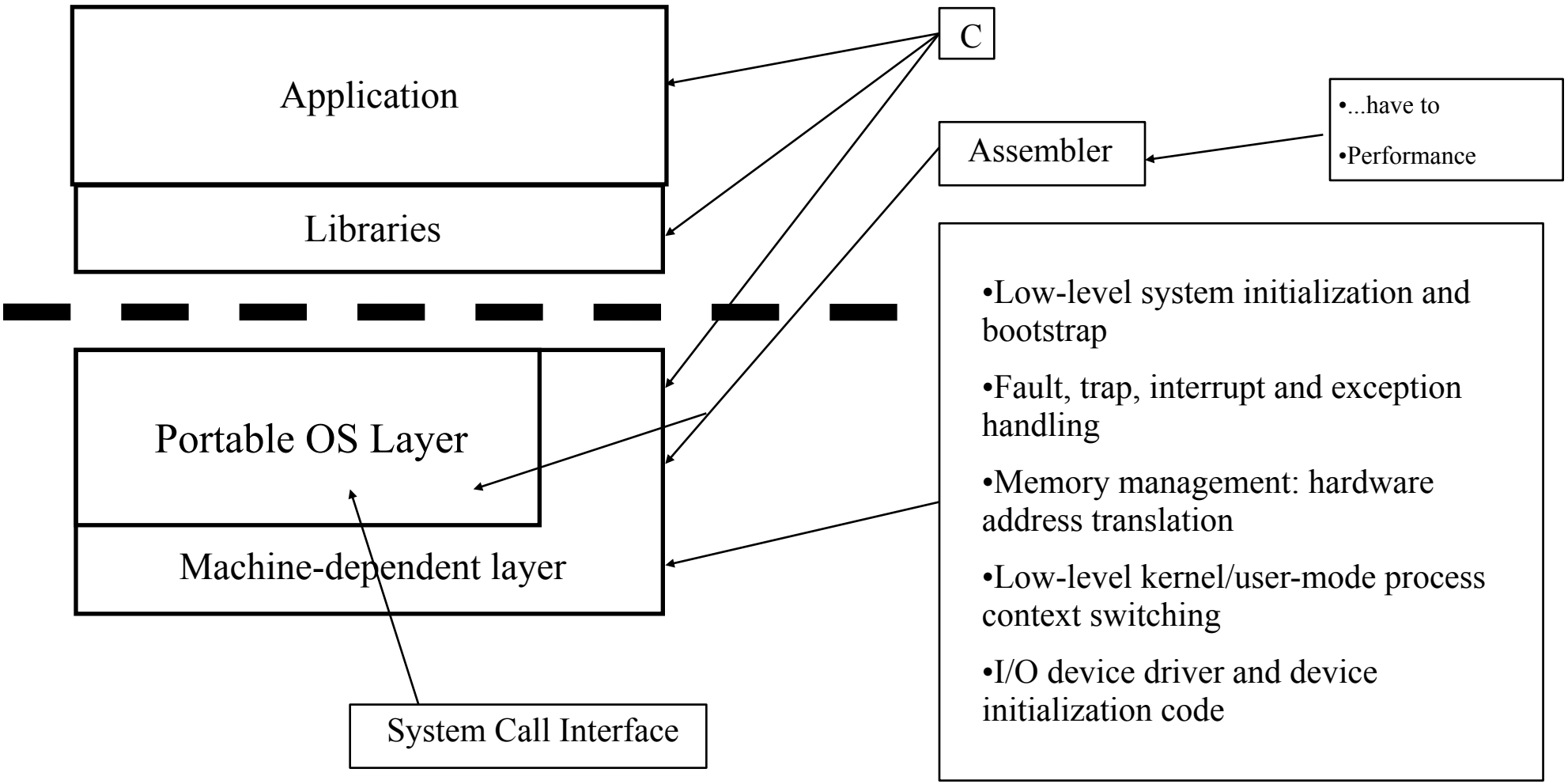
Protection and System Calls

Otto J. Anshus

Protection Issues

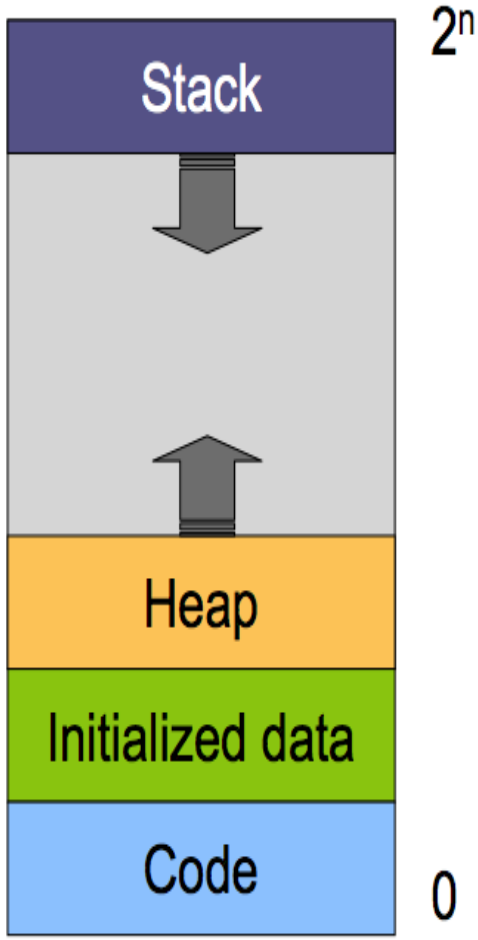
- CPU protection
  - Prevent a user from using the CPU for too long
    - Throughput of jobs, and response time to events (incl. user interactive response time)
- Memory protection
  - Prevent users from modifying kernel code and data structures
  - …and each others code and data
- I/O protection
  - Prevent users from performing illegal I/O's
- Question to ponder during the fast approaching long winter nights
  - what is the difference between protection and security?

