COS 318: Operating Systems

OS Structures and System Calls

Jaswinder Pal Singh
Computer Science Department
Princeton University

(http://www.cs.princeton.edu/courses/cos318/)

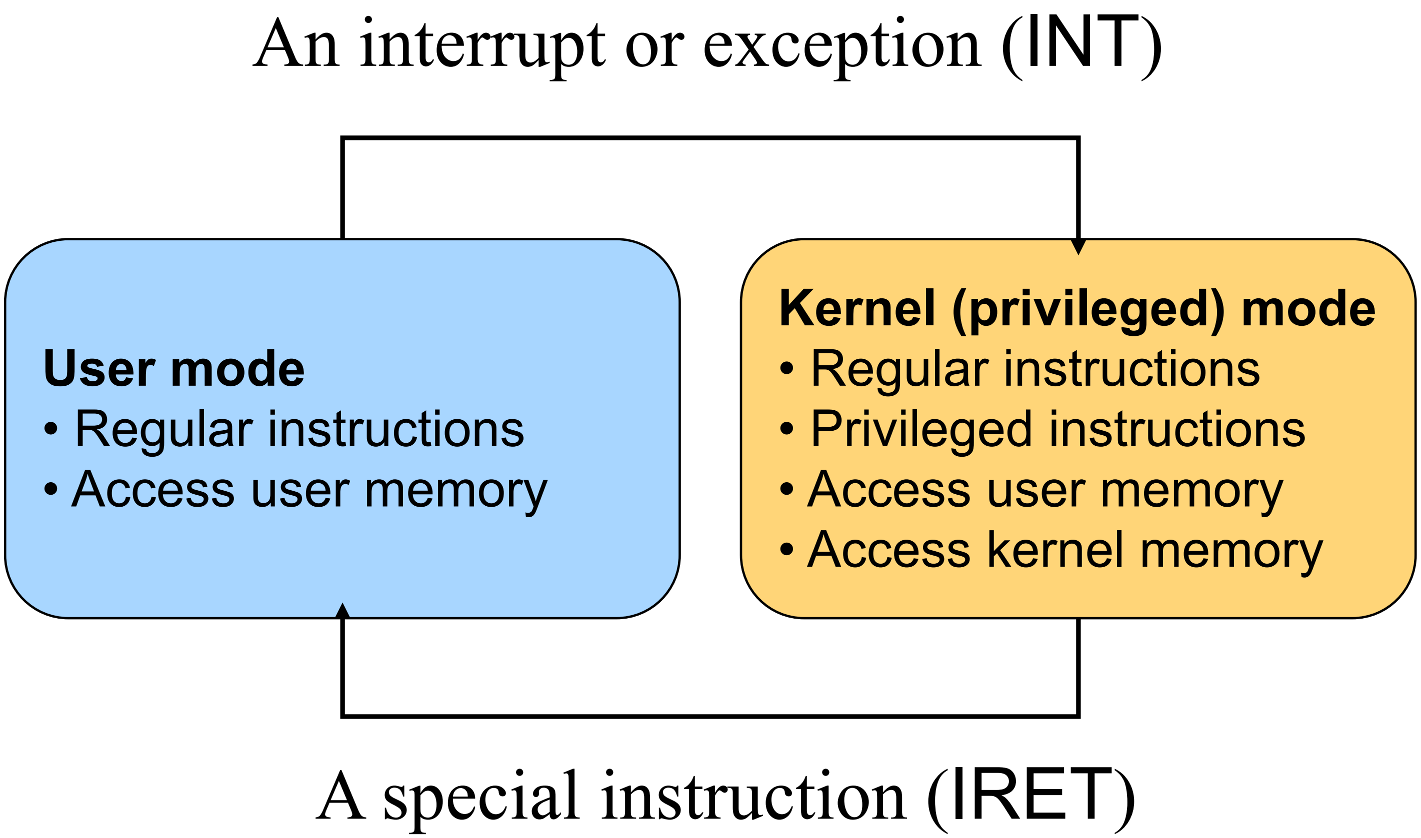
Outline

- 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

Architecture Support: Privileged Mode

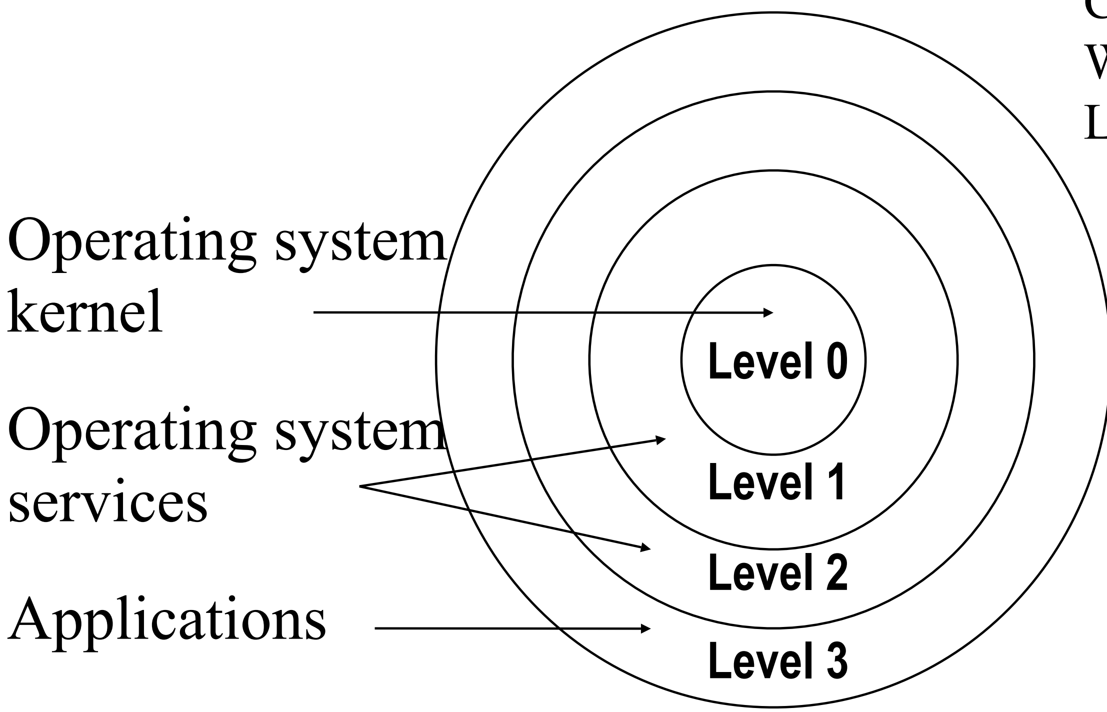


Privileged Instruction Examples

- 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

