# CSE 120
# Operating Systems Principles

## Spring 2025

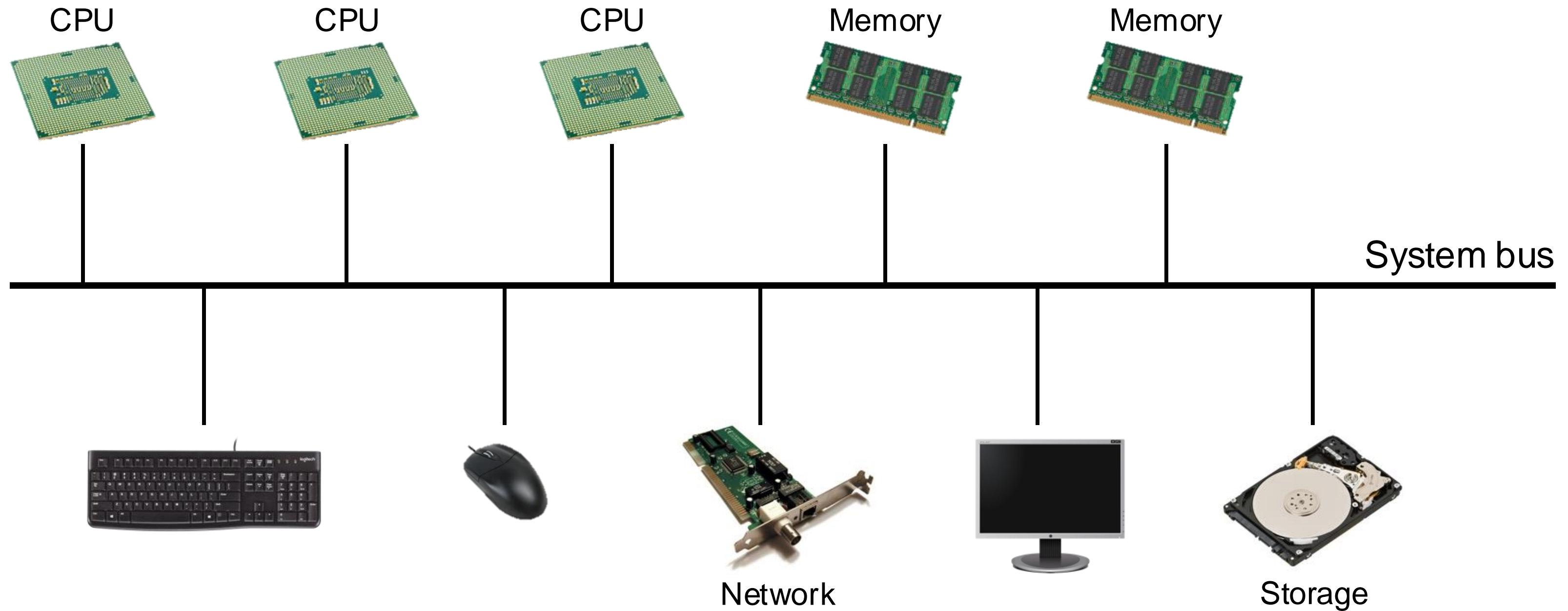
## Lecture 2: Interactions with Apps and Hardware