Typical Unix OS Structure



What is a Process?

- Four segments
  - Code/text: instructions
  - Data: variables
  - Stack
  - Heap
- Why?
  - Separate code and data
  - Stack and heap grow toward each other



- **Stack**
  - Layout by compiler
  - Allocate at process creation (fork)
  - Deallocate at process termination
- **Heap**
  - Linker and loader specify the starting address
  - Allocate/deallocate by library calls such as **malloc()** and **free()** called by application
- **Data**
  - Compiler allocate statically
  - Compiler specify names and symbolic references

– **Linker** translate references and relocate addresses

– **Loader** finally lay them out in memory

*NB: **WE** must figure out how to let the OS **implement** all of this and how to **handle** a running process*

Registers Directly Used by User Level Processes

In “protected mode”, there are **8** 32-bit general-purpose registers for use:

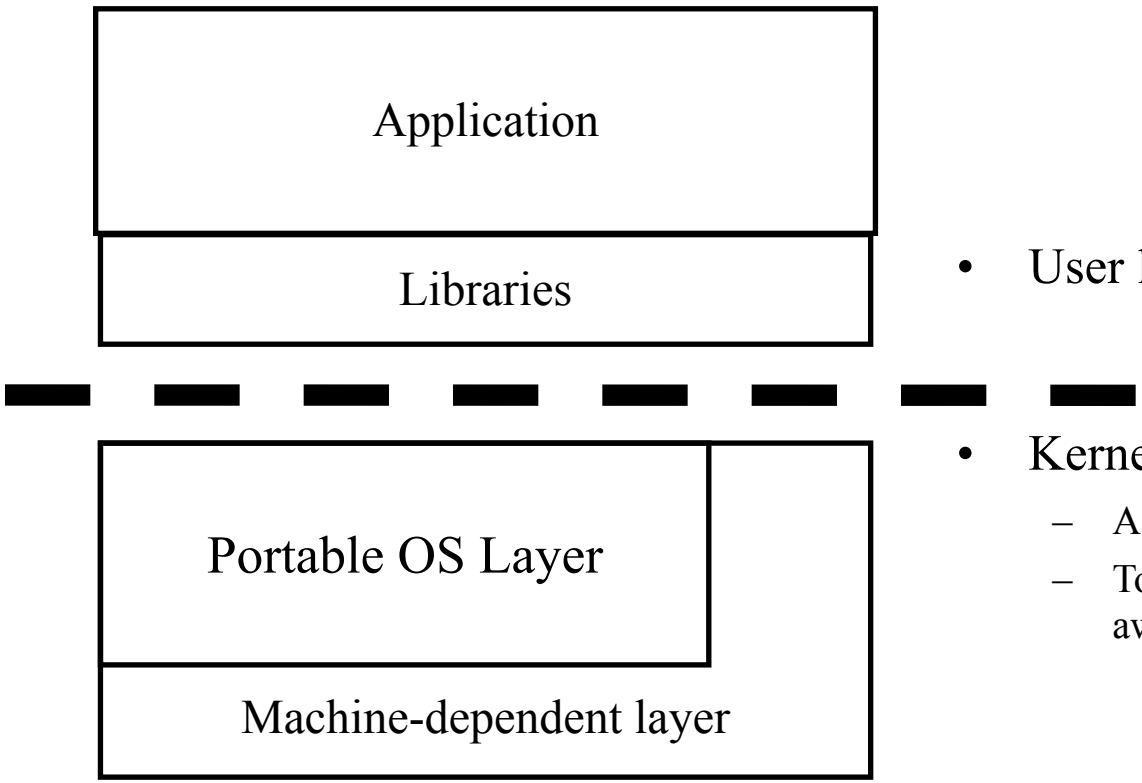
- data registers
  - EAX, the accumulator (32 bits (16 and AX (AH, AL)))
  - EBX, the base register
  - ECX, the counter register
  - EDX, the data register
- address registers
  - ESI, the source register
  - EDI, the destination register
  - ESP, the stack pointer register
  - EBP, the stack base pointer register

Registers accessed by Kernel (cannot be directly accessed by user level processes)

Registers used by OS Kernel changing the state of the processor:

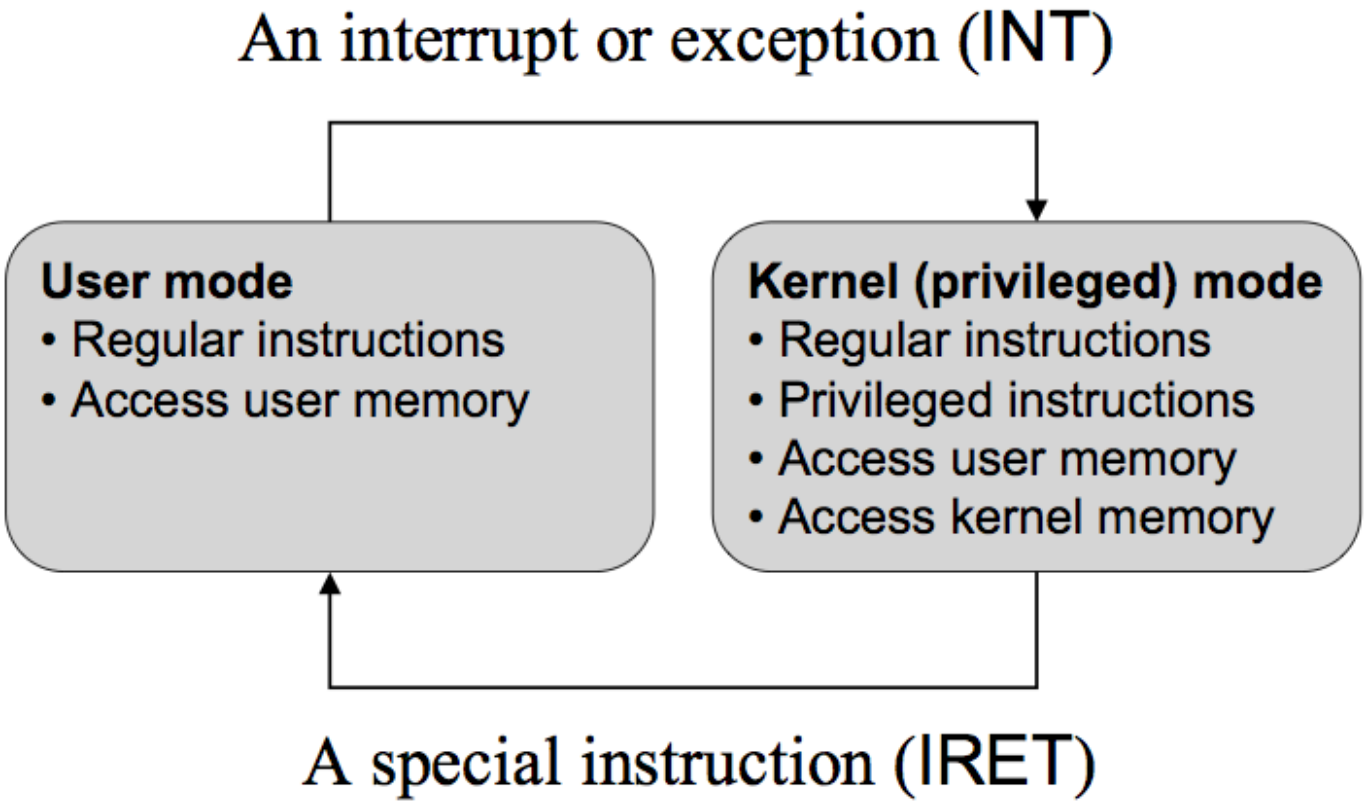
- **control** registers
  - CR0, CR1, CR2, CR3
- **test** registers
  - TR4, TR5, TR6, TR7
- **descriptor** registers
  - GDTR, the global descriptor table register (see below)
  - LDTR, the local descriptor table register (see below)
  - IDTR, the interrupt descriptor table register (see below)
- task register
  - TR

User level vs. Kernel level



- User level
  - Some instructions are not available any more
  - Programs can be modified and substituted by user
- Kernel (a.k.a. supervisory or privileged) level
  - All instructions are available
  - Total control possible so OS should never ever give away to a user process the right to run at kernel level

Architecture Support: Privileged Mode



Boot OS and Run It and User Level Processes

- Boot OS
  - power on/start up code reads boot block from HD to memory and transfer control to boot block code
  - boot block code reads OS from HD to memory
  - OS is given control

- OS starting user program (first time)
  - Read (already compiled and readied program) image from disk
  - Initialize OS kernel data structures with info about the program
  - ATOMICALLY DO {<Set privilege level ”user”>; <Load instruction register from start of program>} **% why ‘atomically?**
  - Now we have a user level PROCESS running

- User level process requesting OS service
  - Make a mark ”somewhere” (memory, stack, registers) indicating which service is requested and where to find the parameters
  - Execute instruction that is bound to trap in decode

- *Still need mechanism that allows OS to ”preempt” user process, OS needs to be activated independently of running program. That can only be achieved “external” to the running program’s instruction stream.*

What to Do When User Level Process is Trying to Execute an Illegal Instruction

- **Instruction Stream**
  - **Fetch instruction**
  - **Decode instruction**
  - **Fetch operands**
  - **Execute**
  - **Write back result**
  - **Next instruction**

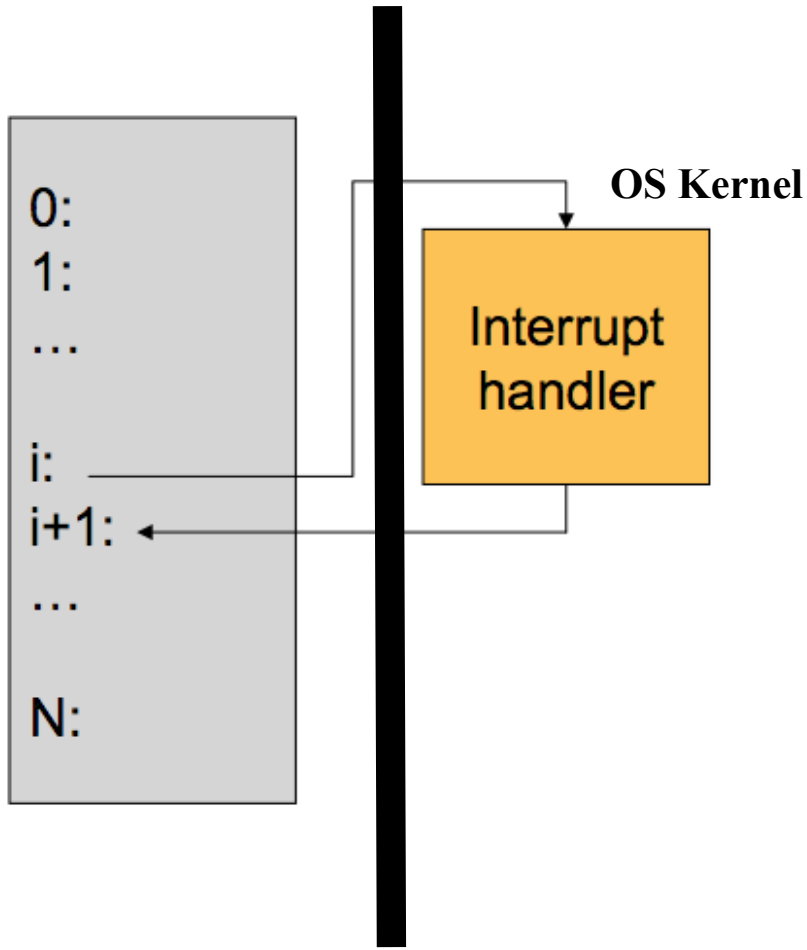
When **decoding** instruction, what to do if bit-pattern doesn't represent a (legal) instruction?

