CS140 – Operating Systems

Instructor: David Mazières
CAs: Varun Arora, Chia-Hui Tai, Megan Wachs

Stanford University

Administrivia

- **Class web page:** `http://cs140.scs.stanford.edu/`
  - All assignments, handouts, lecture notes on-line
- **Textbook:** ***Operating System Concepts, 7th Edition*, by Silberschatz, Galvin, and Gagne**
- **Staff mailing list: cs140-staff@scs.stanford.edu**
  - Please always email staff mailing list for help
- **Newsgroup: su.class.cs140 ← main discussion forum**
- **Key dates:**
  - Lectures: TTh 4:15-5:30, Gates B01
  - Section: Some Fridays 4:15-5:05, Gates B01
  - Midterm: Thursday, October 25, 4:15-5:30pm
  - Final: Wednesday, December 12, 12:15pm

Lecture videos

- **Lectures will be televised for SCPD students**
  - Can also watch if you miss a lecture, or to review
  - But resist temptation to miss a bunch of lectures and watch them all at once
- **SCPD students welcome to attend lecture in person**
  - 4:15pm lecture time conveniently at end of day
  - Many parking spaces don't require permit after 4pm
- **Other notes for SCPD students:**
  - Please attend exams in person if possible
  - Feel free to use newsgroup to find project partners

Course topics

- **Threads & Processes**
- **Concurrency & Synchronization**
- **Scheduling**
- **Virtual Memory**
- **I/O**
- **Disks, File systems, Network file systems**
- **Protection & Security**
- **Non-traditional operating systems**

Course goals

- **Introduce you to operating system concepts**
  - Hard to use a computer without interacting with OS
  - Understanding the OS makes you a more effective programmer
- **Cover important systems concepts in general**
  - Caching, concurrency, memory management, I/O, protection
- **Teach you to deal with larger software systems**
  - Programming assignments much larger than many courses
  - Warning: Many people will consider course very hard
  - In past, majority of people report $\geq$15 hours/week
- **Prepare you to take graduate OS classes (CS240, 240[a-z])**

Programming Assignments

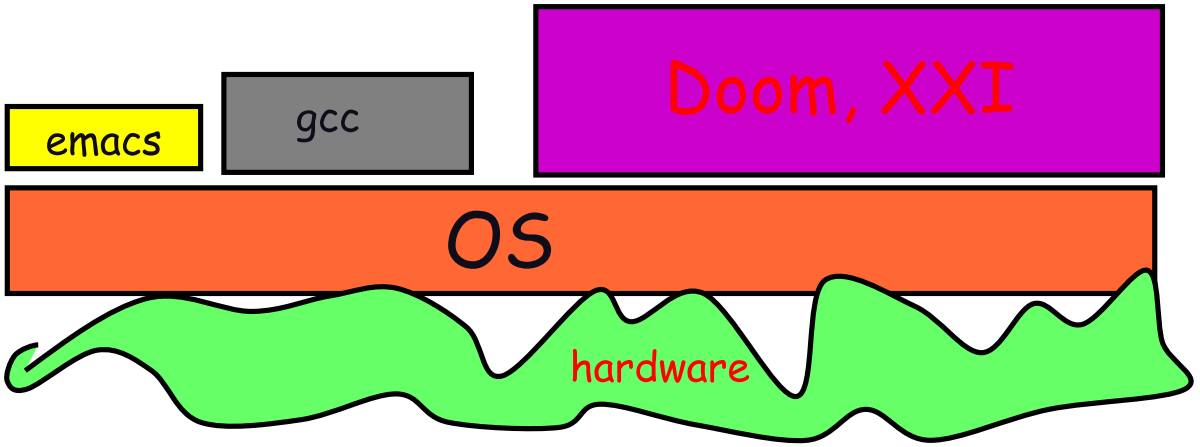
- **Implement parts of Pintos operating system**
  - Built for x86 hardware, you will use hardware emulator
- **Four implementation projects:**
  - Threads
  - Multiprogramming
  - Virtual memory
  - File system
- **First project distributed at end of this week**
- **Attend section this Friday for project 1 overview**
- **Implement projects in groups of up to 3 people**
  - Pick your partners today
  - Lecture will end early so that you can do this

Grading

- **50% of final grade based on midterm and final exams**
  - max (final, (midterm + final) /2)
- **50% of final grade based on projects**
  - For each project, 50% of grade based on test cases
    Please, please, please turn in working code, or **no credit**
  - Remaining 50% based on design, outlined in document
- **Do not look at other people's solutions to projects**
- **Can read but don't copy other OSes (Linux, Open/FreeBSD, etc.)**
- **Cite any code that inspired your code**

What is an operating system?

