What is a shell?

An interpreter

- Runs programs on behalf of the user
- Allows programmer to create/manage set of programs
  - sh — Original Unix shell (Bourne, 1977)
  - csh — BSD Unix C shell (tcsh enhances it)
  - bash — "Bourne again" shell
- Every command typed in the shell starts a child process of the shell
- Runs at user-level. Uses syscalls: fork, exec, etc.

The Unix shell (simplified)

```
while(! EOF)
read input
handle regular expressions
int pid = fork()   // create child
if (pid == 0) { // child here
  exec("program", argc, argv0,...);
}
else {  // parent here
...
}
```

Signals (Virtualized Interrupts)

Just a taste...

Asynchronous notifications in user space

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., CTRL-C from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated
20	SIGSTP	Stop until SIGCONT	Stop signal from terminal (e.g., CTRL-Z from keyboard)

Sending a Signal

- Kernel delivers a signal to a destination process, for a variety of reasons
  - kernel detected a system event (e.g., division by zero (SIGFPE) or termination of a child (SIGCHLD) or...
  - a process invoked the kill systems call requesting kernel to send another process a signal
    - debugging
    - suspension
    - resumption
    - timer expiration

Receiving a Signal

- Each signal prompts one of these default actions
  - terminate the process
  - ignore the signal
  - terminate the process and dump core
  - stop the process
  - continue process if stopped
- Signal can be caught by executing a user-level function called signal handler
  - similar to exception handler invoked in response to an asynchronous interrupt
- Process can also be suspended waiting for a signal to be caught (synchronously)

Signal Example

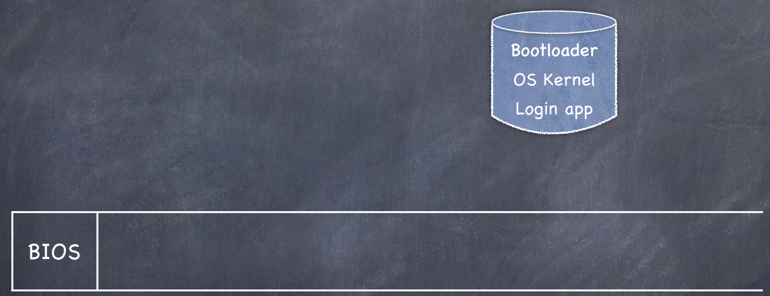
```
int main() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++) // N forks
        if ((pid[i] = fork()) == 0) {
            while(1); // child infinite loop
        }
    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {      // parent continues executing
        printf("Killing proc. %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status)) // parent checks for each child's exit - normal exit returns 1
            printf("Child %d terminated w/exit status %d\n", wpid,
                   WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
    exit(0);
}
```

Handler Example

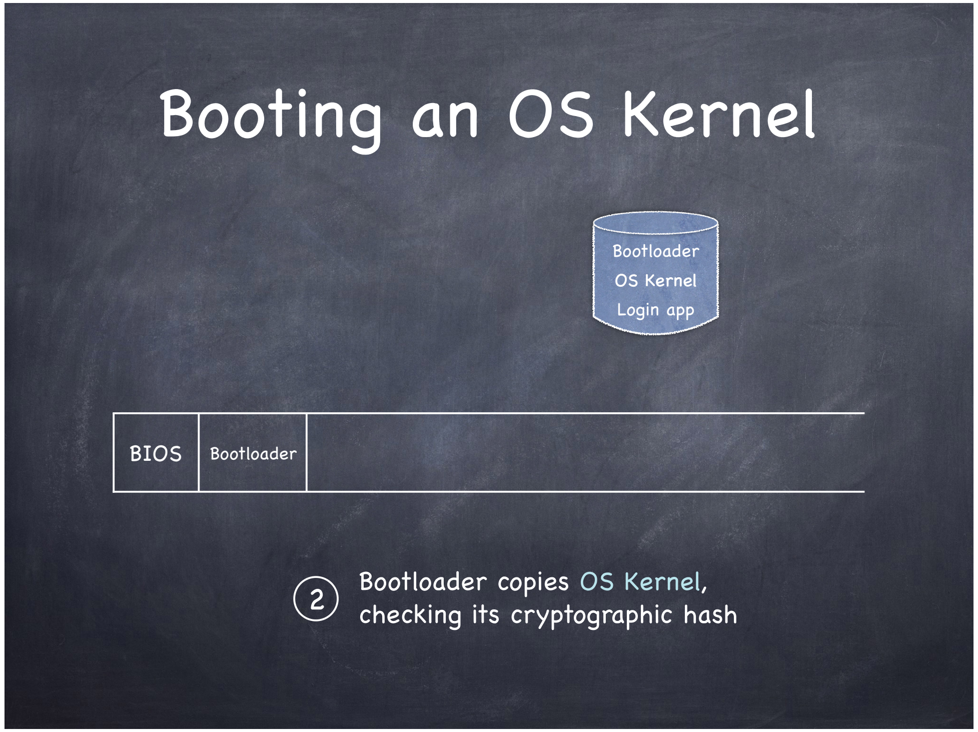
```
void int_handler(int sig) {
    printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}
int main() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler) // register handler for SIGINT
    for (i = 0; i < N; i++) // N forks
        if ((pid[i] = fork()) == 0) {
            while(1); // child infinite loop
        }
    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {      // parent continues executing
        printf("Killing proc. %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status)) // parent checks for each child's exit
            printf("Child %d terminated w/exit status %d\n", wpid,
                   WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
    exit(0);
}
```

