COS 318: Operating Systems

OS Structures and System Calls

Kai Li and Andy Bavier
Computer Science Department
Princeton University

http://www.cs.princeton.edu/courses/archive/fall13/cos318/

Logistics

- Weekly TA office hours posted on Piazza
  - May change from week to week
- Four Lab TAs available over the weekends (Fri – Sun)
  - David Durst
  - Anna Simpson
  - Catherine Wu
  - Harvest Zhang

Baby Steps

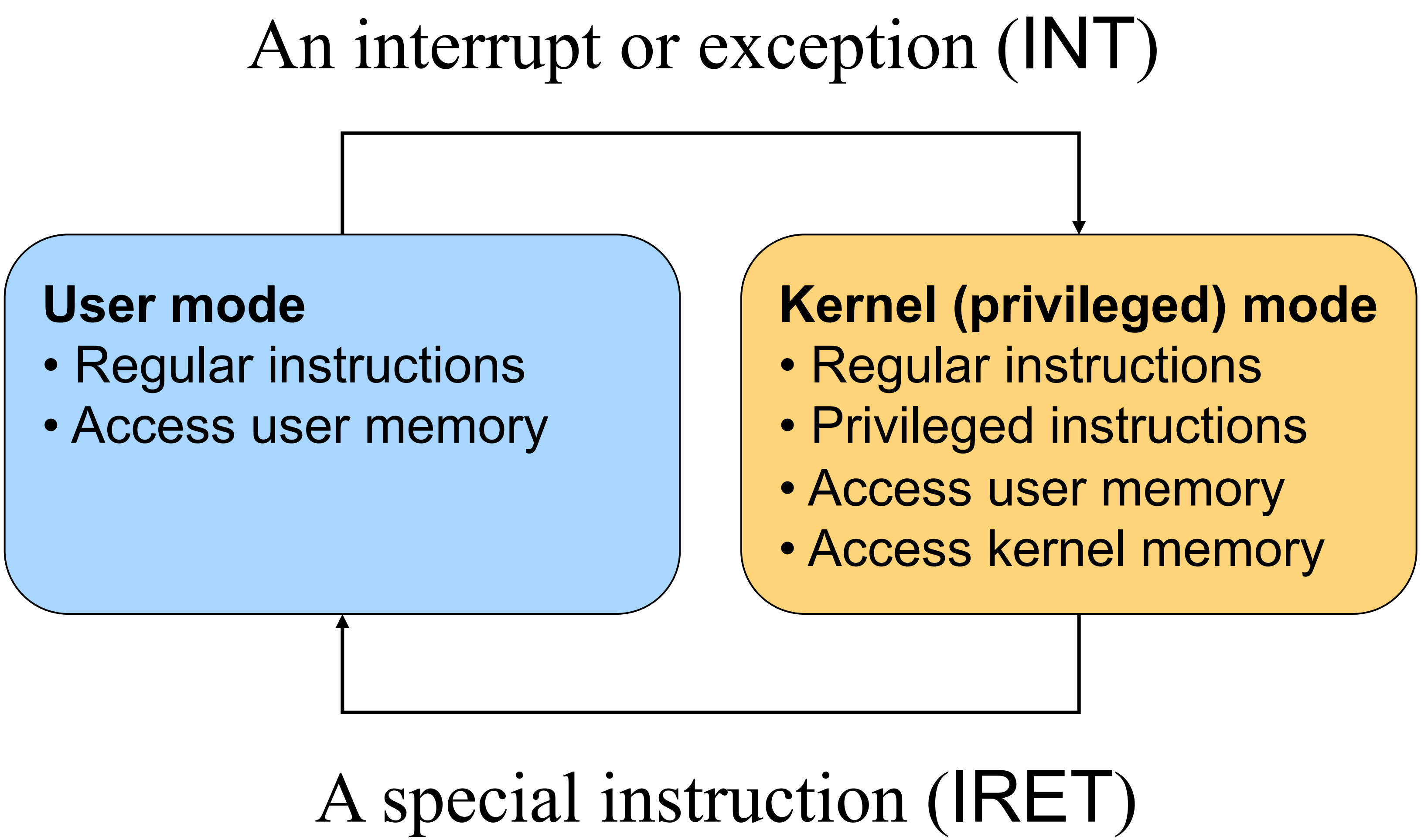
Outline

- Protection mechanisms
- OS structures
- System and library calls

Protection Issues

- CPU
  - Kernel has the ability to take CPU away from users to prevent a user from using the CPU forever
  - Users should not have such an ability
- Memory
  - Prevent a user from accessing others' data
  - Prevent users from modifying kernel code and data structures
- I/O
  - Prevent users from performing "illegal" I/Os
- Question
  - What's the difference between protection and security?