- **Layer between applications and hardware**



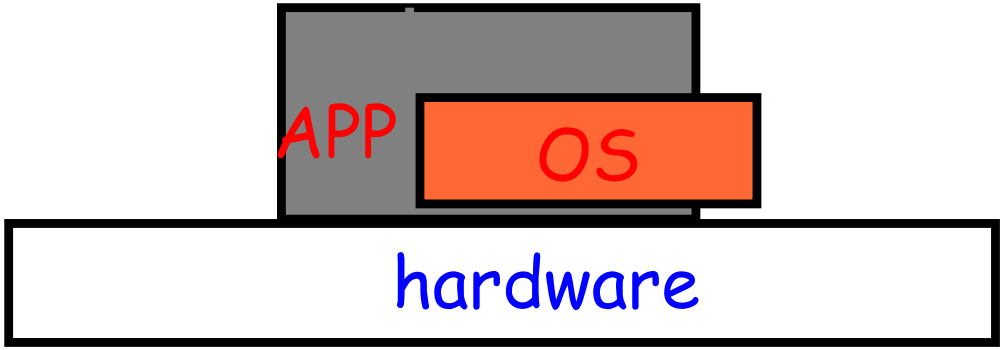
- **Makes hardware useful to the programmer**
- **[Usually] Provides abstractions for applications**
  - Manages and hides details of hardware
  - Accesses hardware through low/level interfaces unavailable to applications
- **[Often] Provides protection**
  - Prevents one process/user from clobbering another

Why study operating systems?

- **Operating systems are a maturing field**
  - Most people use a handful of mature OSes
  - Hard to get people to switch operating systems
  - Hard to have impact with a new OS
- **High-performance servers are an OS issue**
  - Face many of the same issues as OSes
- **Resource consumption is an OS issue**
  - Battery life, radio spectrum, etc.
- **Security is an OS issue**
  - Hard to achieve security without a solid foundation
- **New "smart" devices need new OSes**