Amy Ousterhout

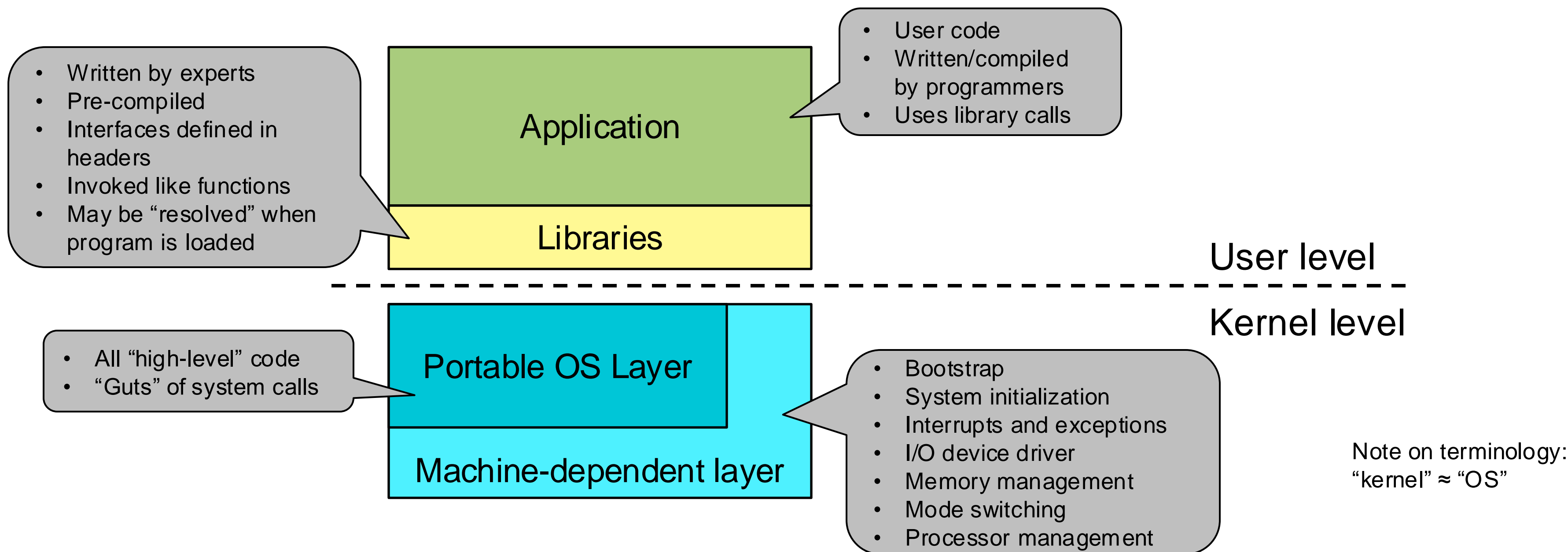
# What is an Operating System?

- Code that sits between applications and hardware
- Provides abstractions to layers above
- Implements abstractions for and manages resources below

# Hardware of a Typical Computer



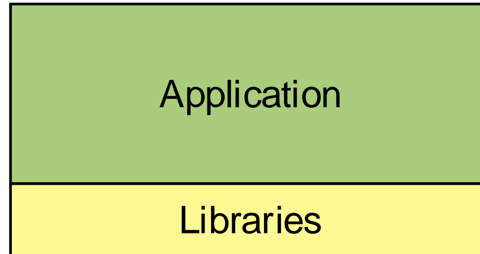
# Software of a Typical (Unix) System




Note on terminology:
“kernel” ≈ “OS”

# Questions for Today

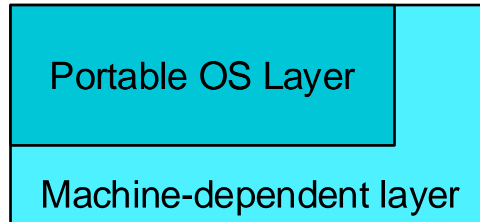
How do we separate the OS layer from apps (and libraries)?




User level

Kernel level

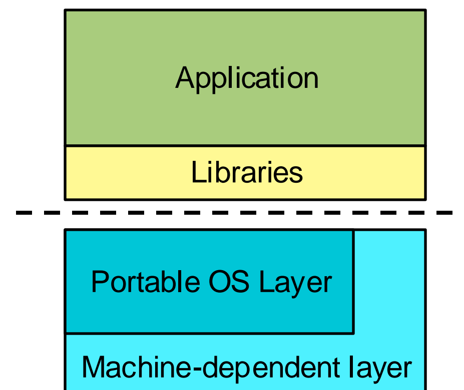
How do we cross between these layers when necessary?




