concept isn't very meaningful on a plugboard or electromechanical device. . .
- Dedicated time only, manually managed

The Early Punchcard Era

- Punchcards — first used for data processing in the 1880s — made preparing programs easier
- But card readers were too slow; they tied up the (very expensive) CPU
- Printers had similar problems
- Solution: use slower, cheaper computer to copy input/output to/from mag tape

Computers Helping Themselves

- The realization that computers could help themselves was fairly early
- The first assembler was written by 1952
- Operating systems descend from this observation: that computers were useful for non-numeric and non-data processing work

Only a Partial Solution!

- Even mag tapes were slower than the CPU
- Changing tapes is slow and manual (these weren't cassette tapes!)
- Something had to read the next "job" from the tape
- Enter the "monitor" program

Monitor Programs

- Monitors did a small part of running the computer
- (Human operators were still very busy)
- The operator's primary interaction was with the monitor, not the hardware

Primitive Monitors

- Often loaded from punch cards or tapes
- The same was true for most compilers — disks were rare (the first model, in 1956, had 50 two-foot platters, and held 5M bytes total. . .)
- Frequent need to reload monitor — no memory protection

IBSYS for the IBM 7090/7094

$JOB

$EXECUTE IBJOB

$IBJOB GO, options

$IBFTC DECK1 options (Invoke Fortran)

(Fortran source program)

$DATA

...

7/8 **EOF** (Special rows 7-8 overpunch)

I/O Libraries

- Monitors also provided I/O libraries
- I/O was *complicated*, especially if done efficiently
- Better to do it right once
- Examples: card column processing, disk rotational delay, card reader “clutch point”

Card Reader Programming

- Start the card moving
- Wait for a column *interrupt*
- When it occurs, read the value of that column
- Store it in the next memory location
- When card finishes, decide *quickly* if another card should be read

3rd Generation Computers — 1964

- Something recognizably an operating system
- Memory protection; multiprogramming
- Programs stored on disk
- Quote from a 1963 paper on disk usage:

  > Some thought has also been given to the storage of frequently used problem codes on the disk. There does not appear to be sufficient benefits attainable at the present time with the present problem "mix." Also, the use of the disk for such activity is not attractive from a "computer-administrative" standpoint due to the addition and deletion of problems nor from the fact that less disk storage would be available for problem usage.

- On that monitor system, programmers allocated disk sectors themselves. . .

Why — Efficiency

- Mainframes still cost millions of dollars
- The CPU was idle while I/O was going on
- Save the cost of I/O computers

Why — Technology

- Important concepts: memory protection, virtual memory, interrupts
- Hardware had improved to the point that an OS was feasible:
  - Enough RAM (“core”) to hold it
  - Disks became the norm
  - Fast enough CPUs
- Beginnings of remote access

Multiprogramming

- Run several programs at once
- Protect each one's memory against the others
- When one program needs to do I/O, let another program use the CPU
- As necessary, pre-empt a CPU-bound program to let another run
- Use OS "daemons" for *spooling* — Simultaneous Peripheral Output OnLine

Disks

- Load OS and applications from disk
- Manage disk space allocation
- Store and retrieve files by name
- Provide some protection for files

Notable Early Operating Systems

- IBM OS/360 (its descendants are still with us)
  - 3 variants, one of which wasn't multiprogramming
- Multics
- TOPS-10 (For DEC's 36-bit machines)

Timesharing

- Support people on low-speed terminals
- Systems ranged from remote editing and batch submission to specialized restricted environments (BASIC) to operating systems designed for the purpose.
- Timesharing systems stressed responsiveness

Sample IBM Job Control Language (JCL)

```
//J1 JOB UR00045,BELLOVIN,TIME=(1,30),REGION=130K
//         EXEC FORTRAN
//SYSOUT   DD SYSOUT=A
//SYSDUMP  DD SYSOUT=A
//TEMP     DD UNIT=DISK,SPACE=(TRK,10),DSN=SMB.TEMP,
//            DISP=(NEW,DELETE)
//SYSIN    DD *
user data
....
/*
```

Controlling Monitors and Early Operating Systems

- Command language — moral equivalent of shell
- Supply accounting information
- Request resources — memory, CPU time, tapes, disk space
- Invoke user or system programs
- Batch-oriented — submit your “job” on punch cards, pick it up a few hours later