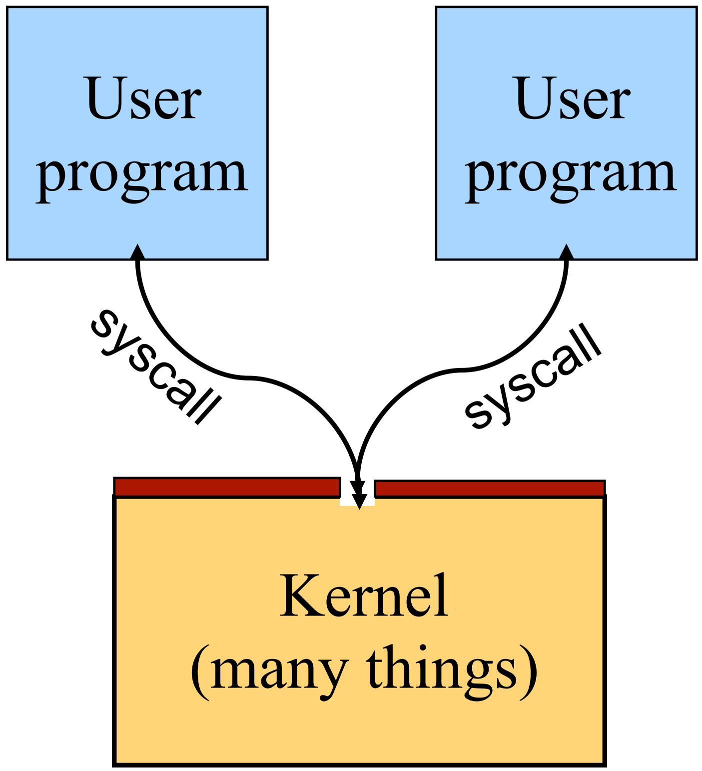


Outline

- Protection mechanisms
- OS structures
- System and library calls

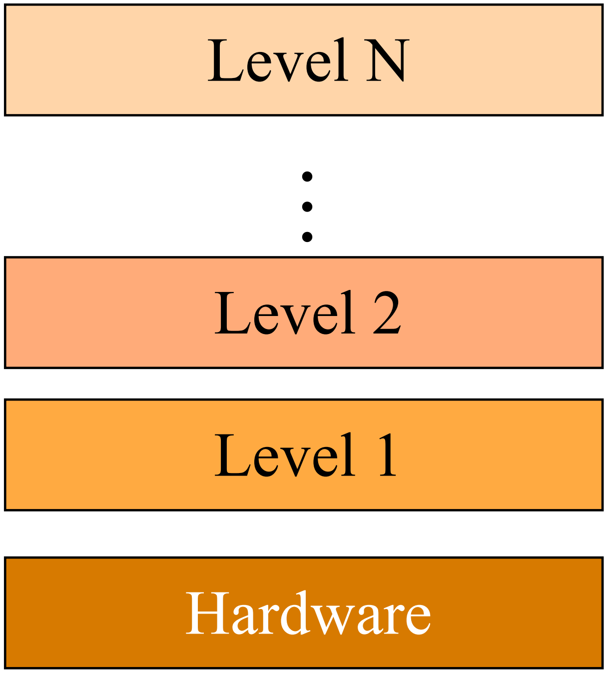
Monolithic

- All kernel routines are together, any can call any
- A system call interface (main program, sys calls, utility funcs)
- Examples:
  - Linux, BSD Unix, Windows
- Pros
  - Shared kernel space
  - Good performance
- Cons
  - No information hiding
  - Inflexible
  - Chaotic
  - Difficult to understand
