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
  - what is the difference between protection and security?

Application Registers

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

Non-Application Registers

In addition there are non-application registers available, which change 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

Flags Registers

- EFLAGS, which contain the processor state.
  - Each flag is one bit - and thus set 0 or 1, also called set, high, and unset or low.
  - Important flags in the EFLAGS register is: carry (bit 0), zero (bit 6), sign flag (bit 7) and overflow (bit 12).

- Flags are used in the x86 architecture for comparisons.
  - A comparison is made between two registers, for example, and in comparison of their difference a flag is raised.
    - A jump instruction then checks the respective flag and jumps if the flag has been raised: for example

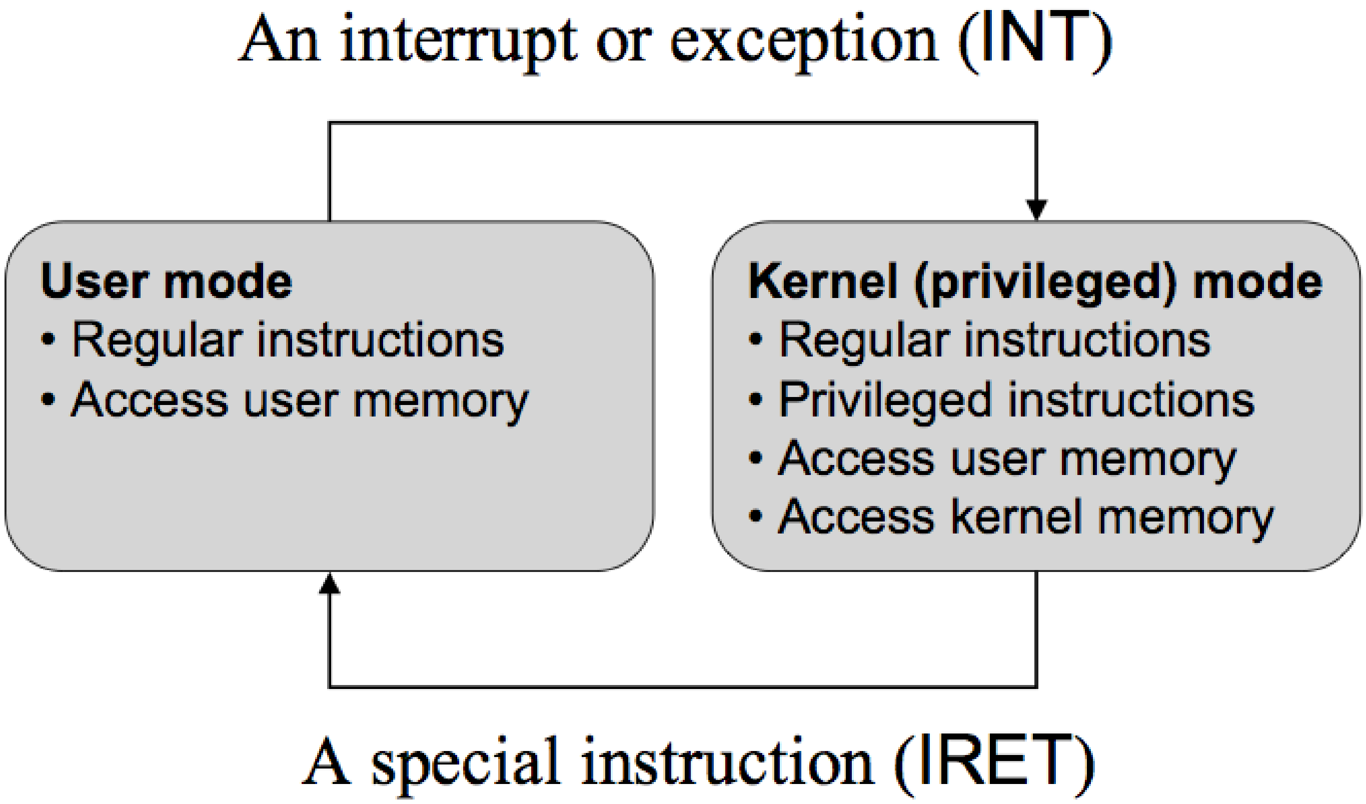
      cmp ax, bx *jne* do_something

      first compares the AX and BX registers, and if they are unequal, the code branches off to the do_something label.