Architecture Support: Privileged Mode

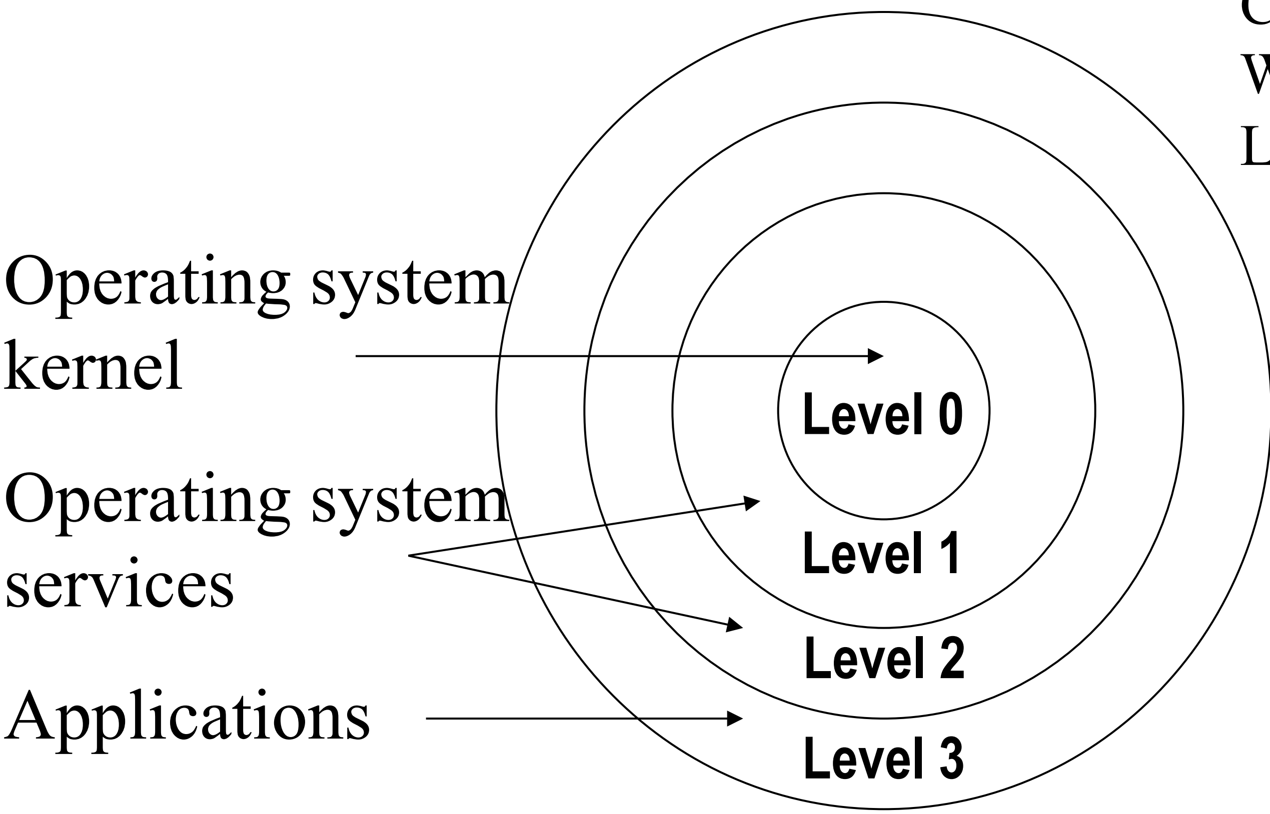


Privileged Instruction Examples

- Memory address mapping
- Flush or invalidate data cache
- Invalidate TLB entries
- Load and read system registers
- Change processor modes from kernel to user
- Change the voltage and frequency of processor
- Halt a processor
- Reset a processor
- Perform I/O operations

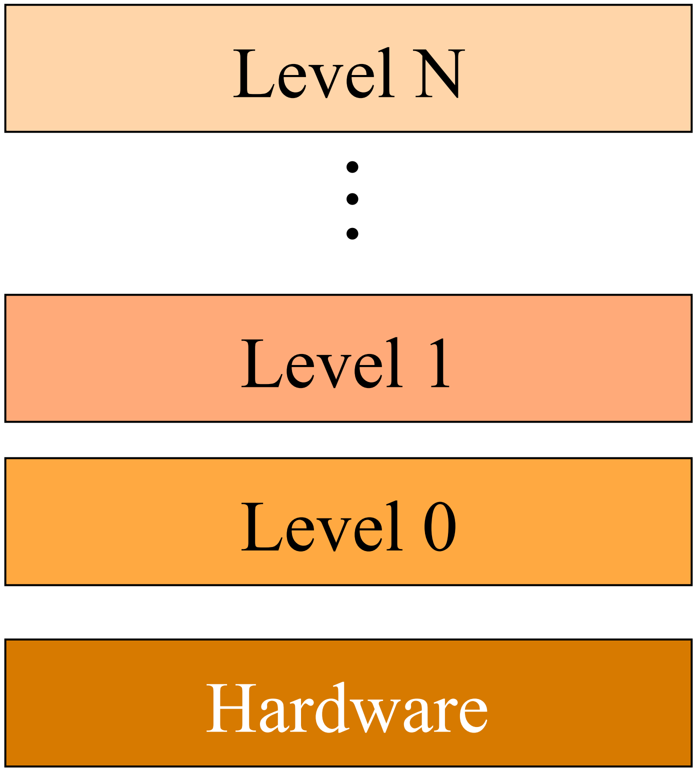
x86 Protection Rings

Privileged instructions
Can be executed only
When current privileged
Level (CPR) is 0



