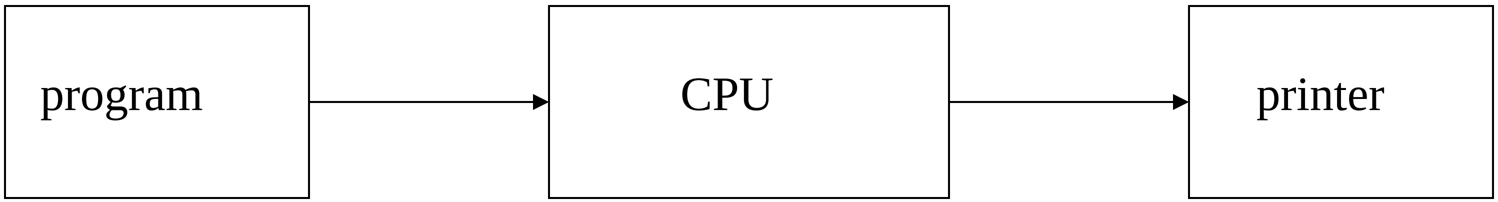
Advanced Operating Systems (CS 202)

OS Evolution and Organization

Dawn of computing



- Pre 1950 : the very first electronic computers
  - valves and relays
  - single program with dedicated function
- Pre 1960 : stored program valve machines
  - single job at a time; OS is a program loader

Phase 0 of OS Evolution (40s to 1955)

- No OS
  - Computers are exotic, expensive, large, slow experimental equipment
  - Program in machine language and using plugboards
  - User sits at console: no overlap between computation, I/O, user thinking, etc..
    - Program manually by plugging wires in
    - Goal: number crunching for missile computations
- Imagine programming that way
  - Painful and slow

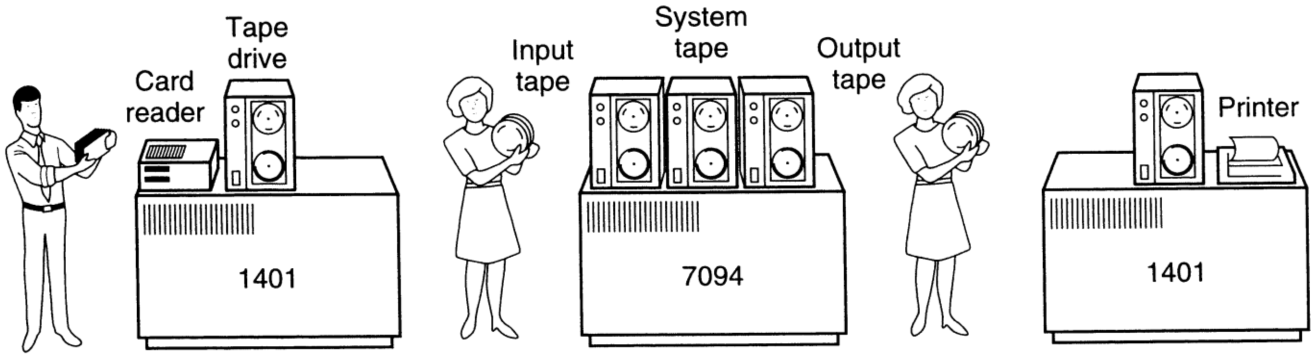
OS progress in this period

- Libraries of routines that are common
  - Including those to talk to I/O devices
  - Punch cards (enabling copying/exchange of these libraries) a big advance!
  - Pre-cursor to OS

Phase 1: 1955-1970

- Computers expensive; people cheap
  - Use computers efficiently – move people away from machine
  - OS becomes a batch monitor
    - Loads a job, runs it, then moves on to next
    - If a program fails, OS records memory contents somewhere
    - More efficient use of hardware but increasingly difficult to debug

- Batch systems on *mainframe* computers
- collections of jobs made up into a *batch*
- example: IBM 1401/7094
  - card decks spooled onto magnetic tape and from tape to printer



- example: English Electric Leo KDF9
  - 32K 48-bit words, 2μsec cycle time
  - punched paper-tape input ‘walk-up’ service or spooling via mag tape

Advances in technology in this stage

- Data channels and interrupts
  - Allow overlap of I/O and computing
  - Buffering and interrupt handling done by OS
  - Spool (buffer) jobs onto “high speed” drums

Phase 1, problems

- Utilization is low (one job at a time)
- No protection between jobs
- Short jobs wait behind long jobs
  - So, we can only run one job at a time
- Coordinating concurrent activities
- Still painful and slow (but less so?)

