Architecture and OS

To do ...

- Architecture impact on OS
- OS impact on architecture
- Next: OS components and structure
Computer architecture and OS

- OS is intimately tied to the hardware it runs on
  - OS design is impacted by it
  - OS needs result on new architectural features

Abstract model of a simple computer
Processor

- The brain with a basic operation cycle
  - Fetch next instruction
  - Decode it to determine type & operands
  - Execute it

- ... and a specific set of instructions
  - Architecture specific – x86 != ARM

- Plus registers, since memory is slow
  - General registers and special ones (PC, SP)

- Clearly simplistic model
  - Pipelining, superscalar, multicore
Memory

- **Ideal** – fast, large, cheap, persistent
- **Real** – storage hierarchy
  - Registers
    - Internal to the CPU & just as fast
  - Cache
    - If word needed is in cache, get it in ~2 cycles
  - Main memory ~ 100x slower
  - Hard disk – from nsecs to msecs (moving the arm 5-10msec)
  - Magnetic tape
  - Cloud storage
Architectural trends impact OS design...

- **Processing power**
  - Doubling every 18 months (100x per decade)
  - but power is a serious issue

- **Disk capacity**
  - Double every 12 months (1000x per decade)
    - 1961 ~ $4,440/MB
    - 2014 ~ $0.00004/MB

1961 IBM 1301
~26MB ~$115,500
(~890k in 2013)
Architectural trends impact OS design

- Memory
  - Same and for the same reason
    - 1980 64KB $405.00 ($6,480/MB) …
    - 2014 4GB $29.99 ($0.007/MB)*

- Optical bandwidth today
  - 10x as fast as disk capacity
  - 100x as fast as processor performance
  - Doubling every 9 months (Butter’s law)
  - Latency is the limiting factor, but there’s room
    - Time to get the HTML index page from popular sites is, in the median, 34 * c-latency (light’s shortest path rtt)

*http://www.jcmit.com/memoryprice.htm
... and OS needs shape the architecture

- Arch support can simplify/complicate OS tasks
  - E.g., early PC OS (DOS) lacked support for virtual memory, partly because HW lacked key features

- Features built primarily to support OS’s
  - Protected modes of execution (kernel vs. user)
  - System calls (and software interrupts)
  - Memory protection
  - I/O control operations
  - Timer (clock) operation
  - Interrupts and exceptions
  - Synchronization instructions
Consider timesharing

- Multiprogramming & timesharing are useful
  - Multiprogramming – “using different parts of the hardware at the same time for different tasks”
  - Timesharing – “several persons making use of the computer at the same time”

- but
  - How to protect programs from each other and the kernel from all?
  - How to handle relocation? OS may need to run a given program at != times from != locations
For protection

- Restrict some instructions to the OS
  - e.g. Directly access I/O devices

- How does the CPU know if a protected instructions should be executed?
  - Architecture must support 2+ modes of operation
  - Mode is set by status bit in a protected register
    - User programs execute in user mode, OS in kernel mode

- Protected instructions can only execute in kernel mode
Crossing protection boundaries

- How can apps do something privileged?
  - e.g. How do you save a file if you can't do I/O?

- User programs must call an OS procedure
  - Ask the OS to do it
  - OS defines a set of **system calls**
  - User-mode program makes a system call
  - How does the user to kernel-mode transition happen?
The system call …

- Causes an exception which vector to a kernel handler
- Passes a parameter indicating which syscall is
- Saves caller's state so it can be restored
  - *What would happen if the kernel didn’t save state?*
- OS must verify caller's parameters
  - *Why should it do that?*
- Must have a way to go back to user once done
  - A special instruction sets PC to the return address and the execution mode to user
Crossing protection boundaries

- A system call

```c
read(int fd, void *buffer, int numbytes)
```

User mode

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

Kernel mode

- trap handler

- Save application state
- Verify system call number
- Find sys_read handler

- sysread() kernel routine

- Verify args
- Initiate read
- Choose next proc to run
- Setup return values
- Restore application state

- ERET instruction

- Restore PC
- Enter user mode

- A bit like a regular subroutine call, right?
Exception handling and protection

- All entries to the OS use the same mechanism
  - Acquiring privileged mode and branching to trap handler are inseparable

- Interrupts, exceptions and traps
  - Interrupt: asynchronous, caused by an external HW event
  - Exception: synchronous; unexpected, automatically generated (coerced, no requested); divide by zero
  - Trap: synchronous; programmer initiated, expected transfer of control to a special handler
Next issue – Memory relocation

- Simplest model – base + limit
  - Base (start) of program + limit registers
  - Used by CDC 6600 (supercomputer) and Intel 8088
  - Changing program means changing base+limit
  - Solves relocation and protection (on CDC 6600)
  - Cost 2 registers + cycle time incr

- More sophisticated alternatives
  - 2 base and 2 limit registers for text & data; allow sharing program text
  - Paging, segmentation, virtual memory

If ≥ base and < base + limit, OK else trap to OS with error
I/O Device

- Device + Controller (simpler I/F to OS; think SCSI)
  - Read sector x from disk y → (disk, cylinder, sector, head), ...

How does the kernel start an I/O?

- Special I/O instructions
- Memory-mapped I/O

How does it notice when the I/O is done?

- Polling – are we done yet?
- Interrupts – let me know when you are done?

How does it exchange data with I/O device?

- Programmed I/O
- Direct Memory Access (DMA)
OS control flow

- OSs are event driven
  - Once booted, all entry to kernel happens as result of an event (e.g. signal by an interrupt), which
    - Immediately stops current execution
    - Changes to kernel mode, event handler is called

- Kernel defines handlers per event type
  - Specific types are defined by the architecture
    - e.g. timer event, I/O interrupt, system call trap
How can the OS retain control when a program gets stuck in an infinite loop?
- HW timer that generates a periodic interrupt
- Before it transfers to a user program, OS loads timer with a time to interrupt
- When time's up, interrupt transfers control back to OS
  - OS pick a program to schedule next (which one?)

Should the timer be privileged?
- For reading or for writing?
Synchronization

- Issues with interrupts
  - Executing code can interfere with interrupted code
  - OS must be able to synchronize concurrent processes

- Synchronization
  - Guarantee that short instruction sequences (e.g. read-modify-write) execute atomically
  - Two methods
    - Turn off interrupts, execute sequence, re-enable interrupts
    - Have special, complex atomic instructions – test-and-set

Management of concurrency & asynchronous events is the biggest difference between systems-level & traditional app programming
This is far from over – new architectural features are still being introduced
- Support for virtual machine monitors
- Hardware transaction support
- Support for security
- Low latency persistent memory
- ...

Transistors are free so Intel/AMD/… need to find applications that require new hardware that you would want to buy …