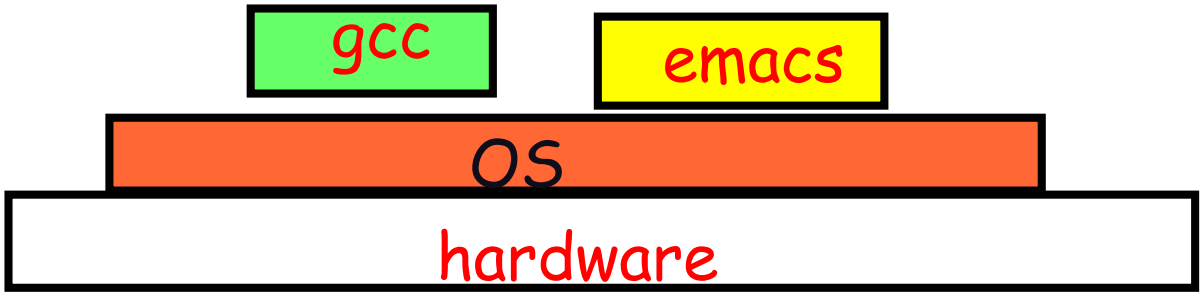
Primitive Operating Systems

- **Just a library of standard services [no protection]**



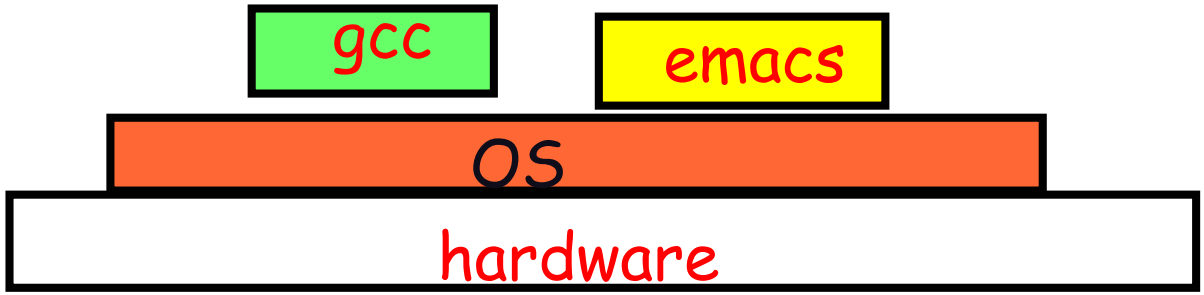
  - Standard device drivers, interrupt handlers, I/O
- **Simplifying assumptions**
  - System runs one program at a time
  - No bad users or programs (often bad assumption)
- **Problem: Poor utilization**
  - ...of hardware (e.g., CPU idle while waiting for disk)
  - ...of human user (must wait for each program to finish)

Multitasking



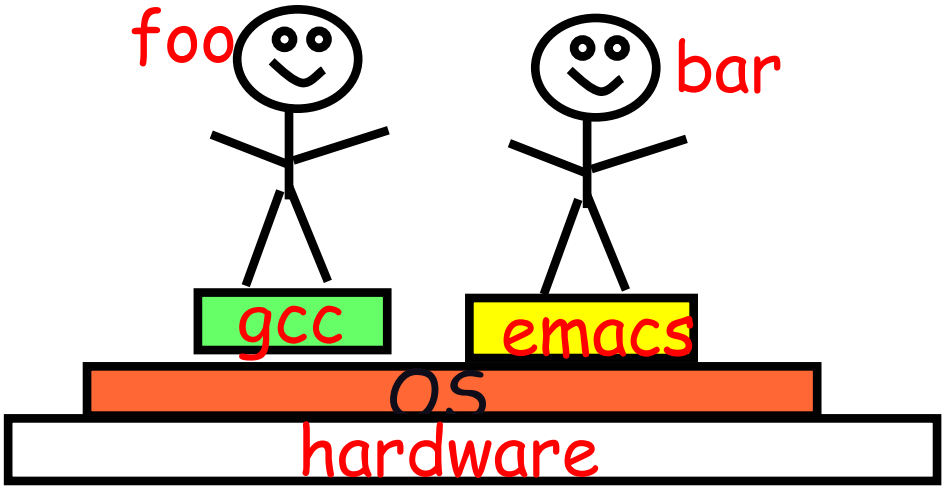
- **Idea: Run more than one process at once**
  - When one process blocks (waiting for disk, network, user input, etc.) run another process
- **Problem: What can ill-behaved process do?**

Multitasking



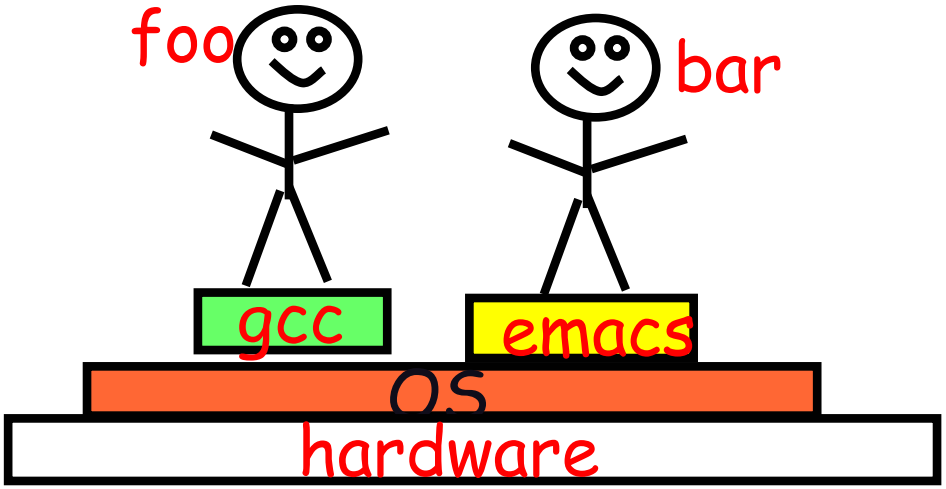
- **Idea: Run more than one process at once**
  - When one process blocks (waiting for disk, network, user input, etc.) run another process
- **Problem: What can ill-behaved process do?**
  - Go into infinite loop and never relinquish CPU
  - Scribble over other processes' memory to make them fail
- **OS provides mechanisms to address these problems**
  - *Preemption* – take CPU away from looping process
  - *Memory protection* – protect process's memory from one another