Advances in OS in this period

- Hardware provided memory support (protection and relocation)
- Multiprogramming (not to be confused with time sharing)
- Scheduling: let short jobs run first
- OS must manage interactions between concurrent things
  - Starts emerging as a field/science
- OS/360 from IBM first OS designed to run on a family of machines from small to large

Some important projects

- Atlas computer/OS from Manchester U. (late 50s/early 60s)
  - First recognizable OS
  - Separate address space for kernel
  - Early virtual memory

- THE Multiprogramming system (early 60s)
  - Introduced semaphores
  - Attempt at proving systems correct; interesting software engineering insights

Not all is smooth

- Operating systems didn't really work
- No software development or structuring tools; written in assembly
- OS/360 introduced in 1963 but did not really work until 1968
  - Reported on in mythical man month
- Extremely complicated systems
  - 5-7 years development time typical
  - Written in assembly, with no structured programming
  - Birth of software engineering?

Phase 2: 1970s

- Computers and people are expensive
  - Help people be more productive
  - Interactive time sharing: let many people use the same machine at the same time
  - Emergence of minicomputers
    - Terminals are cheap
  - Keep data online on fancy file systems
  - Attempt to provide reasonable response times (Avoid thrashing)

Important advances and systems

- Compatible Time-Sharing System (CTSS)
  - MIT project (demonstrated in 1961)
  - One of the first time sharing systems
  - Corbato won Turing award in 1990
  - Pioneered much of the work in scheduling
  - Motivated MULTICS

MULTICS

- Jointly developed by MIT, Bell Labs and GE
- Envisioned one main computer to support everyone
  - People use computing like a utility like electricity – sound familiar? Ideas get recycled
- Many many fundamental ideas: protection rings, hierarchical file systems, devices as files, …
- Building it was more difficult than expected
- Technology caught up

Unix appears

- Ken Thompson, who worked on MULTICS, wanted to use an old PDP-7 laying around in Bell labs
- He and Dennis Richie built a system designed by programmers for programmers
- Originally in assembly. Rewritten in C
  - If you notice for the paper, they are defending this decision
  - However, this is a new and important advance: portable operating systems!
- Shared code with everyone (particularly universities)

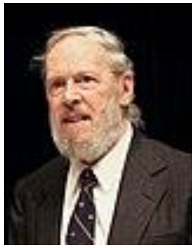
Unix (cont'd)

- Berkeley added support for virtual memory for the VAX
- DARPA selected Unix as its networking platform in arpanet
- Unix became commercial
  - …which eventually lead Linus Torvald to develop Linux

Some important ideas in Unix

- OS written in a high level language
- OS portable across hardware platforms
  - Computing is no longer a pipe stove/vertical system
- Pipes
  - E.g., grep foo file.txt | wc -l
- Mountable file systems
- Many more (we'll talk about unix later)
- 1983 Turing Award

Ken Thompson



Dennis M. Ritchie

Phase 3: 1980s

- Computers are cheap, people expensive
  - Put a computer in each terminal
  - CP/M from DEC first personal computer OS (for 8080/85) processors
  - IBM needed software for their PCs, but CP/M was behind schedule
  - Approached Bill Gates to see if he can build one
  - Gates approached Seattle computer products, bought 86-DOS and created MS-DOS
  - Goal: finish quickly and run existing CP/M software
  - OS becomes subroutine library and command executive