With support from the hardware!

# Today's Outline

- Protection: how can the OS perform special tasks and protect itself from applications?
  - Privileged instructions
  - Memory protection
- Interacting with the OS: when/how does the OS run?
  - Faults
  - System calls
  - Interrupts



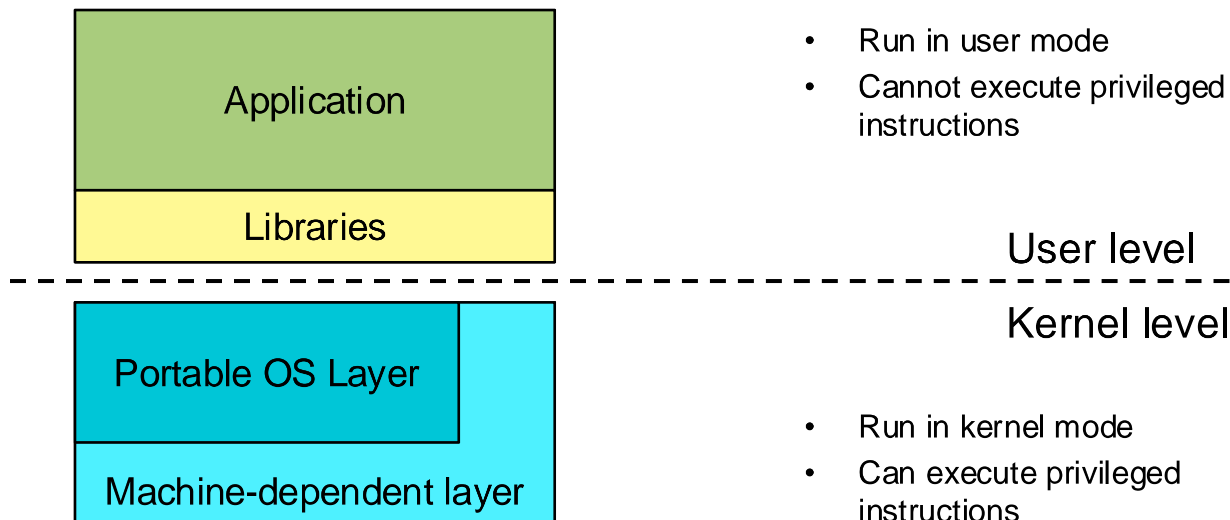
# Dual-Mode Operation

- How can the OS perform special tasks (e.g., manage resources)?
- How can the OS protect itself from applications and protect applications from each other?
- OS needs to be “privileged”
- Every CPU core can run in one of two modes:
  - Kernel mode – can run *all* instructions
  - User mode – can only run *non-privileged* instructions
  - Mode is indicated by a mode bit in a protected CPU control register

# Privileged Instructions

- Privileged instructions: a subset of instructions that can only run in kernel mode
  - CPU checks mode bit when privileged instructions execute
  - Attempts to execute in user mode are detected and prevented by the CPU
- Privileged instructions can:
  - Directly access I/O devices (disk, network, etc.)
    - » For security, fairness
  - Manipulate memory-management state (page table pointers, etc.)
    - » Prevent apps from accessing other apps' memory (or the OS's memory)
  - Manipulate protected control registers (e.g., mode bit)
    - » Prevent apps from giving themselves privileges!

# Software of a Typical (Unix) System




- Run in user mode
- Cannot execute privileged instructions

- Run in kernel mode
- Can execute privileged instructions

# Example of a Privileged Instruction

- HLT: halts the CPU

INSTRUCTION SET REFERENCE, A-L

### HLT—Halt

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| F4 | HLT | ZO | Valid | Valid | Halt |

**Instruction Operand Encoding**

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| ZO | NA | NA | NA | NA |

### Description

Stops instruction execution and places the processor in a HALT state. An enabled interrupt (including NMI and SMI), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

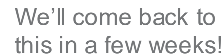
When a HLT instruction is executed on an Intel 64 or IA-32 processor supporting Intel Hyper-Threading Technology, only the logical processor that executes the instruction is halted. The other logical processors in the physical processor remain active, unless they are each individually halted by executing a HLT instruction.