Instruction Pointer Register

- EIP
  - The IP register points to where in the program the processor is currently executing it's code.
  - The IP register cannot be accessed by the programmer *directly*. **WHY NOT?**

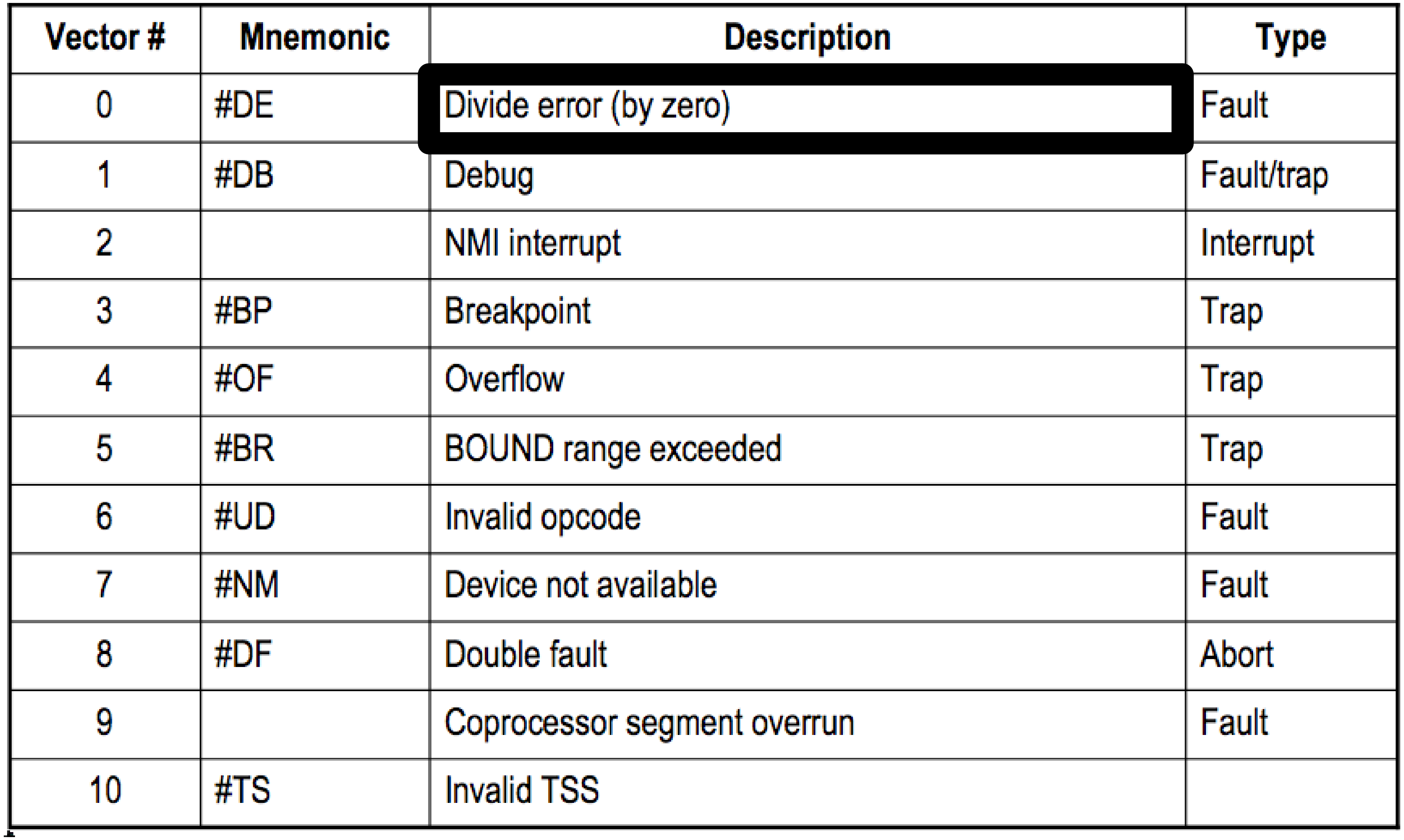
Architecture Support: Privileged Mode



Interrupts and Exceptions