Multi-user OSes



- **Many OSes use *protection* to serve distrustful users**
- **Idea: With N users, system not N times slower**
  - Users' demands for CPU, memory, etc. are bursty
  - Win by giving resources to users who actually need them
- **What can go wrong?**

Multi-user OSes

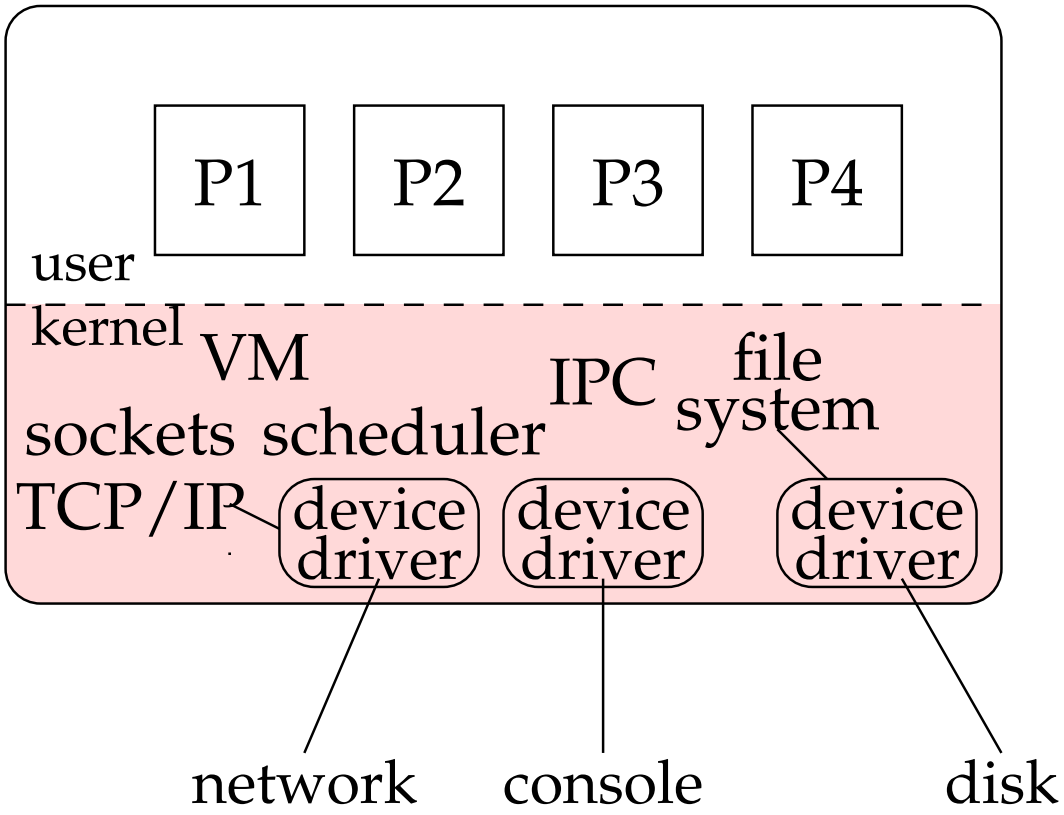


- **Many OSes use *protection* to serve distrustful users**
- **Idea: With N users, system not N times slower**
  - Users' demands for CPU, memory, etc. are bursty
  - Win by giving resources to users who actually need them
- **What can go wrong?**
  - Users are gluttons, use too much CPU, etc. (need policies)
  - Total memory usage greater than in machine (must virtualize)
  - Super-linear slowdown with increasing demand (thrashing)