The HLT instruction is a privileged instruction. When the processor is running in protected or virtual-8086 mode, the privilege level of a program or procedure must be 0 to execute the HLT instruction.

# (Live Demo of HLT)

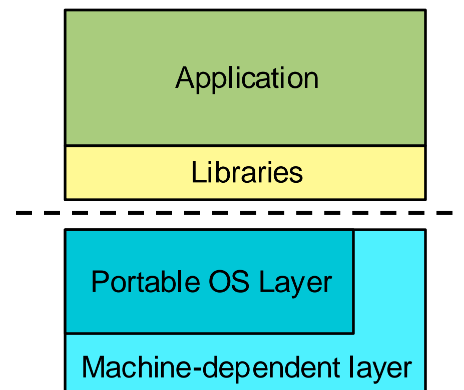
# Memory Protection

- OS must protect itself from user programs
- OS must be able to protect programs from each other
- May or may not protect user programs from the OS
  - Should programs trust the OS?
- Memory-management hardware provides memory protection
  - Page table pointers, page protection, segmentation, TLB
- Manipulating memory-management hardware uses privileged instructions



# Today’s Outline

- Protection: how can the OS perform special tasks and protect itself from applications?
  - Privileged instructions
  - Memory protection
- Interacting with the OS: when/how does the OS run?
  - Faults
  - System calls
  - Interrupts



# Events

- An event is an unnatural change in control flow
  - Immediately stop the current execution
  - Changes mode, context (machine state), or both
- The OS defines a handler for each event type
  - Specific types of events are defined by the machine
  - Event handlers execute in kernel mode
- After the system is booted, all entry to the kernel occurs as the result of an event
  - In effect, the operating system is one big event handler
  - OS only executes in reaction to events

# Types of Events

- Two main types of events: exceptions and interrupts
- Interrupts – caused by an external event
  - Device finishes I/O, timer expires, etc.
  - Analogy: receiving a phone call or text message
- Exceptions – caused by program executing instructions
  - Executing a privileged instruction (fault)
  - Requesting services from the operating system (system calls)
  - Also called “traps”
- Events can be unexpected or deliberate

# Faults

- Hardware detects and reports exceptional conditions
  - Divide by zero, page faults
- Upon exception, hardware faults (verb)
  - Must save state (PC, registers, mode, etc.) so that the faulting process can be restarted
  - Each exception type has an associated number
  - CPU finds the exception handler for that number
  - Switch to kernel mode and start executing the exception handler
- When done, operating system returns to program
  - Reverses the steps above
- Could we prevent faults with software?

Mode change:
- Common across faults, system calls, and interrupts
- Handled by hardware

# Handling Faults (Recovery)

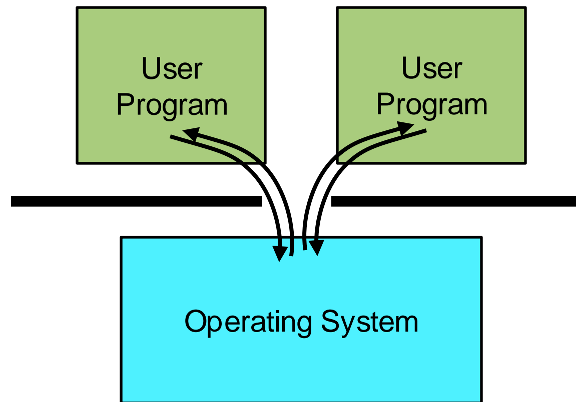
- Some faults are handled by "fixing" the exceptional condition
  - Page faults cause the OS to bring missing pages into memory
  - Fault handler returns to program and re-executes the instruction that cause the page fault
- Some faults are handled by notifying the process
  - Applications can register a fault handler with the OS
  - OS fault handler will return to the user-mode handler
  - Example: Unix signals such as SIGFPE, SIGTERM, SIGSEGV