New advances in OS

- PC OS was a regression for OS
  - Stepped back to primitive phase 1 style OS leaving the cool developments that occurred in phase 2
- Academia was still active, and some developments still occurred in mainframe and workstation space

Phase 4: Networked systems 1990s to 2010s

- Machines can talk to each other
  - its all about connectivity
- We want to share data not hardware
- Networked applications drive everything
  - Web, email, messaging, social networks, …
- Protection and multiprogramming less important for personal machines
  - But more important for servers

Phase 4, continued

- Market place continued horizontal stratification
  - ISPs (service between OS and applications)
  - Information is a commodity
  - Advertising a new marketplace

- New network based architectures
  - Client server
  - Clusters
  - Grids
  - Distributed operating systems
  - Cloud computing (or is that phase 5?)

New problems

- Large scale
  - Google file system, mapreduce, …
- Concurrency at large scale
  - ACID (Atomicity, Consistency, Isolation and Durability) in Internet Scale systems
    - Very large delays
    - Partitioning
- Security and Privacy

Phase 5: 2010s -- ??

- New generation?
- Mobile devices that are powerful
- Sensing: location, motion, …
- Cyberphysical systems
- Computing evolving beyond networked systems
  - But OS for them looks largely the same
  - Is that a good idea?

OS model and Architectural Support

Sleeping Beauty Model

- Answer: Sleeping beauty model
  - Technically known as *controlled direct execution*
  - OS runs in response to “events”; we support the switch in hardware
  - Only the OS can manipulate hardware or critical system state
- Most of the time the OS is sleeping
  - Good! Less overhead
  - Good! Applications are running directly on the hardware

What do we need from the architecture/CPU?

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.
  - Controlling access

- Generating and handling "events"
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap

- Other stuff
  - Mechanisms to handle concurrency, Isolation, virtualization …

Protected Instructions

- OS must have exclusive access to hardware and critical data structures

- Only the operating system can
  - Directly access I/O devices (disks, printers, etc.)
    - Security, fairness (why?)
  - Manipulate memory management state
    - Page table pointers, page protection, TLB management, etc.
  - Manipulate protected control registers
    - Kernel mode, interrupt level
  - Halt instruction (why?)

Privilege mode

- Hardware restricts privileged instructions to OS
- Q: How does the HW know if the executed program is OS?
  - HW must support (at least) two execution modes: OS (kernel) mode and user mode
- Mode kept in a status bit in a protected control register
  - User programs execute in user mode
  - OS executes in kernel mode (OS == “kernel”)
  - CPU checks mode bit when protected instruction executes
  - Attempts to execute in user mode trap to OS

Switching back and forth

- Going from higher privilege to lower privilege
  - Easy: can directly modify the mode register to drop privilege

- But how do we escalate privilege?
  - Special instructions to change mode
    - System calls (`int 0x80, syscall, svc`)
    - Saves context and invokes designated handler
      - You jump to the privileged code; you cannot execute your own
    - OS checks your syscall request and honors it only if safe
  - Or, some kind of event happens in the system

Types of Arch Support

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.
  - Controlling access

- Generating and handling "events"
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap

- Other stuff

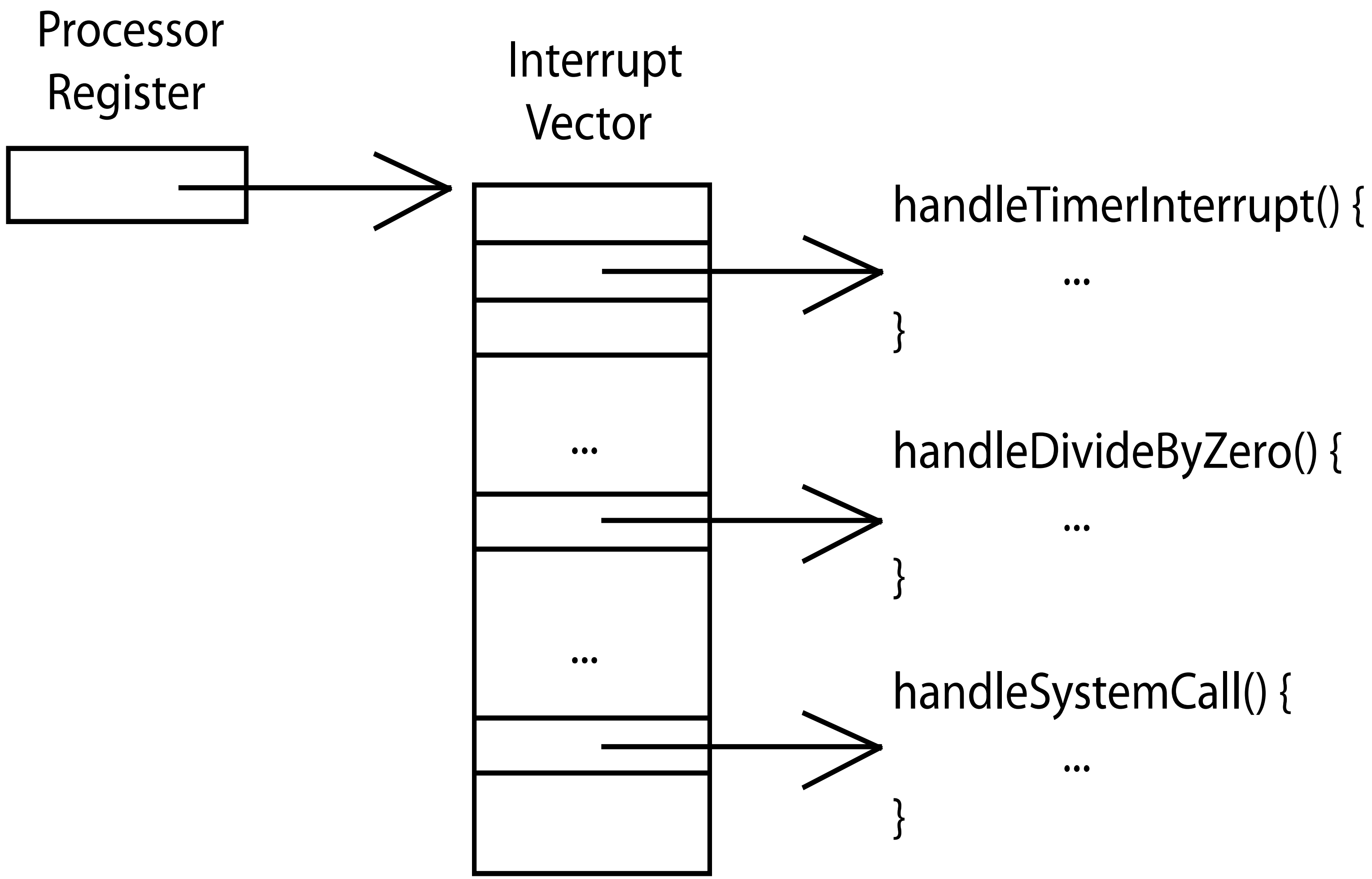
Events

- An event is an “unnatural” change in control flow
  - Events immediately stop current execution
  - Changes mode, context (machine state), or both

- The kernel defines a handler for each event type
  - Event handlers always execute in kernel mode
  - The specific types of events are defined by the machine

- Once the system is booted, OS is one big event handler
  - all entry to the kernel occurs as the result of an event

Handling events – Interrupt vector table



Categorizing Events

- This gives us a convenient table:

	Unexpected	Deliberate
Synchronous	fault	syscall trap
Asynchronous	interrupt	signal

- Terms may be slightly different by OS and architecture
  - E.g., POSIX signals, asynch system traps, async or deferred procedure calls

Faults

- Hardware detects and reports “exceptional” conditions
  - Page fault, memory access violation (unaligned, permission, not mapped, bounds…), illegal instruction, divide by zero