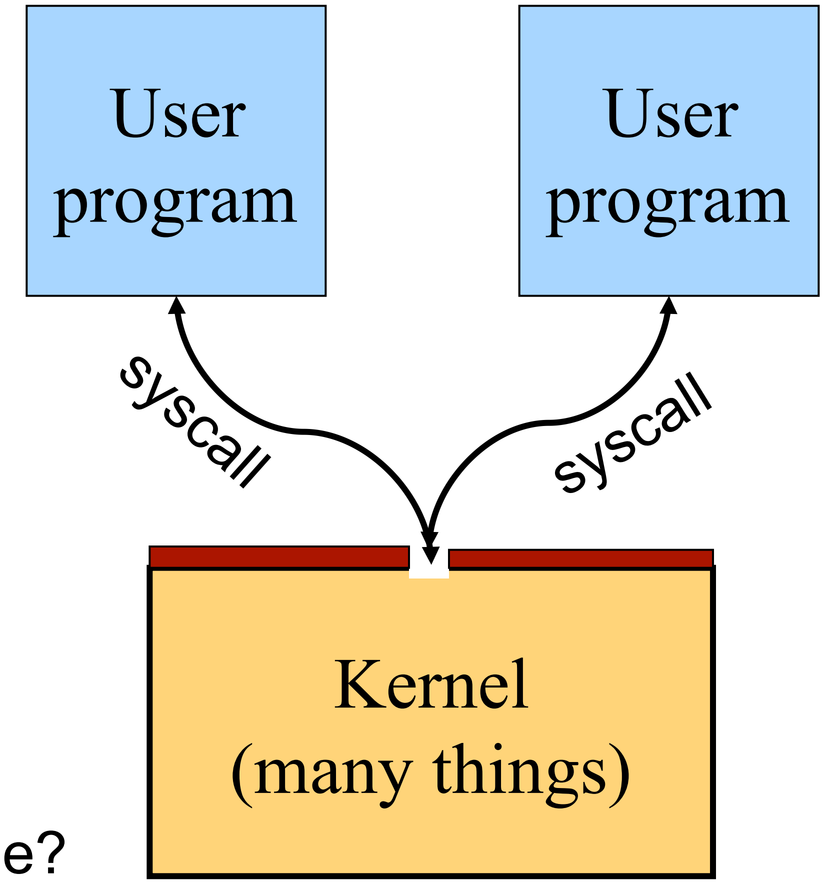
Layered OS Structure

- Hiding information at each layer
- Layered dependency
- Examples:
  - THE (6 layers)
  - MS-DOS (4 layers)
  - MULTICS (8 layers)
- Pros
  - Layered abstraction
  - Separation of concerns
- Cons
  - Inefficient
  - Inflexible



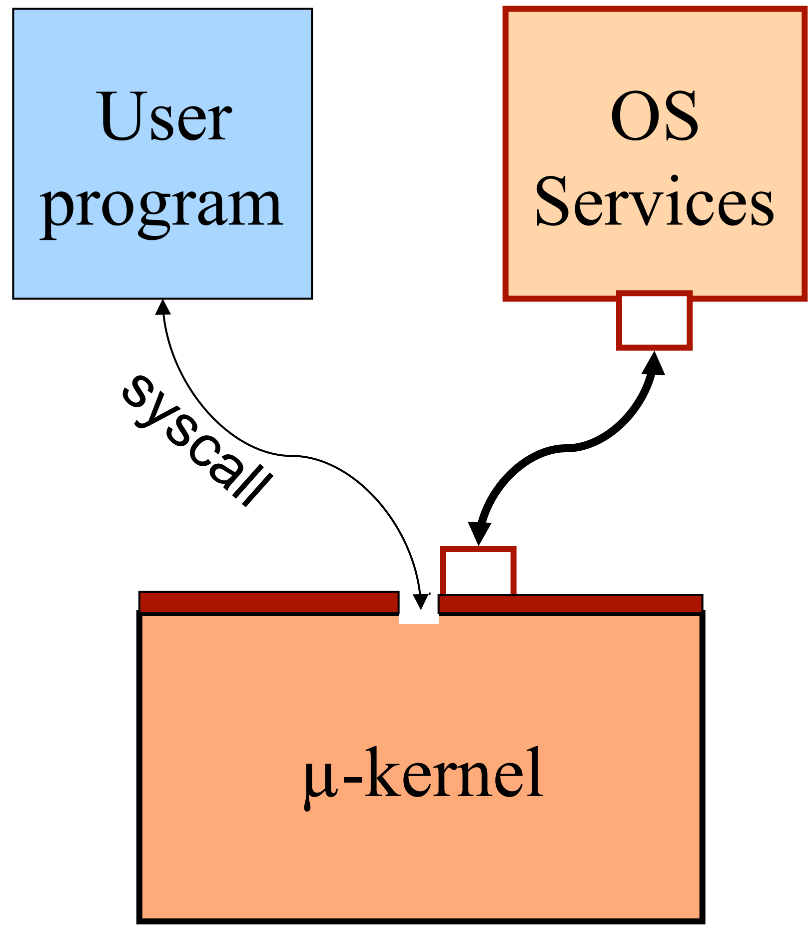
Monolithic OS

- All kernel routines are together, any can call any
- A system call interface
- Examples:
  - Linux, BSD Unix, Windows
- Pros
  - Shared kernel space
  - Good performance
- Cons
  - No information hiding
  - Instability
  - How many bugs in 5M lines of code?