# Handling Faults (Termination)

- OS may handle unrecoverable faults by killing the user process
  - E.g., program fault with no registered handler
  - Halt process, write process state to a file, destroy process
- What about faults in the kernel?
  - E.g., 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 locks up

# System Calls

- For a user application to do something “privileged” it must call an OS procedure
- System calls = operating system API
  - Interface between an application and the operating system
- System call categories
  - Process management
  - Memory management
  - File management
  - Device management
  - Communication



# System Call Mechanism

- CPUs provide a system call instruction that:
  - Causes an exception, which vectors to a kernel handler
  - Passes a parameter determining the system routine to call (which system call)
  - Saves caller state (PC, registers, mode) so it can be restored
  - Returning from system call restores this state
- Requires hardware support to:
  - Restore saved state, reset mode, resume execution

# Examples of System Call Instructions

- INT: executes a syscall

**INT *n*/INTO/INT3/INT1—Call to Interrupt Procedure**

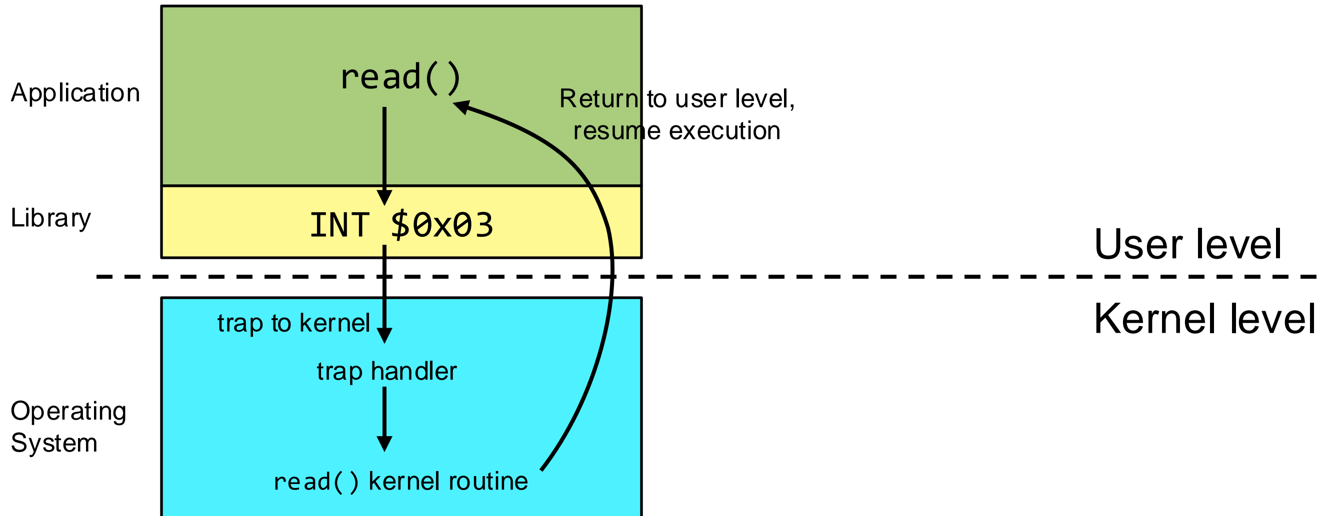
| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| CC | INT3 | ZO | Valid | Valid | Generate breakpoint trap. |
| CD *ib* | INT *imm8* | I | Valid | Valid | Generate software interrupt with vector specified by immediate byte. |
| CE | INTO | ZO | Invalid | Valid | Generate overflow trap if overflow flag is 1. |
| F1 | INT1 | ZO | Valid | Valid | Generate debug trap. |

- SYSCALL: executes a syscall on newer 64-bit CPUs

**SYSCALL—Fast System Call**

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| 0F 05 | SYSCALL | ZO | Valid | Invalid | Fast call to privilege level 0 system procedures. |

# System Call Example



# Assigning Numbers to System Calls

- Who defines which number corresponds to which system call?