Protection

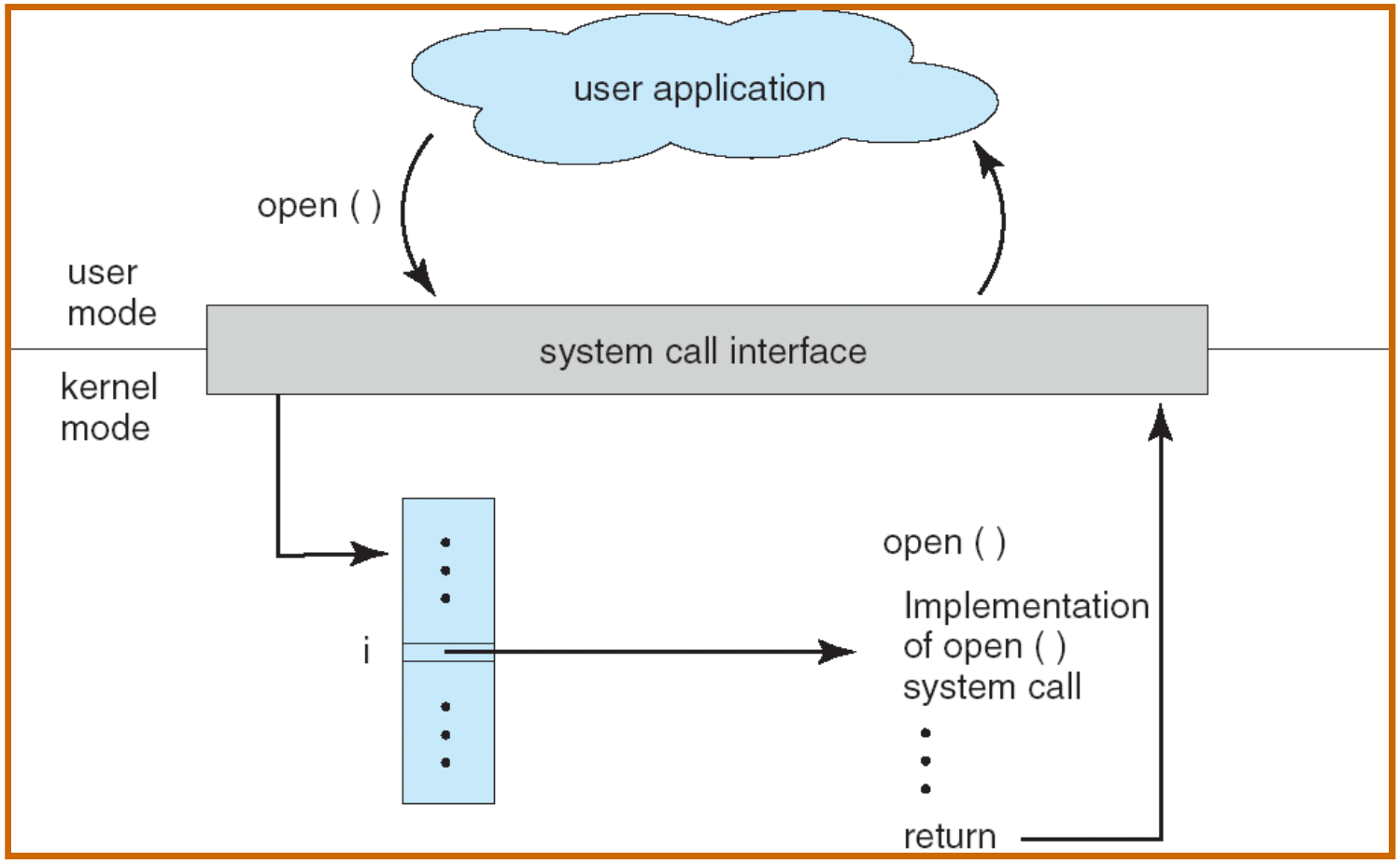
- **Mechanisms that isolate bad programs and people**
- **Pre-emption:**
  - Give application a resource, take it away if needed elsewhere
- **Interposition:**
  - OS between application and "stuff"
  - track all pieces that application allowed to use (e.g., in table)
  - on every access, look in table to check that access legal
- **Privileged/unprivileged mode:**
  - Applications unprivileged (user/unprivileged mode)
  - OS privileged (privileged/supervisor mode)
  - Protection operations can only be done in privileged mode