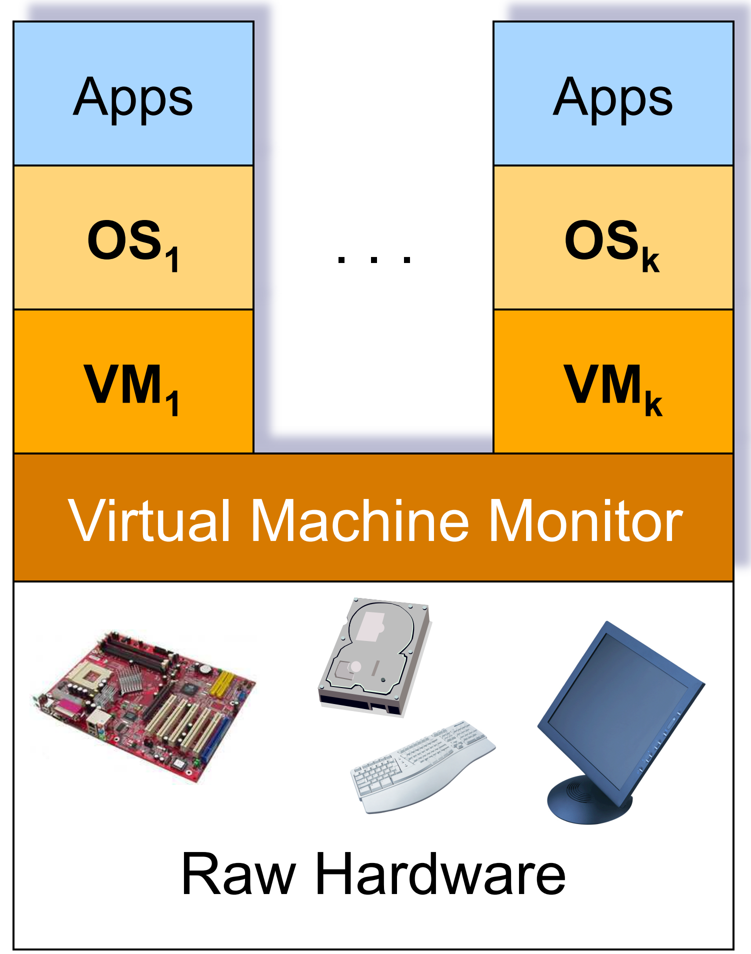
Microkernel

- Put less in kernel mode; only small part of OS
- Services are implemented as regular process
- *µ*-kernel gets svcs on for users by messaging with service processes
- Examples:
  - Mach, Taos, L4, OS-X
- Pros?
  - Modularity
  - Fault isolation
- Cons?
  - Inefficient (lots of boundary crossings)