## LINUX System Call Quick Reference

*Jialong He*
Jialong_he@bigfoot.com
http://www.bigfoot.com/~jialong_he

### Introduction

System call is the services provided by Linux kernel. In C programming, it often uses functions defined in **libc** which provides a wrapper for many system calls. Manual page section 2 provides more information about system calls. To get an overview, use "man 2 intro" in a command shell.

It is also possible to invoke **syscall()** function directly. Each system call has a function number defined in **<syscall.h>** or **<unistd.h>**. Internally, system call is invokded by software interrupt 0x80 to transfer control to the kernel. System call table is defined in Linux kernel source file "**arch/i386/kernel/entry.S**".

### System Call Example

```
#include <syscall.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

int main(void) {

        long ID1, ID2;
        /*-----------------------------*/
        /* direct system call          */
        /* SYS_getpid (func no. is 20) */
        /*-----------------------------*/
        ID1 = syscall(SYS_getpid);
        printf ("syscall(SYS_getpid)=%ld\n", ID1);

        /*-----------------------------*/
        /* "libc" wrapped system call  */
        /* SYS_getpid (Func No. is 20) */
        /*-----------------------------*/
        ID2 = getpid();
        printf ("getpid()=%ld\n", ID2);

        return(0);
}
```

### System Call Quick Reference

| No | Func Name | Description | Source |
|---|---|---|---|
| 1 | exit | terminate the current process | *kernel/exit.c* |
| 2 | fork | create a child process | *arch/i386/kernel/process.c* |
| 3 | read | read from a file descriptor | *fs/read_write.c* |
| 4 | write | write to a file descriptor | *fs/read_write.c* |
| 8 | creat | create a file or device ("man 2 open" for information) | *fs/open.c* |
| 9 | link | make a new name for a file | *fs/namei.c* |
| 10 | unlink | delete a name and possibly the file it refers to | *fs/namei.c* |
| 11 | execve | execute program | *arch/i386/kernel/process.c* |
| 12 | chdir | change working directory | *fs/open.c* |
| 13 | time | get time in seconds | *kernel/time.c* |
| 14 | mknod | create a special or ordinary file | *fs/namei.c* |
| 15 | chmod | change permissions of a file | *fs/open.c* |
| 16 | lchown | change ownership of a file | *fs/open.c* |
| 18 | stat | get file status | *fs/stat.c* |
| 19 | lseek | reposition read/write file offset | *fs/read_write.c* |
| 20 | getpid | get process identification | *kernel/sched.c* |
| 21 | mount | mount filesystems | *fs/super.c* |
| 22 | umount | unmount filesystems | *fs/super.c* |
| 23 | setuid | set real user ID | *kernel/sys.c* |
| 24 | getuid | get real user ID | *kernel/sched.c* |
| 25 | stime | set system time and date | *kernel/time.c* |
| 26 | ptrace | allows a parent process to control the execution of a child process | *arch/i386/kernel/ptrace.c* |
| 27 | alarm | set an alarm clock for delivery of a signal | *kernel/sched.c* |
| 28 | fstat | get file status | *fs/stat.c* |
| 29 | pause | suspend process until signal | *arch/i386/kernel/sys_i386.c* |
| 30 | utime | set file access and modification times | *fs/open.c* |
| 33 | access | check user's permissions for a file | *fs/open.c* |
| 34 | nice | change process priority | *kernel/sched.c* |
| 36 | sync | update the super block | *fs/buffer.c* |
| 37 | kill | send signal to a process | *kernel/signal.c* |
| 38 | rename | change the name or location of a file | *fs/namei.c* |
| 39 | mkdir | create a directory | *fs/namei.c* |
| 40 | rmdir | remove a directory | *fs/namei.c* |
| 41 | dup | duplicate an open file descriptor | *fs/fcntl.c* |
| 42 | pipe | create an interprocess channel | *arch/i386/kernel/sys_i386.c* |
| 43 | times | get process times | *kernel/sys.c* |
| 45 | brk | change the amount of space allocated for the calling process's data segment | *mm/mmap.c* |
| 46 | setgid | set real group ID | *kernel/sys.c* |