Typical OS structure



- **Most software runs as user-level processes (P[1-4])**
- **OS *kernel* runs in *privileged* mode [shaded]**
  - Creates/deletes processes
  - Provides access to hardware

System calls

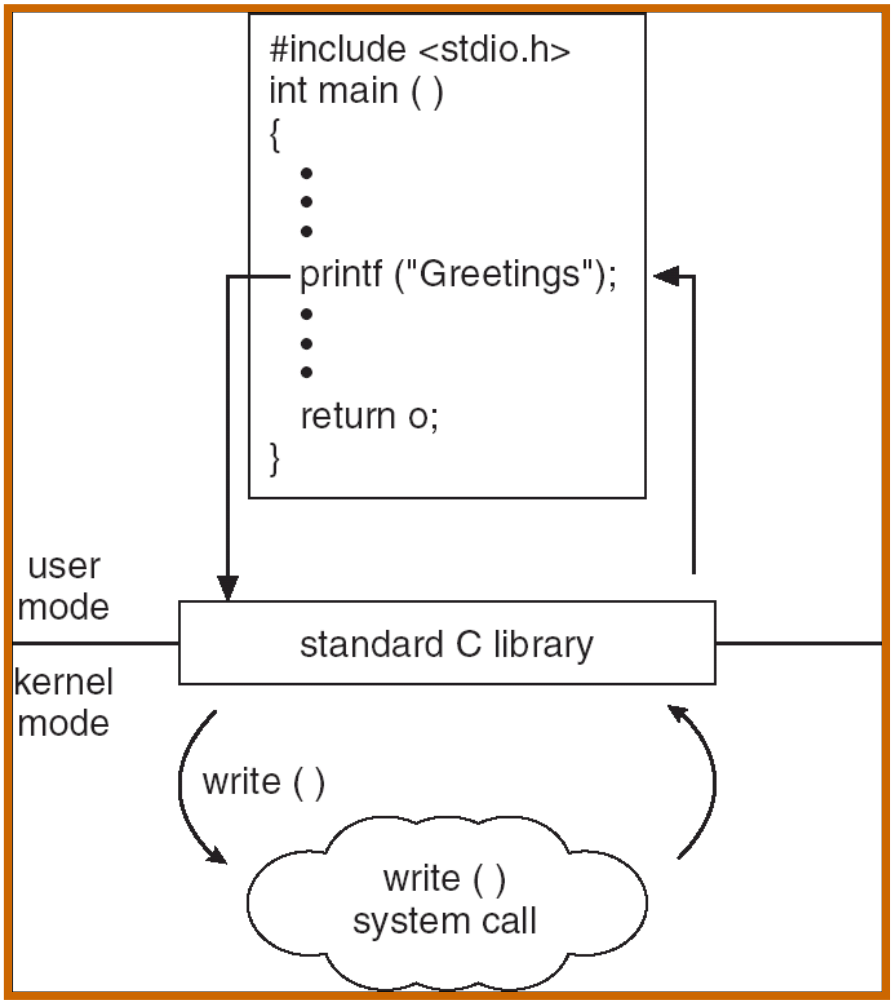


- **Applications can invoke kernel through *system calls***
  - Special instruction transfers control to kernel
  - …which dispatches to one of few hundred syscall handlers

System calls (continued)

- **Goal: Do things app. can't do in unprivileged mode**
  - Like a library call, but into more privileged kernel code
- **Kernel supplies well-defined *system call* interface**
  - Applications set up syscall arguments and *trap* to kernel
  - Kernel performs operation and returns result
- **Higher-level functions built on syscall interface**
  - `printf`, `scanf`, `gets`, etc. all user-level code
- **Example: POSIX/UNIX interface**
  - `open, close, read, write, ...`