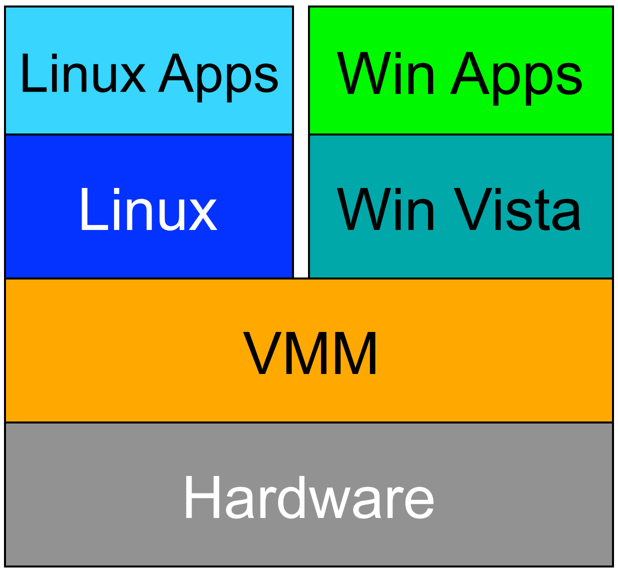


Virtual Machine

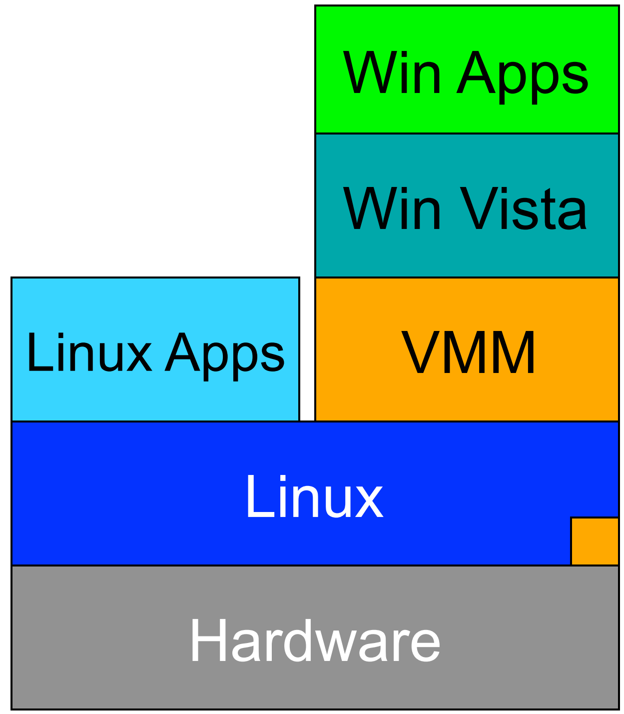
- Separate multiprogramming from abstraction; VMM provides former
- Virtual machine monitor
  - Virtualize hardware, but expose as multiple instances of "raw" HW
  - Run several OSes, one on each instance
  - Examples
    - IBM VM/370
    - Java VM
    - VMWare, Xen
- What would you use a virtual machine for?



Two Popular Ways to Implement VMM



VMM runs on hardware



VMM as an application

(A special lecture later in the semester)

Interlude

"UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity."

-- Dennis Ritchie

Outline

- Protection mechanisms
- OS structures
- System and library calls

System Calls

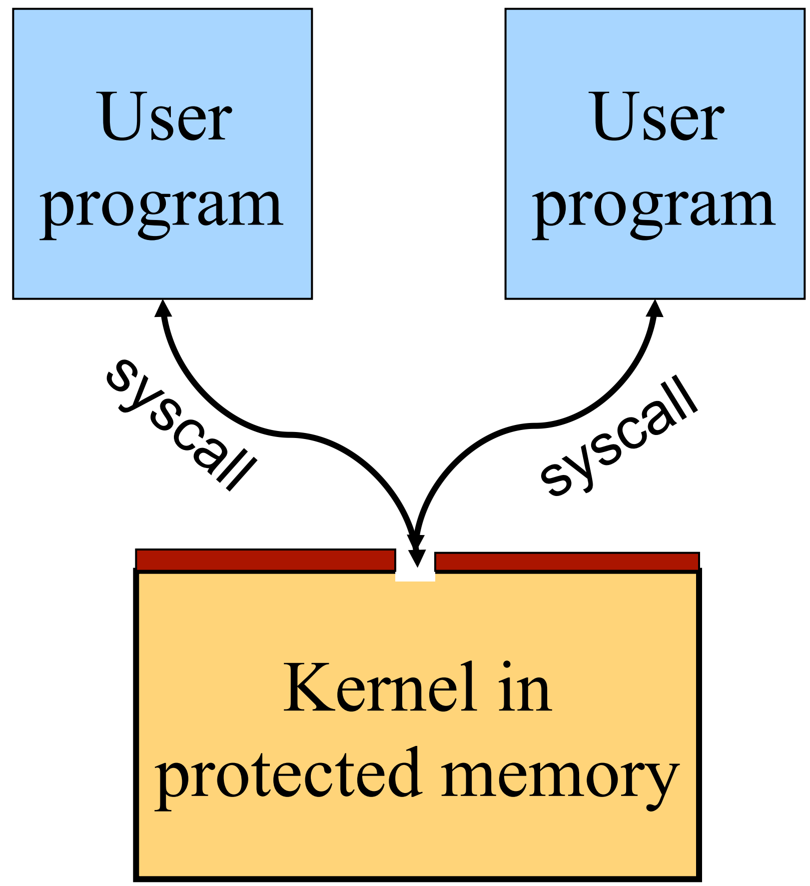
- Operating system API
  - Interface between an application and the operating system kernel
- Categories
  - Process management
  - Memory management
  - File management
  - Device management
  - Communication

How many system calls?

6th Edition Unix:	~45
POSIX:	~130
FreeBSD:	~500
Linux:	~300
Windows:	400? 1000? 1M?

