

CS140 – Operating Systems

Instructor: David Mazières

CAs: Emma Dauterman, Peter Dunn, and Francis Yan

Stanford University

Outline

1 Administrivia

2 Substance

Administrivia

- **Class web page:** <http://cs140.scs.stanford.edu/>
 - All assignments, handouts, lecture notes on-line
- **Textbook:** *Operating System Concepts, 8th Edition*, by Silberschatz, Galvin, and Gagne
 - 8th edition is available for ~\$10 used
 - 9th, 10th editions way more expensive, haven't looked at them
 - Trying to ween class from textbook anyway—consider it optional
- **Goal is to make lecture slides the primary reference**
 - Almost everything I talk about will be on slides
 - PDF slides contain [links](#) to further reading about topics
 - Please download slides from [class web page](#)
 - Will try to post before lecture for taking notes (but avoid calling out answers if you read them from slides)

Administrivia 2

- Google group **19wi-cs140** is main discussion forum
- **Staff mailing list:** `cs140-staff@scs.stanford.edu`
 - Please use google group for questions other people might have
 - Otherwise, please mail staff list, not individual instructors
- **Key dates:**
 - Lectures: MW 3:00pm–4:20pm, Skilling
 - Section: 6 Fridays, time/location TBD, starting this Friday
 - Midterm exam: Wednesday, February 13, 3:00–4:20pm (in class)
 - Final exam: Monday, March 18th, 3:30pm–6:30pm
- **Possibly some joint sections with CS140e**
- **Exams open note, but not open book**
 - Bring notes, slides, any printed materials *except* textbook
 - No electronic devices permitted

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**
- **Other notes for SCPD students:**
 - Please attend exams in person if possible
 - Feel free to use google group to find project partners

Course topics

- **Threads & Processes**
- **Concurrency & Synchronization**
- **Scheduling**
- **Virtual Memory**
- **I/O**
- **Disks, File systems**
- **Protection & Security**
- **Virtual machines**
- **Note: Lectures will often take Unix as an example**
 - Most current and future OSes heavily influenced by Unix
 - Won't talk much about Windows

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 ≥ 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 emulators
- **One setup homework (lab 0) due this Friday**
- **Four implementation projects:**
 - Threads
 - User processes
 - Virtual memory
 - File system
- **Lab 1 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 may end early for this)
 - Please disclose if you are planning to take class pass/fail

Grading

- **No incompletes**
 - Talk to instructor ASAP if you run into real problems
- **Final grades posted March 26**
- **50% of grade based on exams using this quantity:**
 $\max(\text{midterm} > 0 ? \text{final} : 0, \frac{1}{2}(\text{midterm} + \text{final}))$
- **50% of grade from projects**
 - For each project, 50% of score based on passing test cases
 - Remaining 50% based on design and style
- **Most people's projects pass most test cases**
 - Please, please, please turn in working code, or **no credit** here
- **Means design and style matter a lot**
 - Large software systems not just about producing working code
 - Need to produce code other people can understand
 - That's why we have group projects

Style

- **Must turn in a design document along with code**
 - We supply you with templates for each project's design doc
- **CAs will manually inspect code for correctness**
 - E.g., must actually implement the design
 - Must handle corner cases (e.g., handle `malloc` failure)
- **Will deduct points for error-prone code w/o errors**
 - Don't use global variables if automatic ones suffice
 - Don't use deceptive names for variables
- **Code must be easy to read**
 - Indent code, keep lines and (when possible) functions short
 - Use a uniform coding style (try to match existing code)
 - Put comments on structure members, globals, functions
 - Don't leave in reams of commented-out garbage code

Assignment requirements

- **Do not look at other people's solutions to projects**
 - We reserve the right to run **MOSS** on present and past submissions
 - Do not publish your own solutions in violation of the **honor code**
 - **That means using (public) github can get you in big trouble**
- **You may read but not copy other OSES**
 - E.g., Linux, OpenBSD/FreeBSD, etc.
- **Cite any code that inspired your code**
 - As long as you cite what you used, it's not cheating
 - In worst case, we deduct points if it undermines the assignment
- **Projects due Fridays (before section—TBD)**
- **Ask cs140-staff for extension if you run into trouble**
 - Be sure to tell us: How much have you done? How much is left?
When can you finish by?

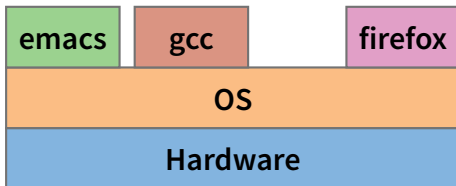
Outline

1 Administrivia

2 Substance

What is an operating system?

- Layer between applications and hardware



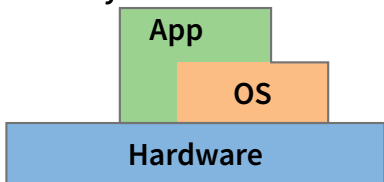
- Makes hardware useful to the programmer
- [Usually] Provides abstractions for applications
 - Manages and hides details of hardware
 - Accesses hardware through low/level interfaces unavailable to applications
- [Often] Provides protection
 - Prevents one process/user from clobbering another

Why study operating systems?