- Halt? (No, but why not?)
- Instead: "**Trap**": fetch next instruction at "predetermined" address in memory. Make sure that you have placed your (OS) code there beforehand

Interrupts and Traps

Application **IS** *program* being executed **IS** at least one *process* (with at least one *thread* each)

- Interrupts
  - Raised by external events
  - CPU resume from the interrupt handler in the kernel
    - switch to next process
      - **iret** instruction: returns by popping return address from stack, and enable interrupts (IA32 instruction set)
- Traps
  - Internal events
  - System calls (syscalls)
  - Also return by **iret**



Interrupts and Exceptions

- Interrupt sources
  - HW (by external devices)
  - SW: **INT n**
- Exceptions (something unexpected or needing action happened)
  - Program errors
    - **faults**: save address (CS, EIP) of faulting instruction, “fix” fault, restart instruction
      - Case: page fault, divide by zero
    - **traps**: save address of instruction
      - Case: debugger
    - **aborts**: oops (no saving, not recoverable)
      - Case: your first attempts at OS kernel code
  - SW generated: **INT 3**
  - Machine-check exceptions
- **See Intel doc Vol. 3 for details**

Interrupts and Exceptions

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	

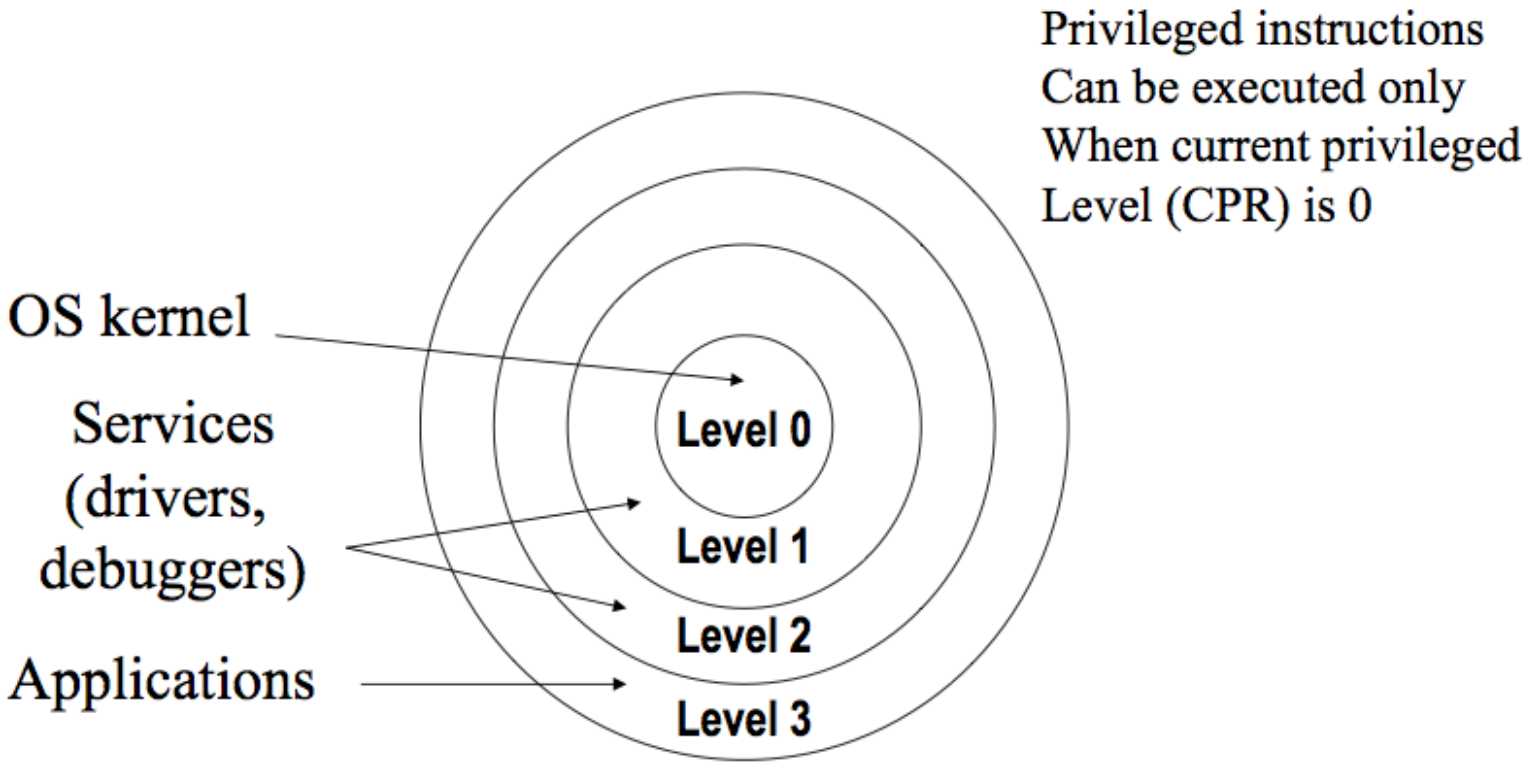
Interrupts and Exceptions

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

Privileged Instruction Examples

- **Memory** address mapping
- Data cache flush and invalidation
- Invalidating TLB entries
- Loading and reading **system** registers
- Changing **processor mode** from kernel to user
- Changing the voltage and frequency of the processor
- **Halting** a processor
- **Reset** a processor
- **I/O** operations