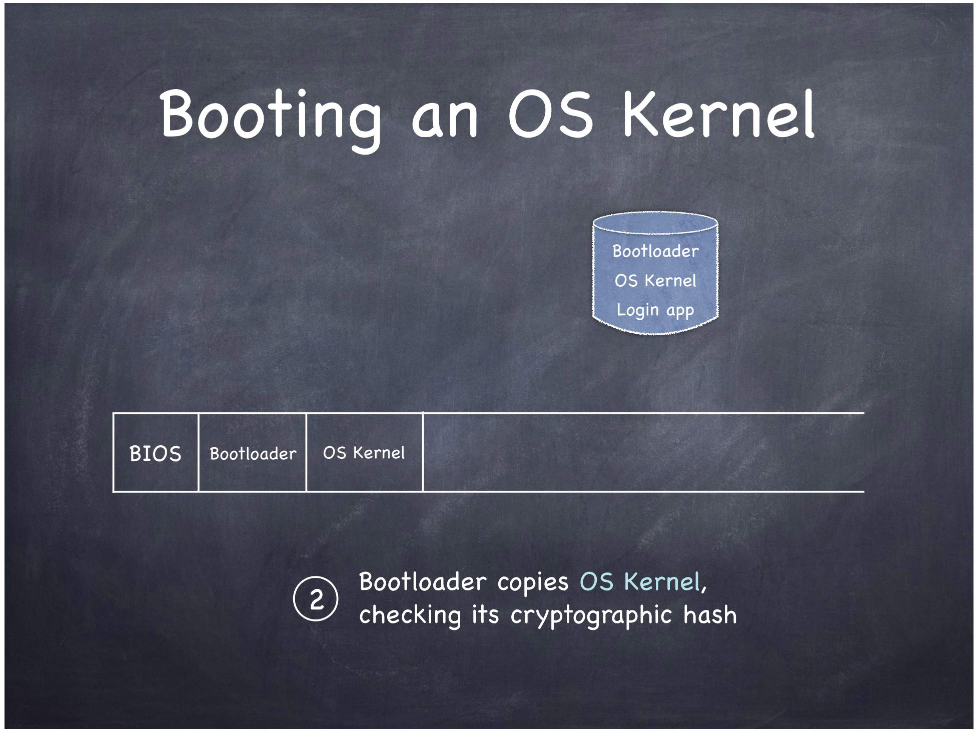
Booting an OS Kernel

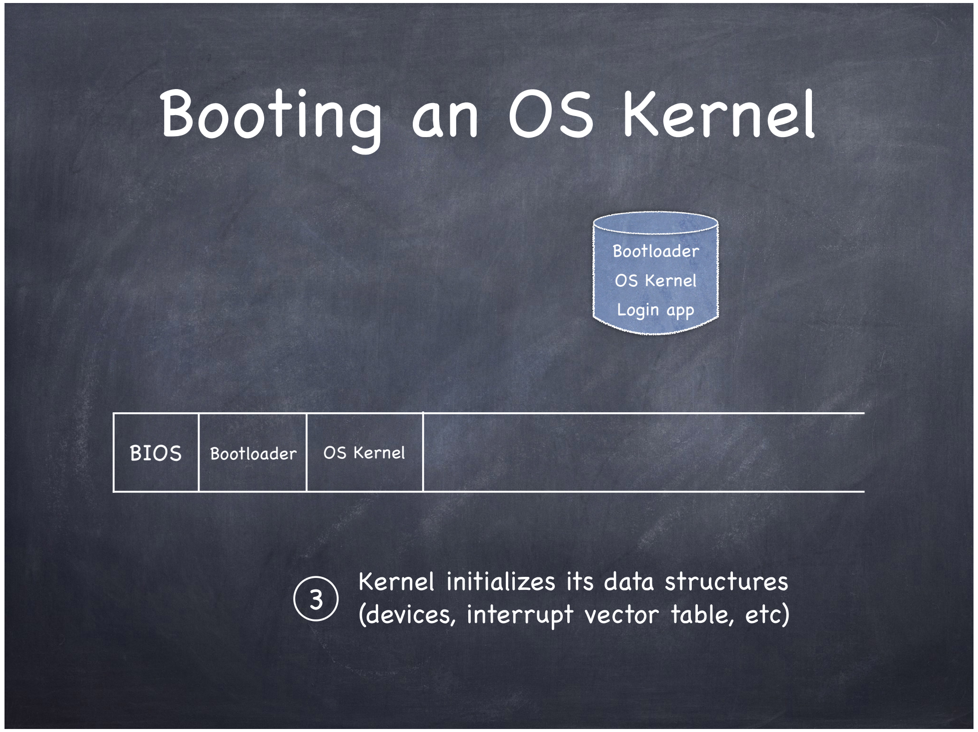


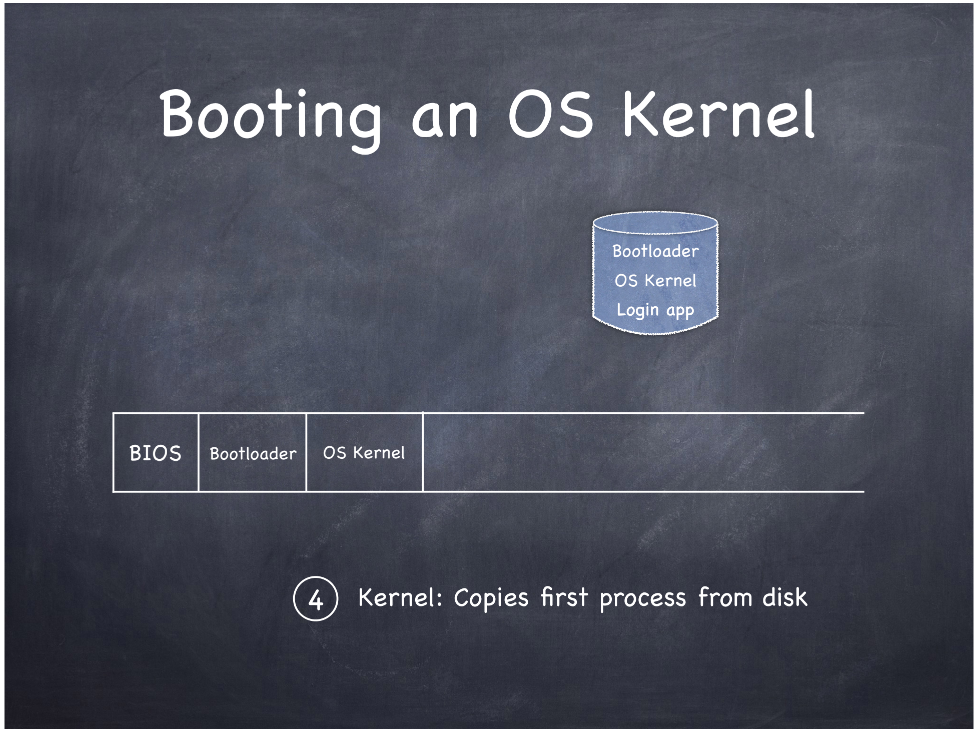
- Basic Input/Output System
  - In ROM; includes the first instructions fetched and executed

(1) BIOS copies Bootloader, checking its cryptographic hash to make sure it has not been tampered with