System call example



- **Standard library implemented in terms of syscalls**
  - *printf* – in libc, has same privileges as application
  - calls *write* – in kernel, which can send bits out serial port

Different system contexts

- **A system can typically be in one of several contexts**
- ***User-level* – running an application**
- **Kernel *"top half"* (called "bottom half" in Linux)**
  - Running kernel code on behalf of a particular process
  - E.g., performing system call
  - Also exception (mem. fault, numeric exception, etc.)
  - Or executing a kernel-only process (e.g., network file server)
- **Kernel code not associated w. a process**
  - Timer interrupt (hardclock)
  - Device interrupt
  - Software interrupt
- **Context switch code – changing address spaces**

Transitions between contexts

- **User → top half: syscall, page fault**
- **User/top half → device/timer interrupt: hardware**
- **Top half → user/context switch: return**
- **Top half → context switch: sleep**
- **Context switch → user/top half**

CPU preemption

- **Protection mechanism to prevent monopolizing CPU**
- **E.g., kernel programs timer to interrupt every 10 ms**
  - Must be in supervisor mode to write appropriate I/O registers
  - User code cannot re-program interval timer
- **Kernel sets interrupt to vector back to kernel**
  - Regains control whenever interval timer fires
  - Gives CPU to another process if someone else needs it
  - Note: must be in supervisor mode to set interrupt entry points
  - No way for user code to hijack interrupt handler
- **Result: Cannot monopolize CPU with infinite loop**
  - At worst get $1/N$ of CPU with N CPU-hungry processes

Protection is not security

- **How *can* you monopolize CPU?**

Protection is not security

- **How *can* you monopolize CPU?**
- **Use multiple processes**
- **Until recently, could wedge many OSes with**

```
int main() { while(1) fork(); }
```

  - Keeps creating more processes until system out of proc. slots
- **Other techniques: use all memory (chill program)**
- **Typically solved with technical/social combination**
  - Technical solution: Limit processes per user
  - Social: Reboot and yell at annoying users
  - Social: Pass laws (often debatable whether a good idea)

Address translation

- **Protect mem. of one program from actions of another**
- **Definitions**
  - *Address space*: all memory locations a program can name
  - *Virtual address*: addresses in process' address space
  - *Physical address*: address of real memory
  - *Translation*: map virtual to physical addresses
- **Translation done on every load and store**
  - Modern CPUs do this in hardware for speed
- **Idea: If you can't name it, you can't touch it**
  - Ensure one process's translations don't include any other process's memory

More memory protection

- **CPU allows kernel-only virtual addresses**
  - Kernel typically part of all address spaces,
    e.g., to handle system call in same address space
  - But must ensure apps can't touch kernel memory
- **CPU allows disabled virtual addresses**
  - Catch and halt buggy program with wild accesses
  - Make virtual memory seem bigger than physical
    (e.g., bring a page in from disk only when accessed)
- **CPU enforced of read-only virtual addresses useful**
  - E.g., allows sharing of code pages between processes
  - Plus many other optimizations
- **CPU enforced execute disable of VAs**
  - Makes certain code injection attacks harder

Resource allocation & performance

- **Multitasking permits higher resource utilization**
- **Simple example:**
  - Process downloading large file mostly waits for network
  - You play a game while downloading the file
  - Higher CPU utilization than if just downloading
- **Complexity arises with cost of switching**
- **Example: Say disk 1,000 times slower than memory**
  - 100 MB memory in machine
  - 2 Processes want to run, each use 100 MB
  - Can switch processes by swapping them out to disk
  - Faster to run one at a time than keep context switching

Useful properties to exploit

- **Skew**
  - 80% of time taken by 20% of code
  - 10% of memory absorbs 90% of references
  - basis behind cache: place 10% in fast memory, 90% in slow, usually looks like one big fast memory
- **Past predicts future (a.k.a. temporal locality)**
  - What's the best cache entry to replace?
  - If past = future, then least-recently-used entry
- **Note conflict between fairness & throughput**
  - Higher throughput (fewer cache misses, etc.) to keep running same process
  - But fairness says should periodically preempt CPU and give it to next process