System Call Mechanism

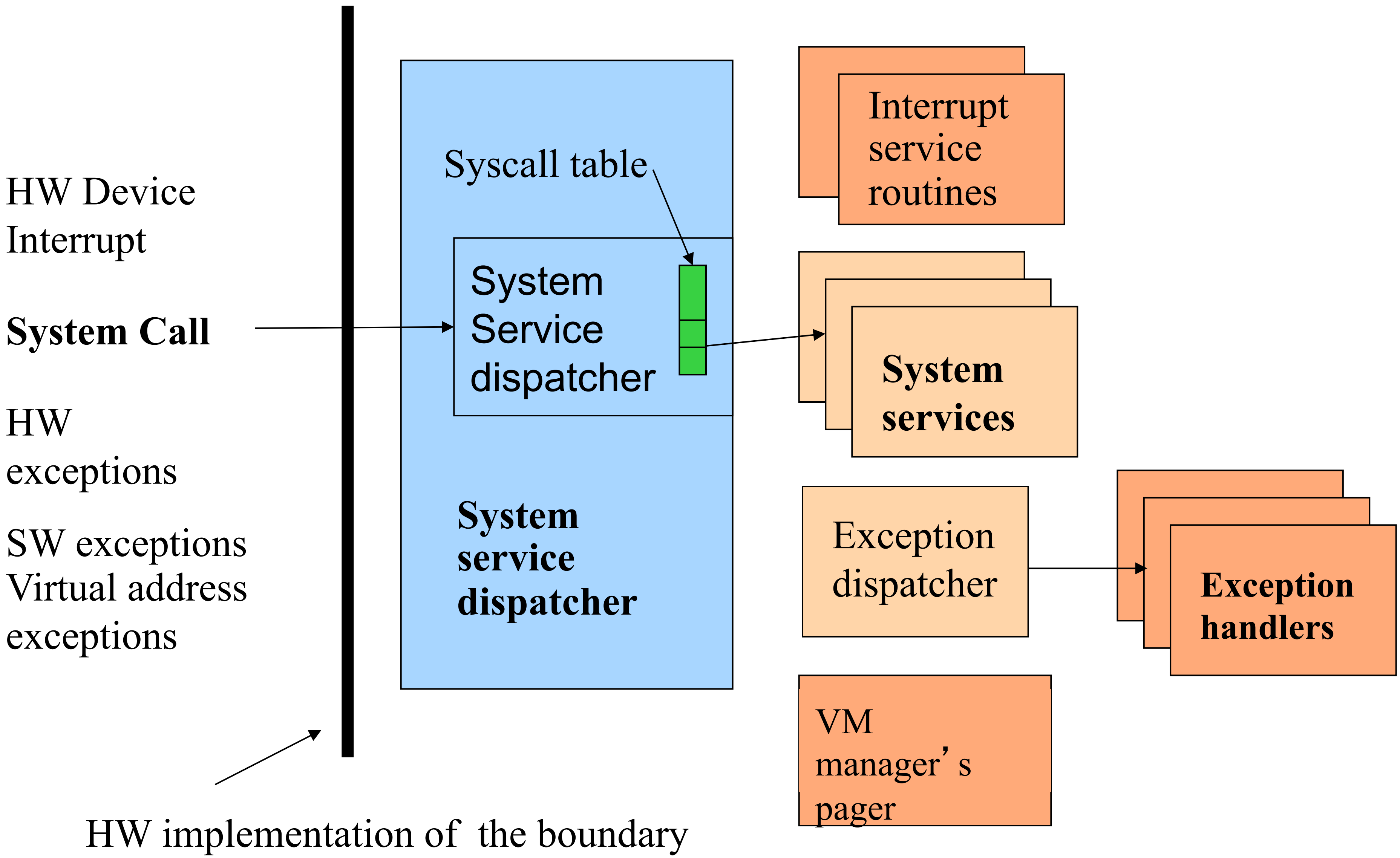
- Assumptions
  - User code can be arbitrary
  - User code cannot modify kernel memory
- Design Issues
  - User makes a system call with parameters
  - The call mechanism switches code to kernel mode
  - Execute system call
  - Return with results



Passing Parameters

- Pass by registers
  - # of registers
  - # of usable registers
  - # of parameters in system call
  - Spill/fill code in compiler
- Pass by a memory vector (list)
  - Single register for starting address
  - Vector in user’s memory
- Pass by stack
  - Similar to the memory vector
  - Procedure call convention

OS Kernel: Trap Handler



From http://minnie.tuhs.org/UnixTree/V6

V6/usr/sys/ken/sysent.c

Find at most [5 ▾] related files. [Search]

☐ including files from this version of Unix.

```
#
/*
 */

/*
 * This table is the switch used to transfer
 * to the appropriate routine for processing a system call.
 * Each row contains the number of arguments expected
 * and a pointer to the routine.
 */
int     sysent[]
{
        0, &nullsys,                    /*  0 = indir */
        0, &rexit,                      /*  1 = exit */
        0, &fork,                       /*  2 = fork */
        2, &read,                       /*  3 = read */
        2, &write,                      /*  4 = write */
        2, &open,                       /*  5 = open */
        0, &close,                      /*  6 = close */
        0, &wait,                       /*  7 = wait */
        2, &creat,                      /*  8 = creat */
        2, &link,                       /*  9 = link */
        1, &unlink,                     /* 10 = unlink */
        2, &exec,                       /* 11 = exec */
        1, &chdir,                      /* 12 = chdir */
        0, &gtime,                      /* 13 = time */
        3, &mknod,                      /* 14 = mknod */
        2, &chmod,                      /* 15 = chmod */
        2, &chown,                      /* 16 = chown */
        1, &sbreak,                     /* 17 = break */
        2, &stat,                       /* 18 = stat */
        2, &seek,                       /* 19 = seek */
        0, &getpid,                     /* 20 = getpid */
        3, &smount,                     /* 21 = mount */
        1, &sumount,                    /* 22 = umount */
        0, &setuid,                     /* 23 = setuid */
        0, &getuid,                     /* 24 = getuid */
        0, &stime,                      /* 25 = stime */
        3, &ptrace,                     /* 26 = ptrace */
        0, &nosys,                      /* 27 = x */
        1, &fstat,                      /* 28 = fstat */
        0, &nosys,                      /* 29 = x */
        1, &nullsys,                    /* 30 = smdate; inoperative */
        1, &stty,                       /* 31 = stty */
        1, &gtty,                       /* 32 = gtty */
        0, &nosys,                      /* 33 = x */
        0, &nice,                       /* 34 = nice */
        0, &sslep,                      /* 35 = sleep */
        0, &sync,                       /* 36 = sync */
        1, &kill,                       /* 37 = kill */
        0, &getswit,                    /* 38 = switch */
        0, &nosys,                      /* 39 = x */
        0, &nosys,                      /* 40 = x */
        0, &dup,                        /* 41 = dup */
        0, &pipe,                       /* 42 = pipe */
        1, &times,                      /* 43 = times */
        4, &profil,                     /* 44 = prof */
        0, &nosys,                      /* 45 = tiu */
        0, &setgid,                     /* 46 = setgid */
        0, &getgid,                     /* 47 = getgid */
        2, &ssig,                       /* 48 = sig */
```