IA32 Protection Rings



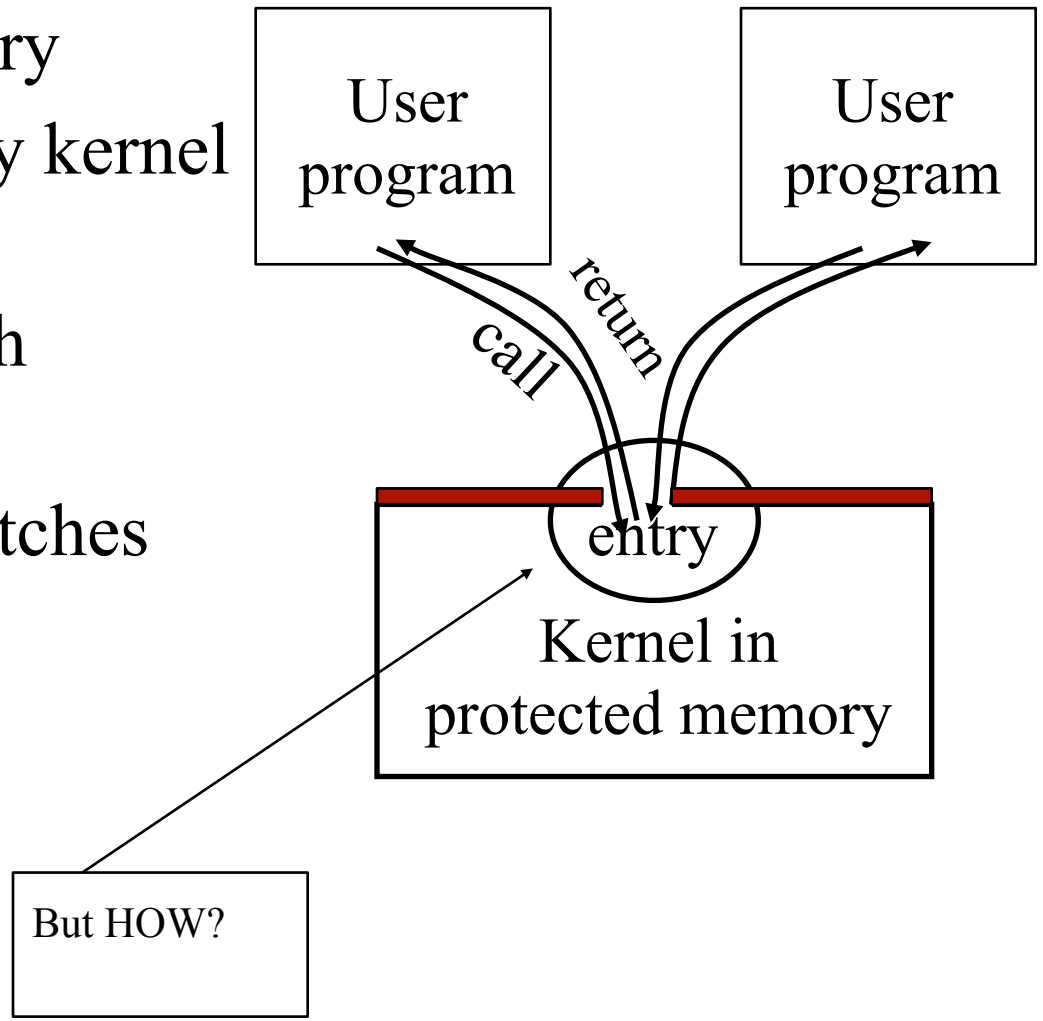
No worries, we will use level 0 and 3

System Calls

- Operating System API
  - Interface between a process and OS kernel
  - Seen as a set of library functions
- Process management
  - end, abort , load, execute, create, terminate, set, wait
- Memory management
    - mmap & munmap, mprotect, mremap, msync, swapon & off,
- File management
    - create, delete, open, close, R, W, seek
- Device management
    - res, rel, R, W, seek, get & set atrib., mount, unmount
- Communication
    - get ID's, open, close, send, receive