Enter the Minicomputer

- Improved technology permitted small, (relatively) cheap minicomputers ($20-40K)
- May or may not have had disks; programmed by punch cards or paper tape
- No memory protection — ran on monitors, not operating systems
- Followed the same evolutionary path as mainframes
- Most important early mini: DEC PDP-8

High-Level Languages and Operating Systems

- Early OSs written in assembler
- Notable exception: Burroughs 5000 designed for Algol (1961!)
- Multics (started in 1965) used PL/I
- Unix (early 1970s) used C
- In mid-1970s, IBM migrated to PL/S, a medium-level language

Microcomputers

- By the late 1970s, minicomputers had memory protection and real operating systems
- By then, microcomputers had come along
- These had no memory protection, no operating systems, and no disks. . .
- Windows XP is the first version of Windows that has few architectural concessions to this history
- But the habit of users running as Administrator dates back to that era
- *Many Windows security problems are due to the recapitulation of 1950s computer history!*

Modern Operating Systems

- Virtual memory, disk, multiprogramming
- Multi-CPU support common (and about to become ubiquitous)
- Designed for highly-interactive use (GUI)
- Network I/O crucial

Operating System Range

- Mainframes — vast I/O bandwidth (Unix, MVS, VM/CMS)
- Servers (Solaris, Linux, Windows 2003)
- Desktops — premium on interaction: mouse, graphics (Windows, Solaris, Linux)
- Embedded systems — often lack memory protection (VxWorks, Symbian)

Hardware Considerations

- What must the hardware do to accomodate a modern OS?
- Is there some reason, other than short-sightedness, why we've seen this endless repetition of design decisions?

Memory Protection

- Protect OS from being overwritten by user programs
- Prevent sensitive OS data from being read by users programs
- Protect user programs from reading or writing other programs' memory

Who Shall Watch the Watchmen?

- The ability to change memory protection must be unavailable to user programs
- In other words, the OS needs access to some parts of the hardware that application programs cannot access
- Two basic strategies: special (protected) memory areas and privileged instructions
- In practice, both are used

Special Memory Areas

- Writing to certain special memory addresses controls some aspects of the machine
- Example: memory protection can be turned on and off by writing to other memory areas
- These other areas are only accessible to the OS, using the exact same mechanism

Privileged Operations

- There exists a CPU state flag that allows or disallows certain operations
- When the OS is running, this flag is set to “privileged”
- The privileged flag can only be changed when running in privileged mode
- To run an application program, this flag is set to unprivileged

Returning to the OS

- When an application program is run, certain memory areas are inaccessible and the privilege flat is off
- By definition, the application cannot change this
- How do we return to the OS?
- Two mechanisms: interrupts and system calls

Interrupts

- External signal to the CPU
- Hardware saves the CPU state and loads new state
- New state includes privileged flag and access to OS memory
- Starts running OS's interrupt handler

I/O Interrupts

- Notify the CPU when an I/O operation is complete
- Vital to multiprogramming — let's the CPU run another job while I/O is taking place
- Eliminates need for busy wait

Other Interrupts

- Timer
- Button-push
- “Shoulder tap” from another CPU
- Environmental (power fail)

System Calls

- Initiated by application program
- Conceptually similar to a function call
- Again, the hardware saves the CPU state and loads new state
- Again, the new state includes privileged flag and access to OS memory
- Hardware starts executing OS's system call handler
- On some architectures, just another kind of interrupt

Other Traps

- Some traps happen due to program behavior
- Illegal operations, privileged operation, memory protection violation, divide-by-zero, etc.
- Virtual memory-related traps require OS assistance

Saving State

- What state does the hardware save?
- As little as possible — saving state is expensive
- Minimum: privileged status, memory access, return address (often combined in a *Program Status Word* (PSW))
- Anything else likely to be changed by early part of interrupt handle (i.e., branch condition code)
- Often collected into a *processor status word*

Where is State Saved?

- One obvious place is on the stack
- What stack?
- The OS can't trust the application's stack
- Conclusion: must save state in an OS-only location
- As we shall see, most of the state has to be saved under software control

Masking Interrupts

- The CPU can generally block interrupts temporarily
- The interrupt mask state is often part of the PSW
- PSW after an interrupt generally blocks new interrupts, to avoid recursion

Interrupts and Complexity

- Interrupts occur *asynchronously* to application programs
- In general, the interrupted program is not the one interested in the interrupt
- They are a major cause of OS complexity