- **Operating systems are a mature 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
- **Still open questions in operating systems**
 - Security – Hard to achieve security without a solid foundation
 - Scalability – How to adapt concepts when hardware scales $10\times$ (fast networks, low service times, high core counts, big data...)
- **High-performance servers are an OS issue**
 - Face many of the same issues as OSES, sometimes bypass OS
- **Resource consumption is an OS issue**
 - Battery life, radio spectrum, etc.
- **New “smart” devices need new OSES**

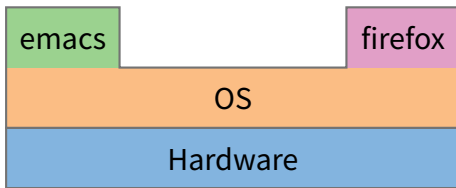
Primitive Operating Systems

- Just a library of standard services [no protection]



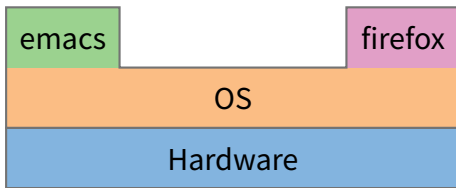
- Standard interface above hardware-specific drivers, etc.
- **Simplifying assumptions**
 - System runs one program at a time
 - No bad users or programs (often bad assumption)
- **Problem: Poor utilization**
 - ...of hardware (e.g., CPU idle while waiting for disk)
 - ...of human user (must wait for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

Multitasking



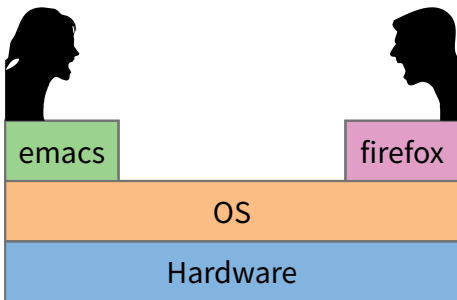
- **Idea: More than one process can be running at once**
 - When one process blocks (waiting for disk, network, user input, etc.) run another process
- **Problem: What can ill-behaved process do?**

Multitasking



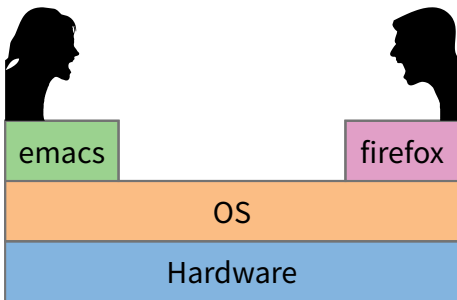
- **Idea: More than one process can be running at once**
 - When one process blocks (waiting for disk, network, user input, etc.) run another process
- **Problem: What can ill-behaved process do?**
 - Go into infinite loop and never relinquish CPU
 - Scribble over other processes' memory to make them 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/apps
- **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

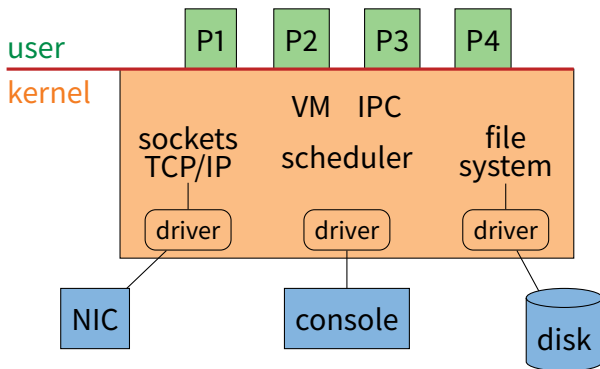


- Many OSeS use *protection* to serve distrustful users/apps
- Idea: With N users, system not N times slower
 - Users' demands for CPU, memory, etc. are bursty
 - Win by giving resources to users who actually need them
- What can go wrong?
 - Users are gluttons, use too much CPU, etc. (need policies)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

Protection

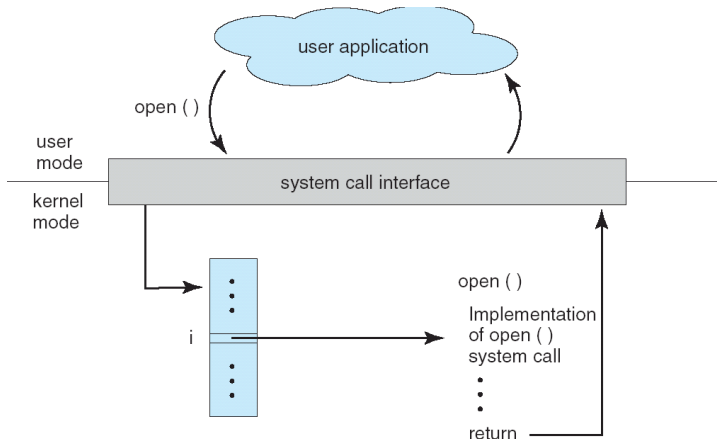
- **Mechanisms that isolate bad programs and people**
- **Pre-emption:**
 - Give application a resource, take it away if needed elsewhere
- **Interposition/mediation:**
 - Place OS between application and “stuff”
 - Track all pieces that application allowed to use (e.g., in table)
 - On every access,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