Library Stubs for System Calls

- Example:

int read(int fd, char * buf, int size)

{

move fd, buf, size to R_1, R_2, R_3

move READ to R_0

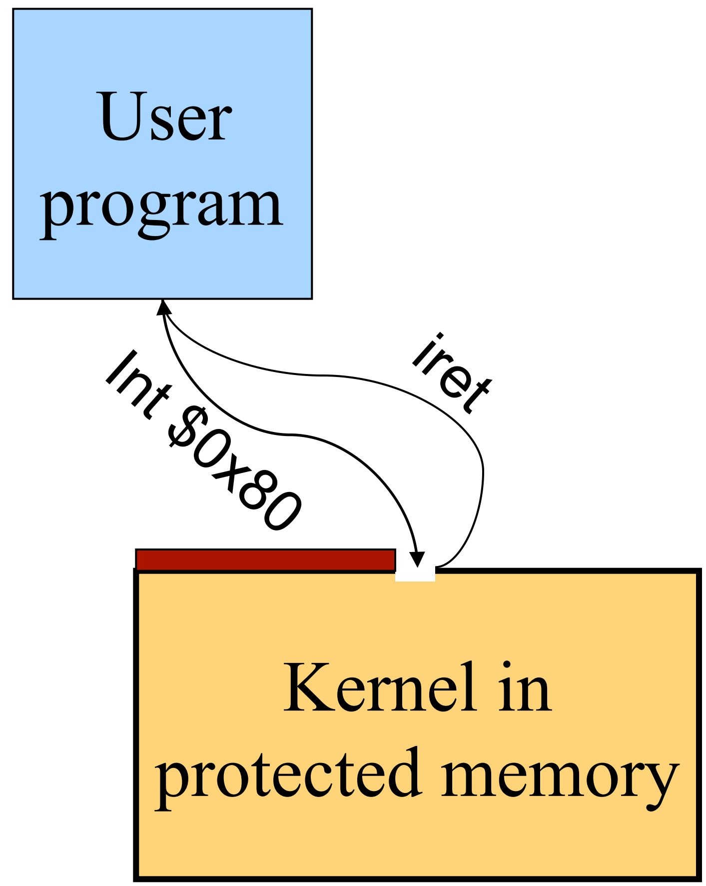
int $0x80

move result from R_{result}

}

Linux: 80

NT: 2E



System Call Entry Point

EntryPoint:

save context

switch to kernel stack

check R_0

call the real code pointed by R_0

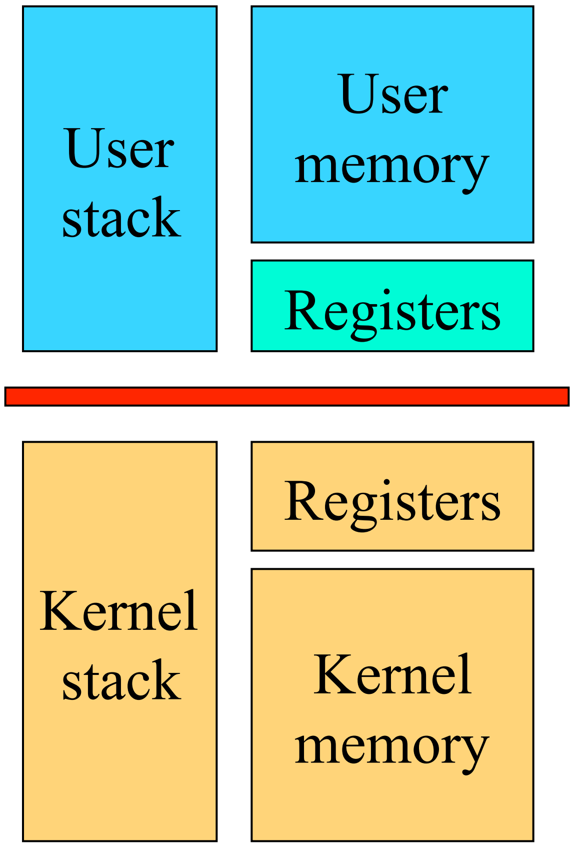
place result in R_{result}

switch to user stack

restore context

iret (change to user mode and return)

(Assume passing parameters in registers)



Design Issues

- System calls
  - There is one result register; what about more results?
  - How do we pass errors back to the caller?
- System calls vs. library calls
  - What should go in system calls?
  - What should go in library calls?