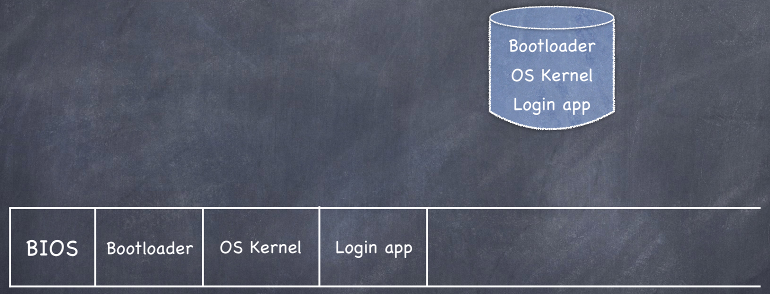

Booting an OS Kernel
Bootloader
OS Kernel
Login app
BIOS
Bootloader
2
Bootloader copies OS Kernel,
checking its cryptographic hash


Booting an OS Kernel
Bootloader
OS Kernel
Login app
BIOS
Bootloader
OS Kernel
2
Bootloader copies OS Kernel,
checking its cryptographic hash


Booting an OS Kernel
Bootloader
OS Kernel
Login app
BIOS
Bootloader
OS Kernel
3
Kernel initializes its data structures
(devices, interrupt vector table, etc)


Booting an OS Kernel
Bootloader
OS Kernel
Login app
BIOS
Bootloader
OS Kernel
4
Kernel: Copies first process from disk

Booting an OS Kernel



(4) Kernel: Copies first process from disk
Changes PC and sets mode bit to 1

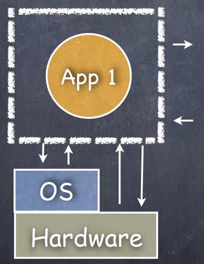
And the dance begins!

Threads

An abstraction for concurrency

(Chapters 25-27)

Rethinking the Process Abstraction



- Processes serve two key purposes:
  - defines the granularity at which the OS offers isolation
    - address space identifies what can be touched by the program
  - define the granularity at which the OS offers scheduling and can express concurrency
    - a stream of instructions executed sequentially

Threads: a New Abstraction for Concurrency

- A single-execution stream of instructions that represents a separately schedulable task
  - OS can run, suspend, resume a thread at any time
  - bound to a process (lives in an address space)
  - Finite Progress Axiom: execution proceeds at some unspecified, non-zero speed
- Virtualizes the processor
  - programs run on machine with a seemingly infinite number of processors
- Allows to specify tasks that should be run concurrently...
  - ...and lets us code each task sequentially

How Threads Can Help

for (k = 0; k < n; k++)
  a[k] = b[k] × c[k] + d[k] × e[k]

- Consider a Web server
  - get network message from client
  - get URL data from disk
  - compose response
  - send response

Overlapping I/O & Computation

Request 1
Thread 1

- get network message (URL) from client
- get URL from disk

(disk access latency)

- send data over network

Request 2
Thread 2

- get network message (URL) from client
- get URL from disk

(disk access latency)

- send data over network

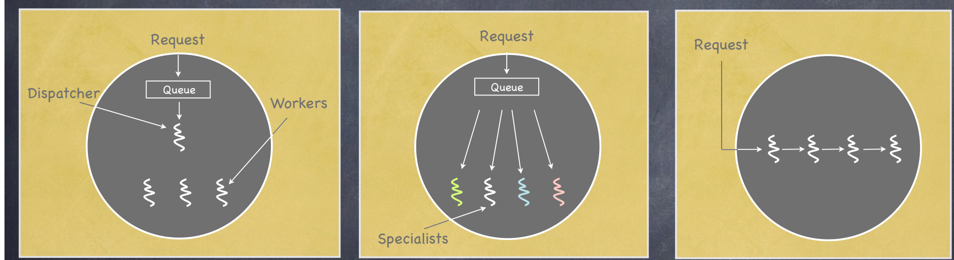
Time

Total time is less than Request 1 + Request 2

Why Threads?