- Upon exception, hardware “faults” (verb)
  - Must save state (PC, regs, mode, etc.) so that the faulting process can be restarted
  - Invokes registered handler

Handling Faults

- Some faults are handled by “fixing” the exceptional condition and returning to the faulting context
  - Page faults cause the OS to place the missing page into memory
  - Fault handler resets PC of faulting context to re-execute instruction that caused the page fault

Handling Faults

- The kernel may handle unrecoverable faults by killing the user process
  - Program fault with no registered handler
  - Halt process, write process state to file, destroy process
  - In Unix, the default action for many signals (e.g., SIGSEGV)

- What about faults in the kernel?
  - Dereference NULL, divide by zero, undefined instruction
  - These faults considered fatal, operating system crashes
  - Unix panic, Windows "Blue screen of death"
    - Kernel is halted, state dumped to a core file, machine locked up

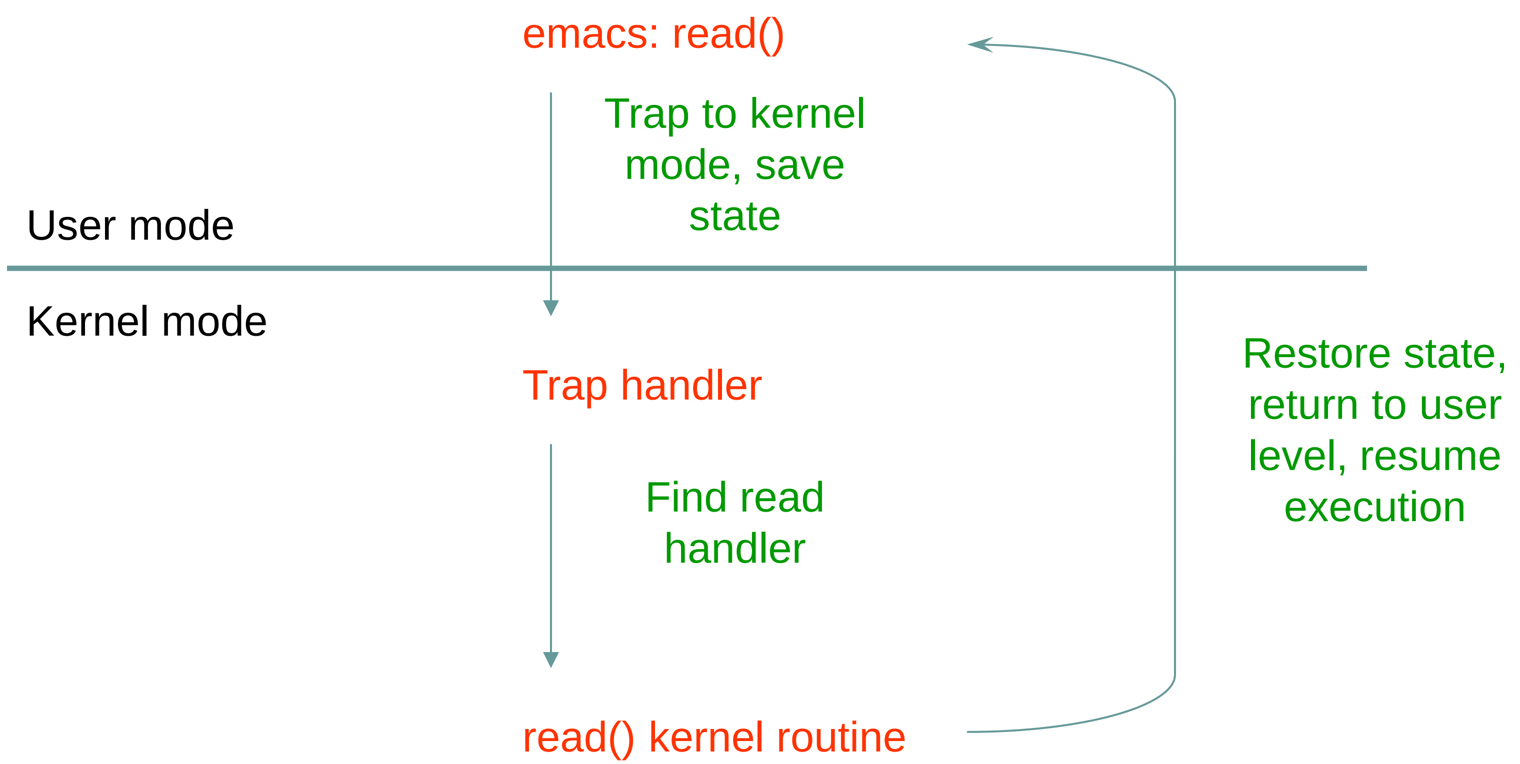
Categorizing Events

	Unexpected	Deliberate
Synchronous	fault	syscall trap
Asynchronous	interrupt	signal

System Calls

- For a user program to do something "privileged" (e.g., I/O) it must call an OS procedure
  - Known as crossing the protection boundary, or a protected procedure call
- Hardware provides a system call instruction that:
  - Causes an exception, which invokes a kernel handler
    - Passes a parameter determining the system routine to call
  - Saves caller state (PC, regs, mode) so it can be restored
    - Why save mode?
  - Returning from system call restores this state

System Call


emacs: read()
Trap to kernel mode, save state
User mode
Kernel mode
Trap handler
Find read handler
read() kernel routine
Restore state, return to user level, resume execution

System Call Questions

- There are hundreds of syscalls. How do we let the kernel know which one we intend to invoke?
  - Before issuing **int $0x80** or **sysenter**, set **%eax/%rax** with the syscall number
- System calls are like function calls, but how to pass parameters?
  - Just like calling convention in syscalls, typically passed through **%ebx, %ecx, %edx, %esi, %edi, %ebp**

More questions