System Call Mechanism

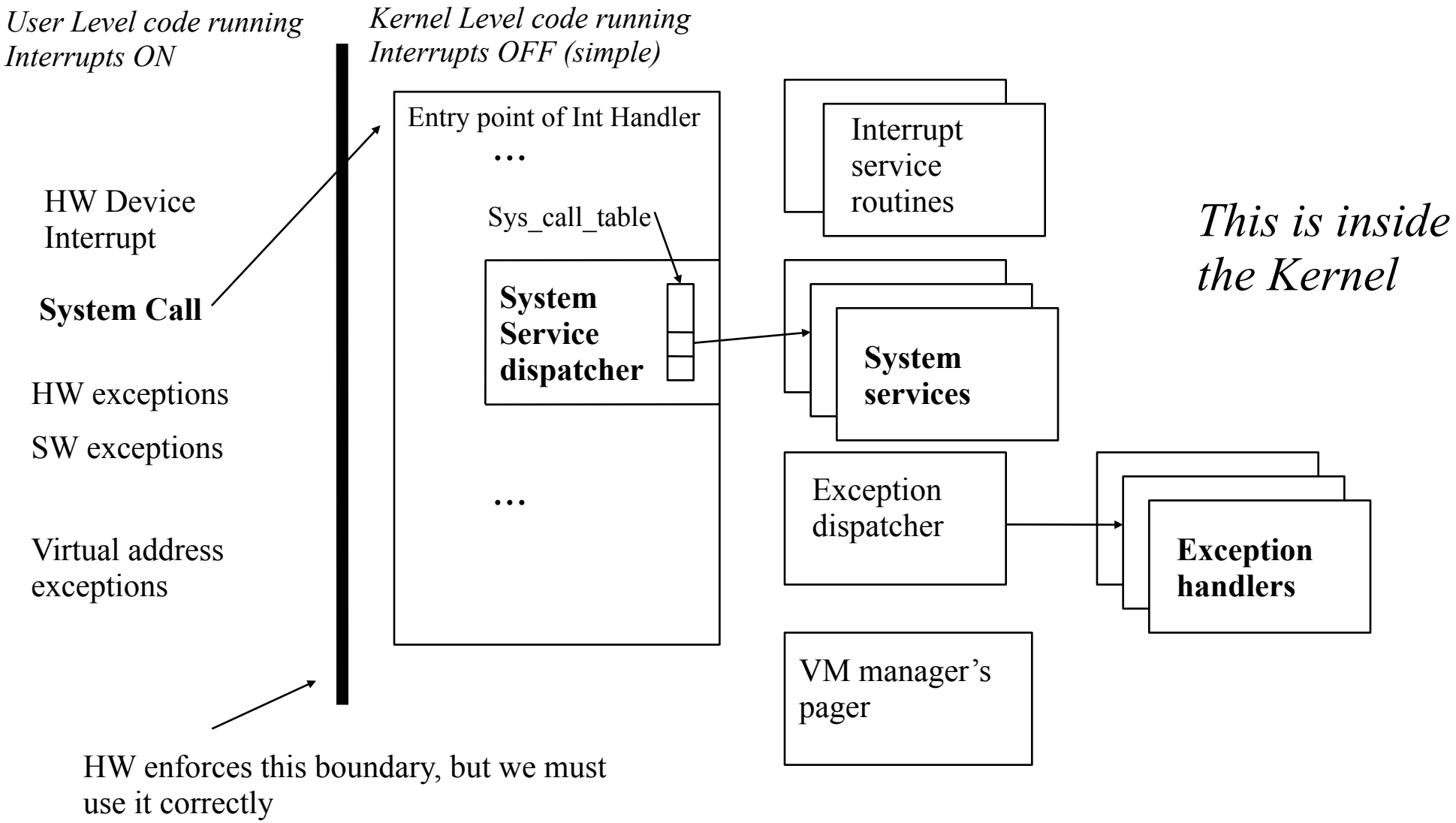
- User code can be arbitrary
- User code cannot modify kernel memory
- Makes a system call with parameters
- The call mechanism switches code to kernel mode
- Execute system call
- Return with results



System Call Implementation

- Use an “interrupt”
  - Hardware devices (keyboard, serial port, timer, disk,…) and **software** can request service using interrupts
  - The CPU is interrupted
    - ...and a service handler routine is run
  - …when finished the CPU resumes from where it was interrupted (or somewhere else determined by the OS kernel)

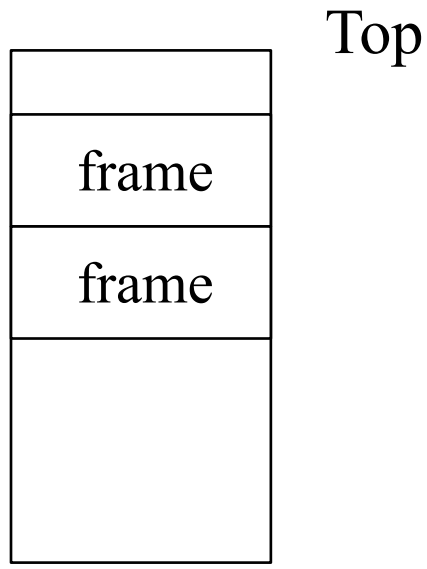
OS Kernel: Interrupt/Trap/Exception Handler



Passing Parameters

- Pass *by registers*
  - #registers
  - #usable registers
  - #parameters in syscall
- Pass *by memory vector*
  - A register holds the address of a location in users memory
- Pass *by stack*
  - Push: done by library
  - Pop: done by Kernel