  - How many bugs in 5 million lines of code?



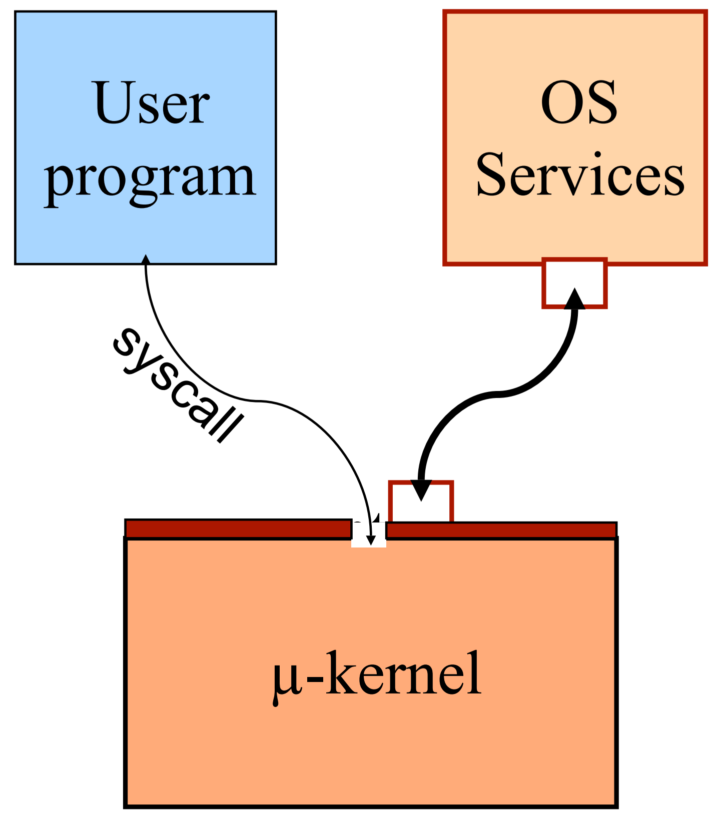
Layered Structure

- Level N constructed on top of N-1
- Hiding information at each layer
- E.g. level 1 is processor allocation, level 1 memory management, level 2 comm, level 3 I/O, etc.
- Examples
  - THE System (6 layers)
  - MS-DOS (4 layers)
- Pros
  - Layered abstraction
  - Separation of concerns, elegance
- Cons
  - Protection, boundary crossings
  - Performance