- How to reference kernel objects (e.g., files, sockets)?
  - Naming problem – an integer mapped to a unique object
    - int fd = open("file"); read(fd, buffer);
  - Why can't we reference the kernel objects by memory address?

Categorizing Events

	Unexpected	Deliberate
Synchronous	fault	syscall trap
Asynchronous	interrupt	software interrupt

- Interrupts signal asynchronous events
  - I/O hardware interrupts
  - Software and hardware timers

Timer

- The key to a timesharing OS

- The fallback mechanism by which the OS reclaims control
  - Timer is set to generate an interrupt after a period of time
    - Setting timer is a privileged instruction
    - When timer expires, generates an interrupt
      - Handled by the OS, forcing a switch from the user program
    - Basis for OS scheduler *(more later…)*

- Also used for time-based functions (e.g., *sleep()*)

I/O Control

- I/O issues
  - Initiating an I/O
  - Completing an I/O
- Initiating an I/O
  - Special instructions
  - Memory-mapped I/O
    - Device registers mapped into address space
    - Writing to address sends data to I/O device

I/O using Interrupts

- Interrupts are the basis for asynchronous I/O
  - OS initiates I/O
  - Device operates independently of rest of machine
  - Device sends an interrupt signal to CPU when done
  - OS maintains a vector table containing a list of addresses of kernel routines to handle various events
  - CPU looks up kernel address indexed by interrupt number, context switches to routine

I/O Example

1. Ethernet receives packet, writes packet into memory
2. Ethernet signals an interrupt
3. CPU stops current operation, switches to kernel mode, saves machine state (PC, mode, etc.) on kernel stack
4. CPU reads address from vector table indexed by interrupt number, branches to address (Ethernet device driver)
5. Ethernet device driver processes packet (reads device registers to find packet in memory)
6. Upon completion, restores saved state from stack

Summary

- Protection
  - User/kernel modes
  - Protected instructions
- System calls
  - Used by user-level processes to access OS functions
  - Access what is "in" the OS
- Exceptions
  - Unexpected event during execution (e.g., divide by zero)
- Interrupts
  - Timer, I/O