# Referencing Data

- Processes and the OS are in different address spaces
  - How can the OS return references to kernel data structures?
- Use names instead of pointers
  - E.g., integer object handles or descriptors such as Unix file descriptors

# Interrupts

- Interrupts signal external events
- Interrupts are generated by hardware
  - I/O hardware interrupts
  - Timers
- Interrupts on modern CPUs are precise
  - CPU transfers control only on instruction boundaries

# Handling Interrupts

- Interrupt handler is in the kernel
- Steps:
  - Disable interrupts at lower priorities
  - Save state
  - Transfer control to the interrupt service routine (in the kernel)
  - When done, re-enable interrupts
  - Resume the user-level program at the next instruction

# Example of an Interrupt: Timer

- The timer is critical for an operating system
- Fallback mechanism by which the OS reclaims control over the machine
  - Timer is set to generate an interrupt after a period of time
    - » Setting the timer is a privileged instruction
  - Handled by the kernel, which decides what program to run next
    - » Basis for the OS scheduler (more later…)
- Prevents programs from hogging the CPU
  - OS can always regain control from erroneous or malicious programs
  - E.g., regain control during an infinite loop
- Also used for time-based functions (e.g., `sleep`)

# Example of an Interrupt: I/O

- Asynchronous I/O
  - OS initiates I/O
  - Device (e.g., disk) operates independently of the rest of the machine
  - Device sends an interrupt to CPU when done
  - CPU context switches to the interrupt handler
  - Eventually resumes the original process

# x86 Interrupts and Exceptions (1)

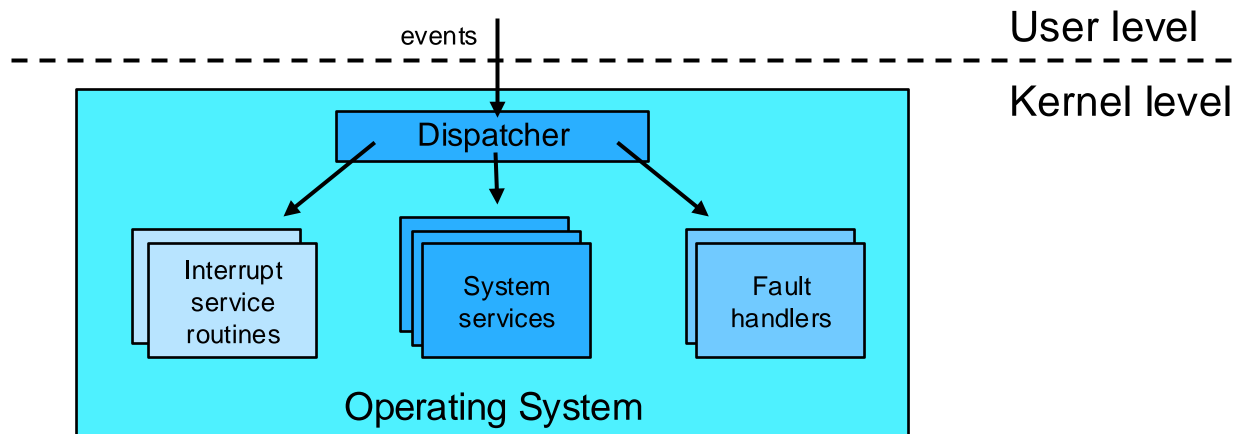
| Vector # | Mnemonic | Description | Type |
| --- | --- | --- | --- |
| 0 | #DE | Divide error (by zero) | Fault |
| 1 | #DB | Debug | Fault/trap |
| 2 | | Non-Maskable interrupt | Interrupt |
| 3 | #BP | Breakpoint | Trap |
| 4 | #OF | Overflow | Trap |
| 5 | #BR | BOUND range exceeded | Trap |
| 6 | #UD | Invalid opcode | Fault |
| 7 | #NM | Device not available | Fault |
| 8 | #DF | Double fault | Abort |
| 9 | | Coprocessor segment overrun | Fault |
| 10 | #TS | Invalid TSS | |