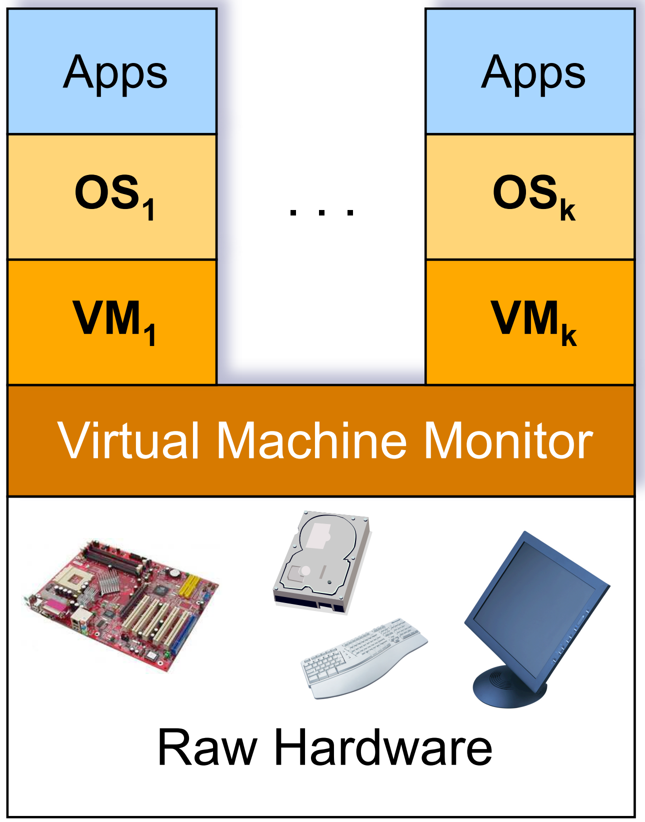
Microkernel

- Put less in kernel mode: only small part of OS; reduce kernel bugs
- Services are regular processes; one file system crashing doesn't crash full system; can't corrupt kernel memory
- μ-kernel gets svcs on behalf of users by messaging with service processes
- Examples:
  - Mach, Taos, L4, OS-X
- Pros?
  - Flexibility
  - Fault isolation and reliability (used in avionics and military apps)
- Cons?
  - Inefficient (boundary crossings)
  - Insufficient protection
  - Inconvenient to share data between kernel and services