Syscall or library?

```
/*
 * open system call
 */
open()
{
        register *ip;
        extern uchar;

        ip = namei(&uchar, 0);
        if(ip == NULL)
                return;
        u.u_arg[1]++;
        open1(ip, u.u_arg[1], 0);
}

/*
 * creat system call
 */
creat()
{
        register *ip;
        extern uchar;

        ip = namei(&uchar, 1);
        if(ip == NULL) {
                if(u.u_error)
                        return;
                ip = maknode(u.u_arg[1]&07777&(~ISVTX));
                if (ip==NULL)
                        return;
                open1(ip, FWRITE, 2);
        } else
                open1(ip, FWRITE, 1);
}
```

```
/*
 * common code for open and creat.
 * Check permissions, allocate an open file structure,
 * and call the device open routine if any.
 */
open1(ip, mode, trf)
int *ip;
{
        register struct file *fp;
        register *rip, m;
        int i;

        rip = ip;
        m = mode;
        if(trf != 2) {
                if(m&FREAD)
                        access(rip, IREAD);
                if(m&FWRITE) {
                        access(rip, IWRITE);
                        if((rip->i_mode&IFMT) == IFDIR)
                                u.u_error = EISDIR;
                }
        }
        if(u.u_error)
                goto out;
        if(trf)
                itrunc(rip);
        prele(rip);
        if ((fp = falloc()) == NULL)
                goto out;
        fp->f_flag = m&(FREAD|FWRITE);
        fp->f_inode = rip;
        i = u.u_ar0[R0];
        openi(rip, m&FWRITE);
        if(u.u_error == 0)
                return;
        u.u_ofile[i] = NULL;
        fp->f_count--;

out:
        iput(rip);
}
```

Backwards compatibility...

The Open Group Base Specifications Issue 6
IEEE Std 1003.1, 2004 Edition
Copyright © 2001-2004 The IEEE and The Open Group, All Rights reserved.

NAME

open - open a file

SYNOPSIS

```
[OH] ⌦ #include <sys/stat.h> ⌫

#include <fcntl.h>

int open(const char *path, int oflag, ... );
```

The use of *open*() to create a regular file is preferable to the use of *creat*(), because the latter is redundant and included only for historical reasons.

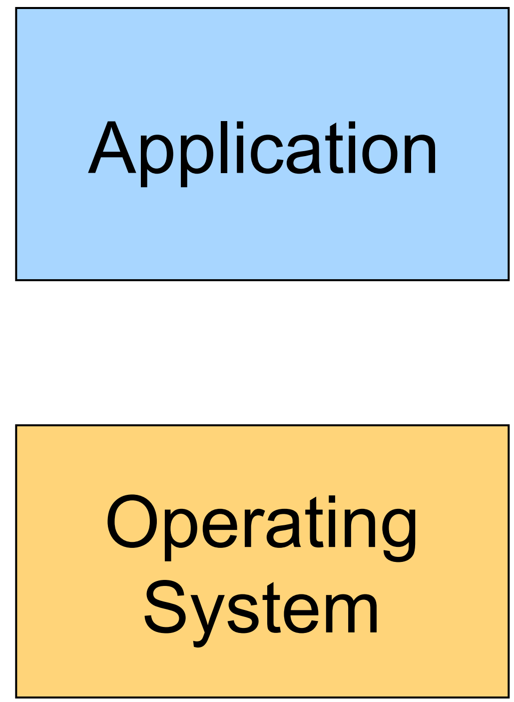
Division of Labors

Memory management example

- Kernel
  - Allocates "pages" with hardware protection
  - Allocates a big chunk (many pages) to library
  - Does not care about small allocs
- Library
  - Provides malloc/free for allocation and deallocation
  - Application use these calls to manage memory at fine granularity
  - When reaching the end, library asks the kernel for more

Feedback To The Program

- Applications view system calls and library calls as procedure calls
- What about OS to apps?
  - Various exceptional conditions
  - General information, like screen resize
- What mechanism would OS use for this?



Interrupts and Exceptions

- Interrupt Sources
  - Hardware (by external devices)
  - Software: INT n
- Exceptions
  - Program error: faults, traps, and aborts
  - Software generated: INT 3
  - Machine-check exceptions
- See Intel document volume 3 for details

Interrupts and Exceptions (1)

Vector #	Mnemonic	Description	Type
0	#DE	Divide error (by zero)	Fault
1	#DB	Debug	Fault/trap
2		NMI 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	

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

Example: Divide error

- What happens when your program divides by zero?
  - Processor exception
    - Defined by x86 architecture as INT 0
  - Jump to kernel, execute handler 0 in interrupt vector
  - Handler 0 sends SIGFPE to process
  - Kernel returns control to process
  - Process has outstanding signal
  - Did process register SIGFPE handler?
    - Yes:
      - Execute SIGFPE handler
      - When handler returns, resume program and redo divide
    - No: kills process

Summary

- Protection mechanism
  - Architecture support: two modes
  - Software traps (exceptions)
- OS structures
  - Monolithic, layered, microkernel and virtual machine
- System calls
  - Implementation
  - Design issues
  - Tradeoffs with library calls