- To express a natural program structure
  - updating the screen, fetching new data, receiving user input — different tasks within the same address space
- To exploit multiple processors
  - different threads may be mapped to distinct processors
- To maintain responsiveness
  - splitting commands, spawn threads to do work in the background
- Masking long latency of I/O devices
  - do useful work while waiting

Multithreaded Processing Paradigms



Dispatcher/Workers   Specialists   Pipeline

All You Need is Love
(and a stack)

- All threads within a process share
  - heap
  - global/static data
  - libraries
- Each thread has separate
  - program counter
  - registers
  - stack

A simple API

Syscall	Description
void thread_create (thread, func, arg)	Creates a new thread in thread, which will execute function func with arguments arg.
void thread_yield()	Calling thread gives up processor. Scheduler can resume running this thread at any time
int thread_join (thread)	Wait for thread to finish, then return the value thread passed to thread_exit. May be called only once for each thread.
void thread_exit (ret)	Finish caller; store ret in caller's TCB and wake up any thread that invoked thread_join(caller).

Preemption

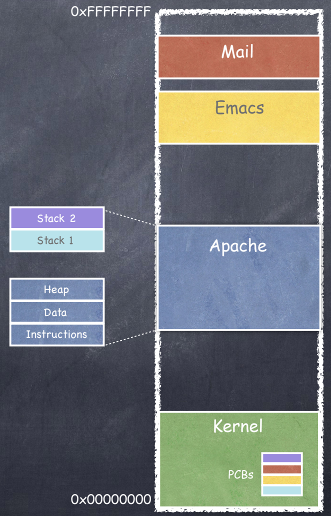
- Preemptive
  - yield automatically upon clock interrupts
  - true of most modern threading systems
- Non-preemptive
  - explicitly yield to pass control to other threads
  - true of CS4411 P1 project

One Abstraction, Two Implementations

- Kernel Threads
  - each thread has its own PCB in the kernel
  - PCBs of threads mapped to the same process point to the same physical memory
  - visible (and schedulable) by kernel
- User Threads
  - one PCB for the process
  - each thread has its own Thread Control Block (TCB) [implemented in the host process' heap]
  - implemented entirely in user space; invisible to the kernel

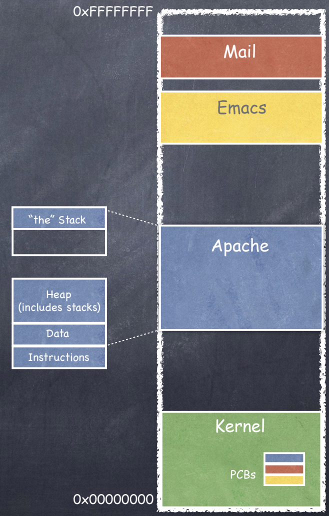
Kernel-level Threads

- Kernel knows about threads existence, and schedules them as it does processes
- Each thread has a separate PCB
- PCBs of threads mapped in the same process have
  - same address space
    - page table base register
  - different PC, SP, registers, interrupt stack



User-level Threads

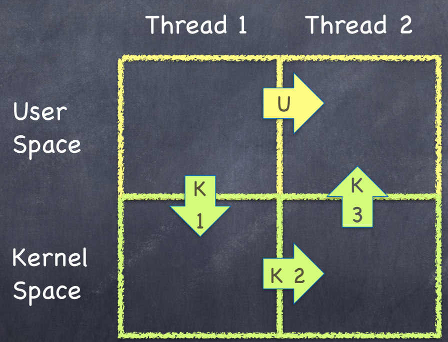
- Run OS-like code in user space
  - real OS is unaware of threads
    - holds a single PCB for all user threads within the same process
  - each thread has associated a Thread Control Block (TCB) kept by process in user space
- User-level threads incur lower overhead than kernel-level threads...
- ...but kernel level threads simplify system call handling and scheduling



Kernel- vs. User-level Threads

	Kernel-level Threads	User-Level Threads
Ease of implementation	Easy to implement: just like process, but with shared address space	Requires implementing user-level schedule and context switches
Handling system calls	Thread can run blocking systems call concurrently	Blocking system call blocks all threads: needs OS support for non-blocking system calls (scheduler activations)
Cost of context switch	Thread requires three context switches	Thread switch efficiently implemented in user space

Kernel- vs. User-level Thread Switching



Threads considered harmful

- Creating a thread or process for each unit of work (e.g., user request) is dangerous
  - High overhead to create & delete thread/process
  - Can exhaust CPU & memory resource
- Thread/process pool controls resource use
  - Allows service to be well conditioned
    - output rate scales to input rate
    - excessive demand does not degrade pipeline throughput