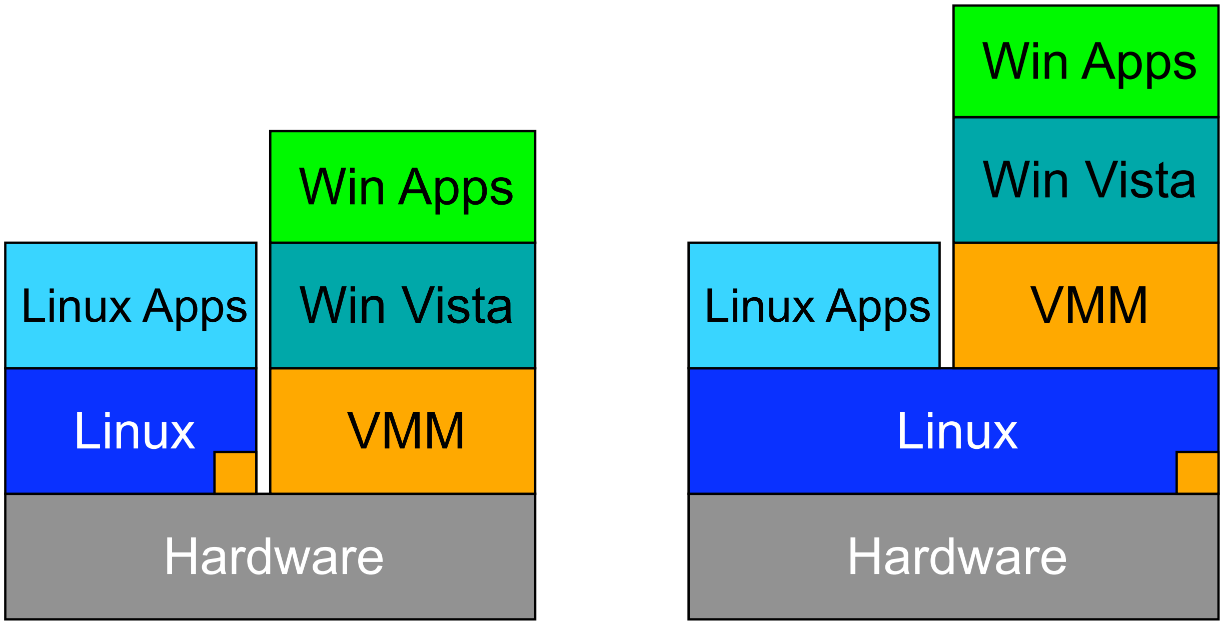


Virtual Machine

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



Two Popular Ways to Implement VMM



VMM runs on hardware

VMM as an application

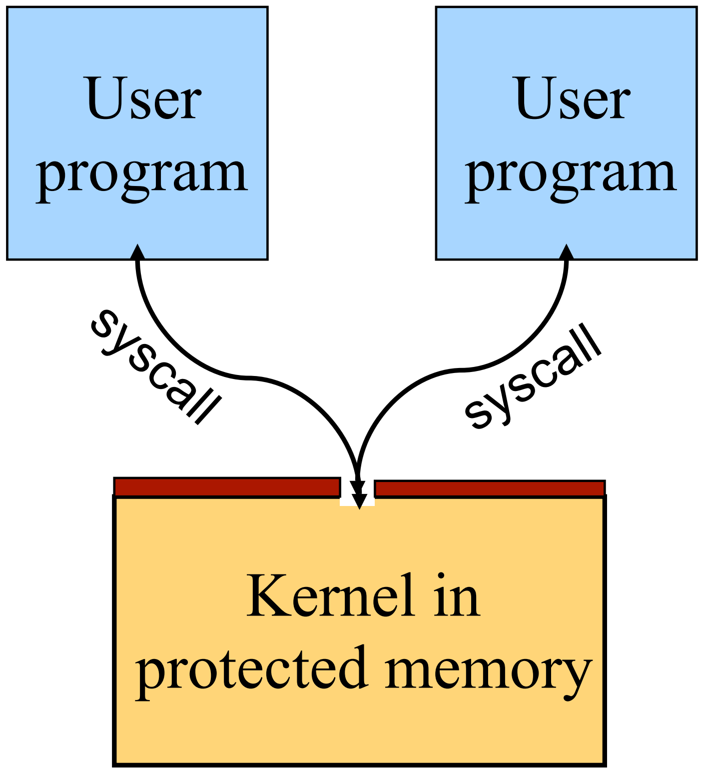
(A special lecture later in the semester)