REMEMBER: Kernel has access to callers address space, but not vice versa



The Stack

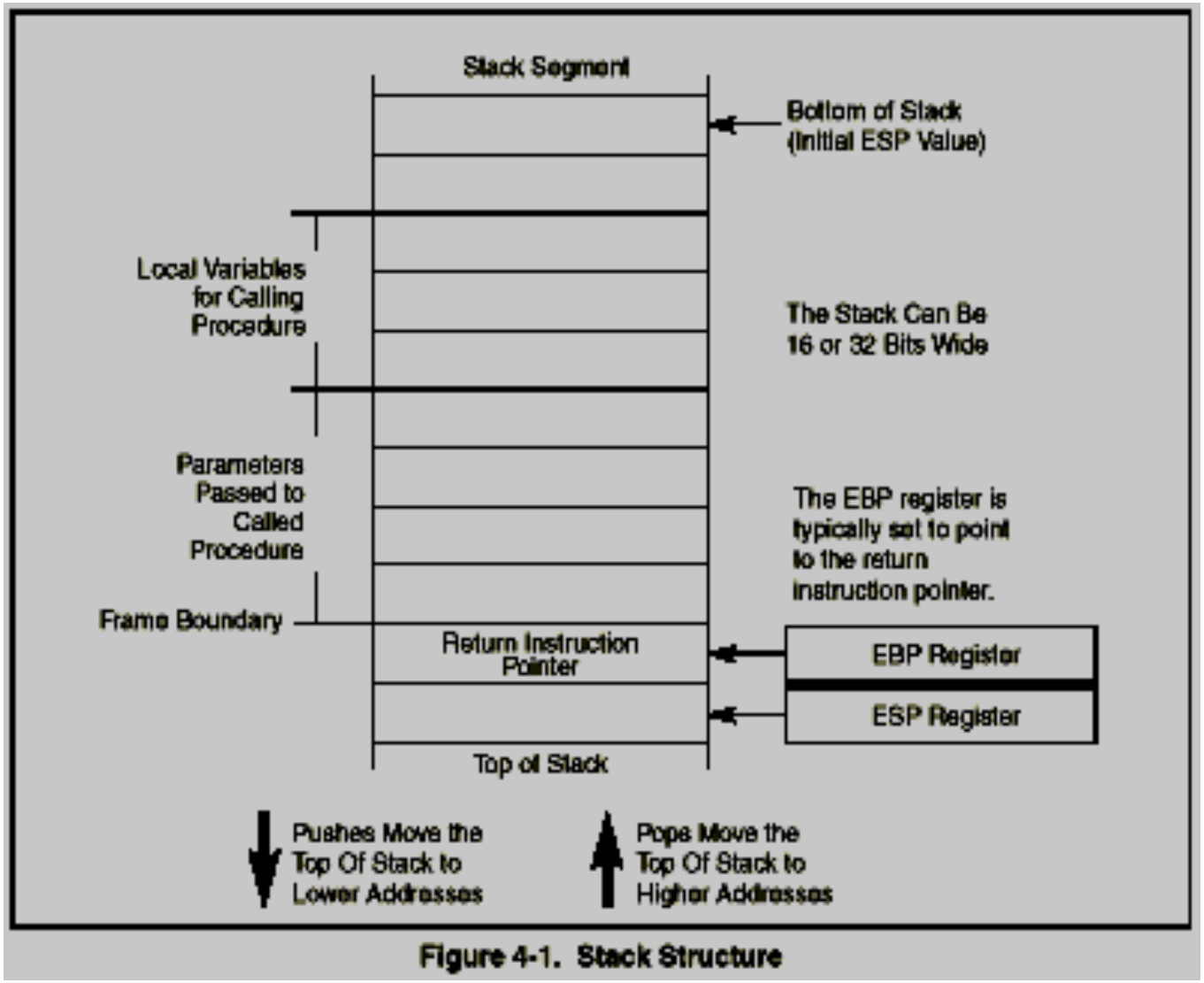


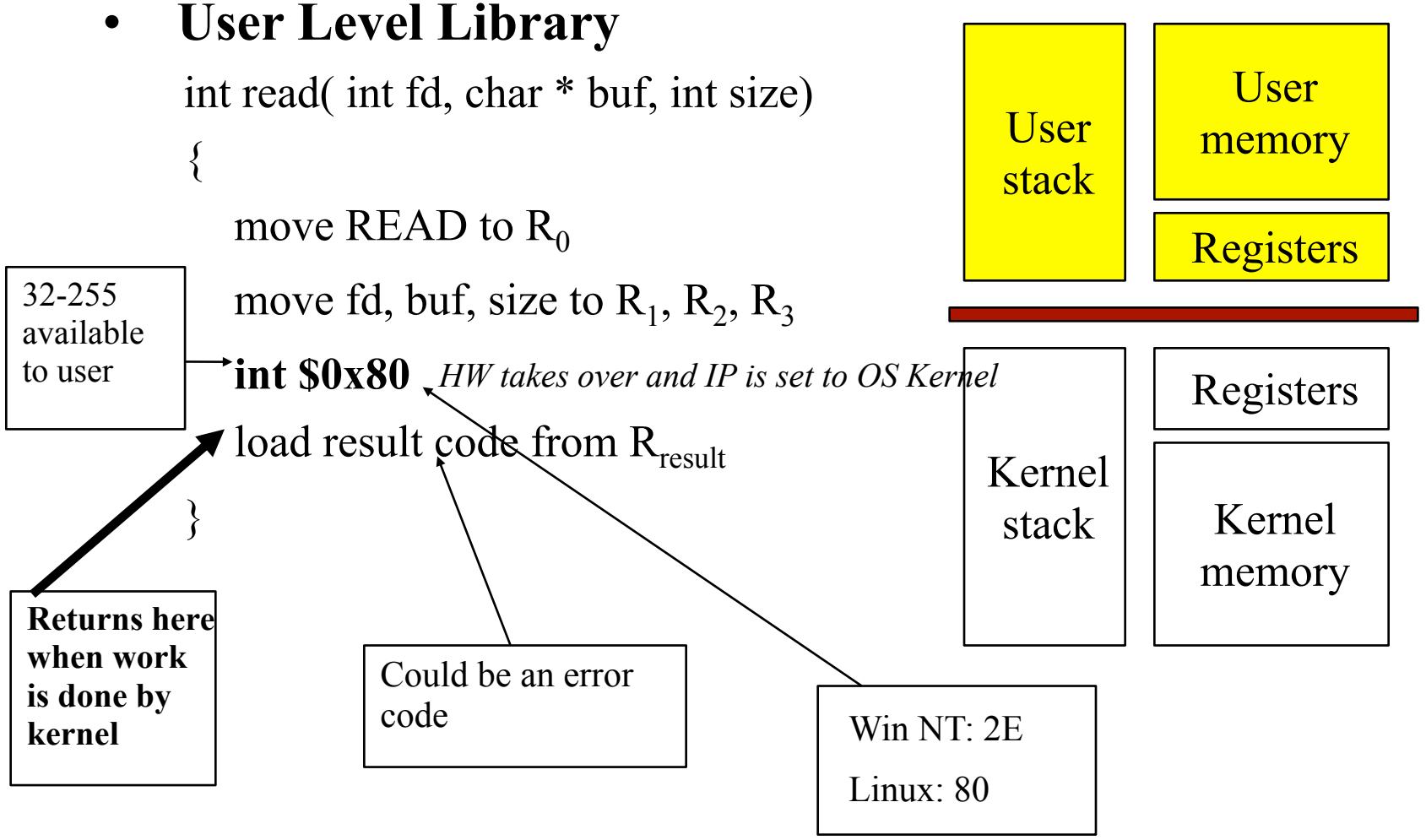
Figure 4-1. Stack Structure

- Many stacks possible, but only one is "current": the one in the segment referenced by the SS register
- Max size 4 gigabytes
- PUSH: write (--ESP);
- POP: read(ESP++);
- When setting up a stack remember to align the stack pointer on 16 bit word or 32 bit double-word boundaries

Library Stubs for System Calls

- **User Level Process** calls: read(fd, buf, size);
- **User Level Library**

```
int read( int fd, char * buf, int size)
{
    move READ to R0
    move fd, buf, size to R1, R2, R3
    int $0x80      HW takes over and IP is set to OS Kernel
    load result code from Rresult
}
```



A simplified Interrupt Handler

System Call Entry Point in Kernel

int 0x80

SW interrupt

- Assume passing parameters in registers

EntryPoint inside OS Kernel:

switch to kernel stack;

save user context;

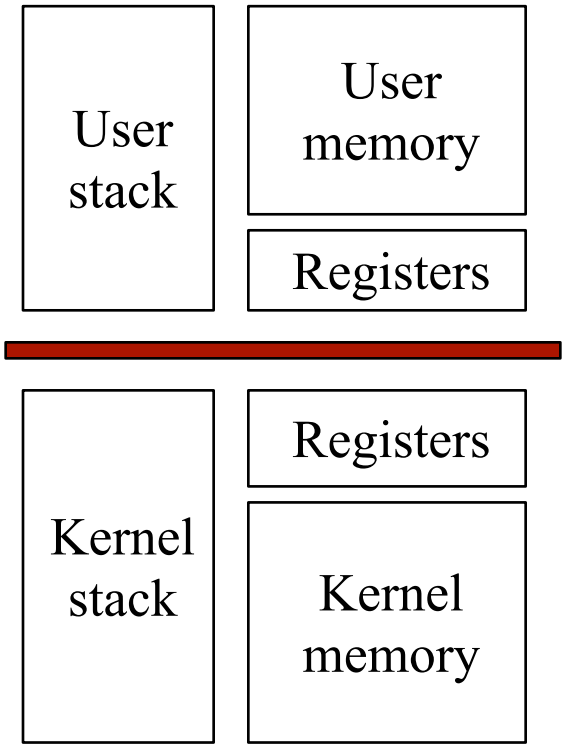
if legal(R_0) **call** service;

restore user context;

switch to user stack;

iret;

Kernel Mode: Total control. All interrupts are disabled



NB, black frame means KL...

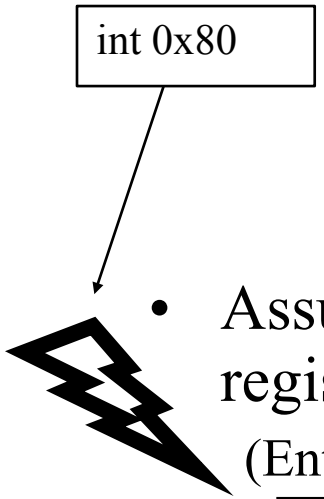
Change to user mode and return

Puts results into buf

Or: User stack

Or: some register

System Call Entry Point



- Assume passing parameters in registers

(EntryPoint:)

switch to kernel stack;

save all registers;

if legal(R_0) call sys_call_table[R_0];

restore user registers;

switch to user stack;

iret; %next instr in user space app

Save/Restore Context?

If envoked code executes for a long time: should SCHEDULE or at least ENABLE interrupts (here or inside service routine).