# x86 Interrupts and Exceptions (2)

| Vector # | Mnemonic | Description | Type |
|---|---|---|---|
| 11 | #NP | Segment not present | Fault |
| 12 | #SS | Stack-segment fault | Fault |
| 13 | #GP | General protection | Fault |
| 14 | #PF | Page fault | Fault |
| 15 | | Reserved | Fault |
| 16 | #MF | Floating-point error (math fault) | Fault |
| 17 | #AC | Alignment check | Fault |
| 18 | #MC | Machine check | Abort |
| 19-31 | | Reserved | |
| 32-255 | | User defined | Interrupt |

# The OS as a giant interrupt handler

- Once the system is booted, all entry to the kernel occurs due to interrupts, system calls, or faults
  - Timer interrupts
  - I/O interrupts
  - Faults
  - System calls



# (Live Demo of Divide by Zero)

# Practice Question - Mode Switches

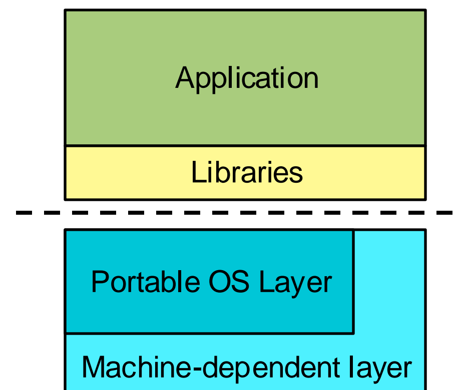
- How many times does this program switch from user space to kernel space, after the call to `main`?

```
void signal_handler(int signum, siginfo_t *si, void *context)
{
        /* was this an integer divide-by-zero exception? */
        if (si->si_signo == SIGFPE && si->si_code == FPE_INTDIV) {
                printf("oh no, we tried to divide by zero!\n");
        } else {
                printf("unexpected signal %d\n", si->si_signo);
        }

        /* let's exit */
        exit(-1);
}
```

```
int main(int argc, char *argv[])
{
        struct sigaction sigact;
        int x = 0;

        /* let's register our signal handler with the OS */
        sigact.sa_sigaction = signal_handler;
        sigact.sa_flags = SA_SIGINFO | SA_RESTART;
        if (sigaction(SIGFPE, &sigact, NULL) != 0) {
                printf("sigaction returned error %d\n", errno);
        }

        /* now let's try to divide by zero */
        int y = 1/x;

        printf("divided by zero!\n");
        return 0;
}
```

# Summary

- Protection: how can the OS perform special tasks and protect itself from applications?
  - Privileged instructions
  - Memory protection
- Interacting with the OS: when/how does the OS run?
  - Faults
  - System calls
  - Interrupts



# Upcoming Tasks

- Read chapters 3-5
- Discussion tomorrow: ieng6, Nachos, and project 0
- Project 0
  - Due Tuesday 4/8 at 11:59 pm, done individually
  - See updated instructions: run “cs120sp25” (instead of running “prep cs120sp25”)
- Project groups
  - Google form to collect group members (see Piazza)
  - Just need one submission per group, fill it out even if you are working alone
  - Due Tuesday 4/8 at 11:59 pm
- Homework 1
  - Due Tuesday 4/15 at 11:59 pm