Outline

- Protection mechanisms
- OS structures
- System and library calls

System Call Mechanism

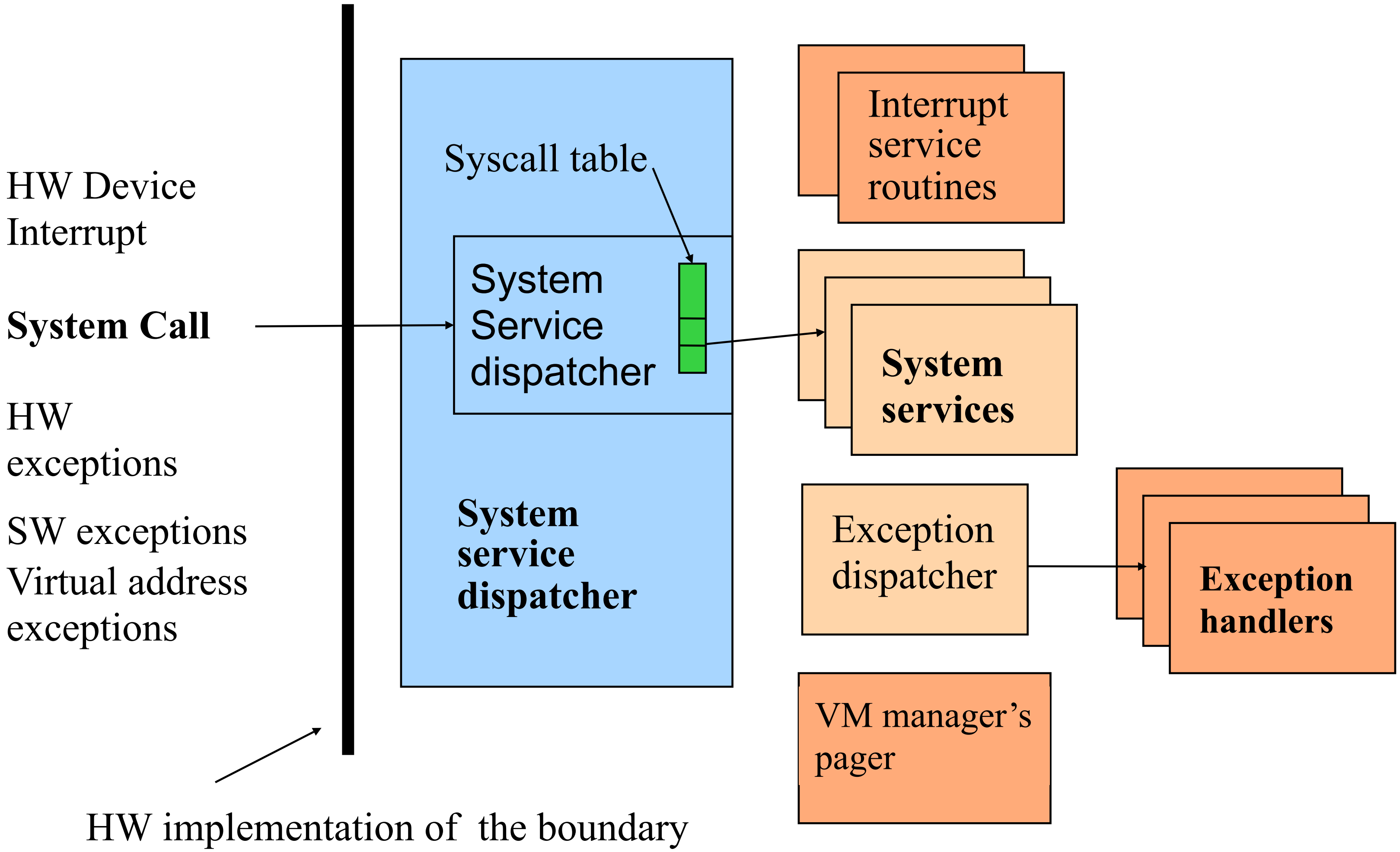
- Assumptions
  - User code can be arbitrary
  - User code cannot modify kernel memory
- Design Issues
  - User code makes a system call with parameters
  - The call mechanism switches code to kernel mode
  - Execute system call
  - Return with results
  - (Like a procedure call, just crosses kernel boundary)