READ returns with result and handler must return them to user

(Or **SCHEDULE** to run another process)

Polling instead of Interrupt?

- OS kernel could check a request queue of syscalls instead of using an interrupt?
  - Waste CPU cycles checking
  - All have to wait while the checks are being done
  - When to check?
    - Non-predictable
    - Pulse every 10-100ms?
      - » too long time
- Same valid for HW Interrupts vs. Polling

But used for Servers

- However, *spinning* can give good performance (more later)

Design Issues for Syscall

- We used only one result reg, what if more results?
- In kernel and in called service: Use caller's stack or a special stack?
  - Use a special stack
- Quality assurance
  - Use a single entry or multiple entries?
    - Simple is good?
      - Then a single entry is simpler, easier to make robust
- Can kernel code call system calls?
  - Yes, but should avoid the entry point mechanism

System calls vs. Library calls

- Division of labor (a.k.a. Separation of Concerns)
- Memory management example
  - Kernel
    - Allocates “pages” (w/HW protection)
    - Allocates many “pages” (a big chunk) to library
      - Big chunks, no “small” allocations
  - Library
    - Provides malloc/free for allocation and deallocation of memory
    - Application use malloc/free to manage its own memory at **fine** granularity
    - When no more memory, library asks kernel for a new chunk of pages

User process vs. kernel

- To go from User Level Process to Kernel
  - syscalls using *int*
- To go from Kernel to User Level Process
  - *iret* (with "good" stack)
  - Kernel is all powerful
    - Can write into user memory
    - Can terminate, block and activate user processes