Typical OS structure



- **Most software runs as user-level processes (P[1-4])**
 - process \approx instance of a program
- **OS *kernel* runs in *privileged* mode (orange)**
 - Creates/deletes processes
 - Provides access to hardware

System calls

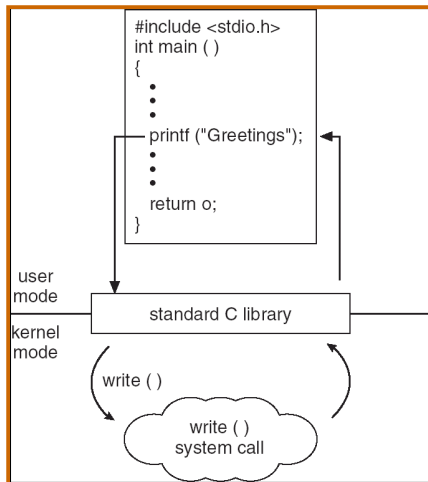


- Applications can invoke kernel through *system calls*
 - Special instruction transfers control to kernel
 - ...which dispatches to one of few hundred syscall handlers

System calls (continued)

- **Goal: Do things application 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`, `fgets`, 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

UNIX file system calls

- **Applications “open” files (or devices) by name**
 - I/O happens through open files
- `int open(char *path, int flags, /*int mode*/...);`
 - flags: `O_RDONLY`, `O_WRONLY`, `O_RDWR`
 - `O_CREAT`: create the file if non-existent
 - `O_EXCL`: (w. `O_CREAT`) create if file exists already
 - `O_TRUNC`: Truncate the file
 - `O_APPEND`: Start writing from end of file
 - mode: final argument with `O_CREAT`
- **Returns file descriptor—used for all I/O to file**

Error returns

- What if `open` fails? Returns -1 (invalid fd)
- Most system calls return -1 on failure
 - Specific kind of error in global int `errno`
- `#include <sys/errno.h>` for possible values
 - 2 = `ENOENT` “No such file or directory”
 - 13 = `EACCES` “Permission Denied”
- `perror` function prints human-readable message
 - `perror ("initfile");`
→ “initfile: No such file or directory”

Operations on file descriptors

- `int read (int fd, void *buf, int nbytes);`
 - Returns number of bytes read
 - Returns 0 bytes at end of file, or -1 on error
- `int write (int fd, const void *buf, int nbytes);`
 - Returns number of bytes written, -1 on error
- `off_t lseek (int fd, off_t pos, int whence);`
 - whence: 0 – start, 1 – current, 2 – end
 - ▷ Returns previous file offset, or -1 on error
- `int close (int fd);`

File descriptor numbers

- **File descriptors are inherited by processes**
 - When one process spawns another, same fds by default
- **Descriptors 0, 1, and 2 have special meaning**
 - 0 – “standard input” (`stdin` in ANSI C)
 - 1 – “standard output” (`stdout`, `printf` in ANSI C)
 - 2 – “standard error” (`stderr`, `perror` in ANSI C)
 - Normally all three attached to terminal
- **Example:** `type.c`
 - Prints the contents of a file to `stdout`

```
void
typefile (char *filename)
{
    int fd, nread;
    char buf[1024];

    fd = open (filename, O_RDONLY);
    if (fd == -1) {
        perror (filename);
        return;
    }

    while ((nread = read (fd, buf, sizeof (buf))) > 0)
        write (1, buf, nread);

    close (fd);
}
```

- Can see system calls using strace utility (ktrace on BSD)

Protection example: 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
- For many years, could wedge most 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

Address translation

- **Protect memory 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 lets OS disable (invalidate) particular virtual addresses**
 - Catch and halt buggy program that makes wild accesses
 - Make virtual memory seem bigger than physical (e.g., bring a page in from disk only when accessed)
- **CPU enforced 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

Different system contexts

- A CPU (core) is at any point in one of several contexts
- *User-level* – CPU in user mode running application
- **Kernel process context**
 - Running kernel code on behalf of a particular process
 - E.g., performing system call
 - Also exception (memory fault, numeric exception, etc.)
 - Or executing a kernel-only process (e.g., network file server)
- **Kernel code not associated with a process**
 - Timer interrupt (hardclock)
 - Device interrupt
 - “Softirqs”, “Tasklets” (Linux-specific terms)
- **Context switch code – change which process is running**
 - Requires changing the current address space
- **Idle – nothing to do (might powerdown CPU)**

Transitions between contexts

- User → kernel process context: syscall, page fault
- User/process context → interrupt handler: hardware
- Process context → user/context switch: return
- Process context → context switch: sleep
- Context switch → user/process context

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**
 - 1 GB memory in machine
 - 2 Processes want to run, each use 1 GB
 - 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 \approx 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