System Calls

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

OS Kernel: Trap Handler



Passing Parameters

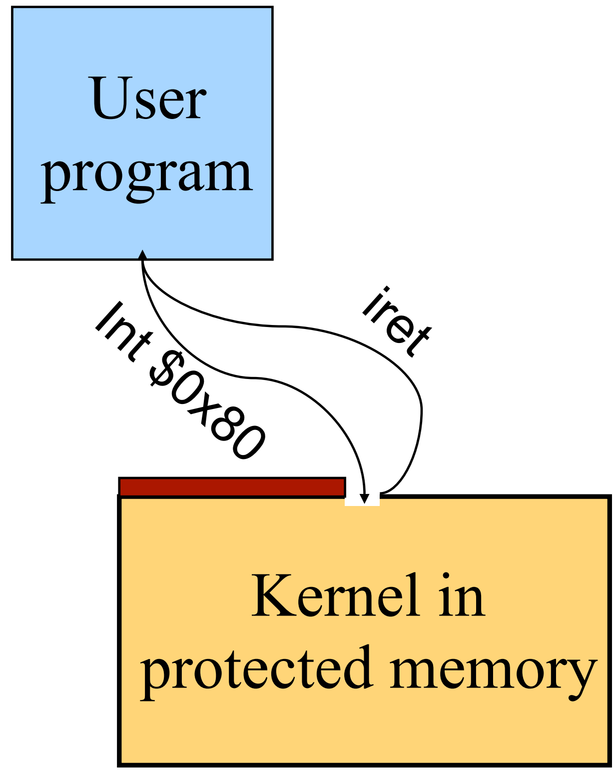
- Pass by registers
  - # of registers
  - # of usable registers
  - # of parameters in system call
  - Spill/fill code in compiler
- Pass by a memory vector (list)
  - Single register for starting address
  - Vector in user's memory
- Pass by stack
  - Similar to the memory vector
  - Procedure call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

Library Stubs for System Calls

- Example:

```
int read( int fd, char * buf, int size)
{
    move fd, buf, size to R1, R2, R3
    move READ to R0
    int $0x80
    move result to Rresult
}
```

Linux: 80
NT: 2E



System Call Entry Point

EntryPoint:

switch to kernel stack

save context

check R_0

call the real code pointed by R_0

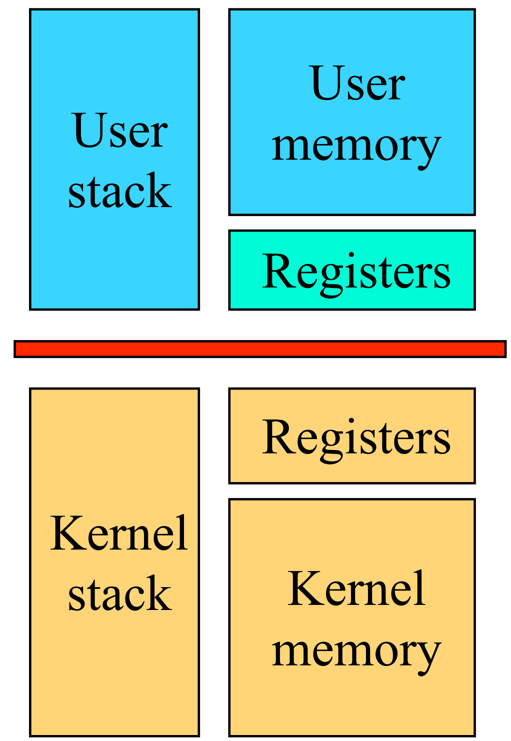
place result in R_{result}

restore context

switch to user stack

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?
  - Can user code lie?
  - How would you perform QA on system calls?
- System calls vs. library calls
  - What should be system calls?
  - What should be library calls?

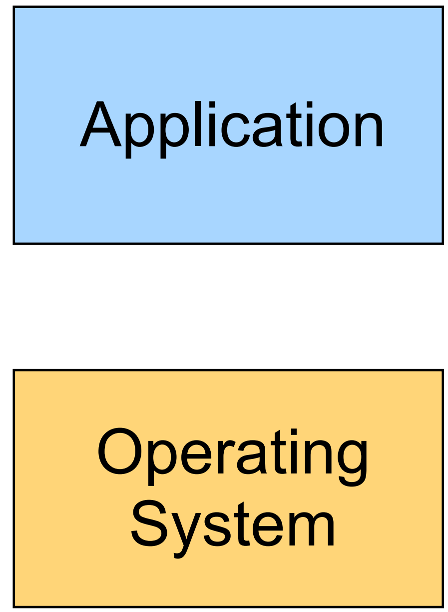
Division of Labor (or Separation Of Concerns)

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?



Interrupt 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

Interrupt 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	

Interrupt 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

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