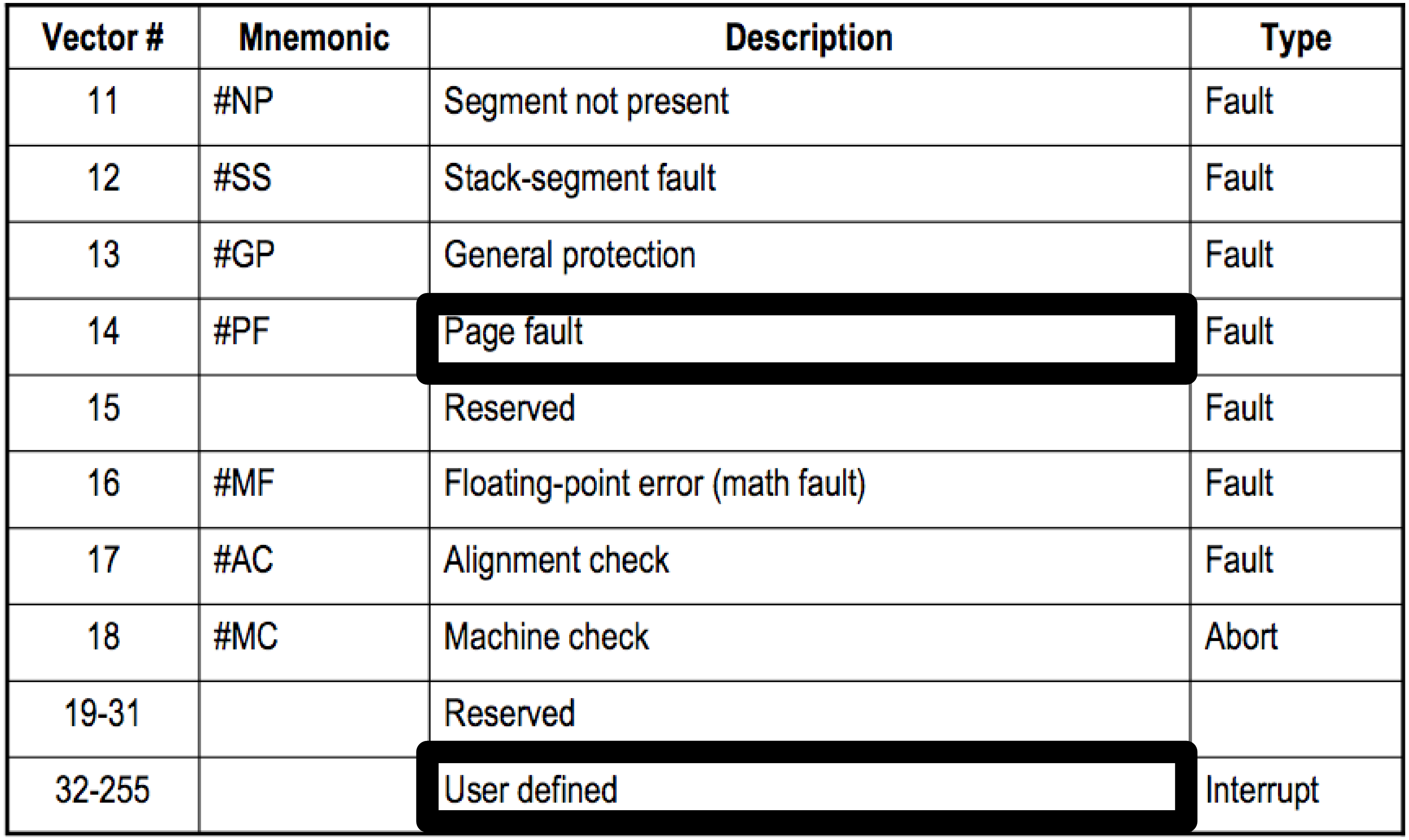
- Interrupt sources
  - HW (by external devices)
  - SW: **INT n**
- Exceptions
  - Program errors: faults, traps, aborts
  - SW generated: **INT 3**
  - Machine-check exceptions
- See Intel doc Vol. 3 for details

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

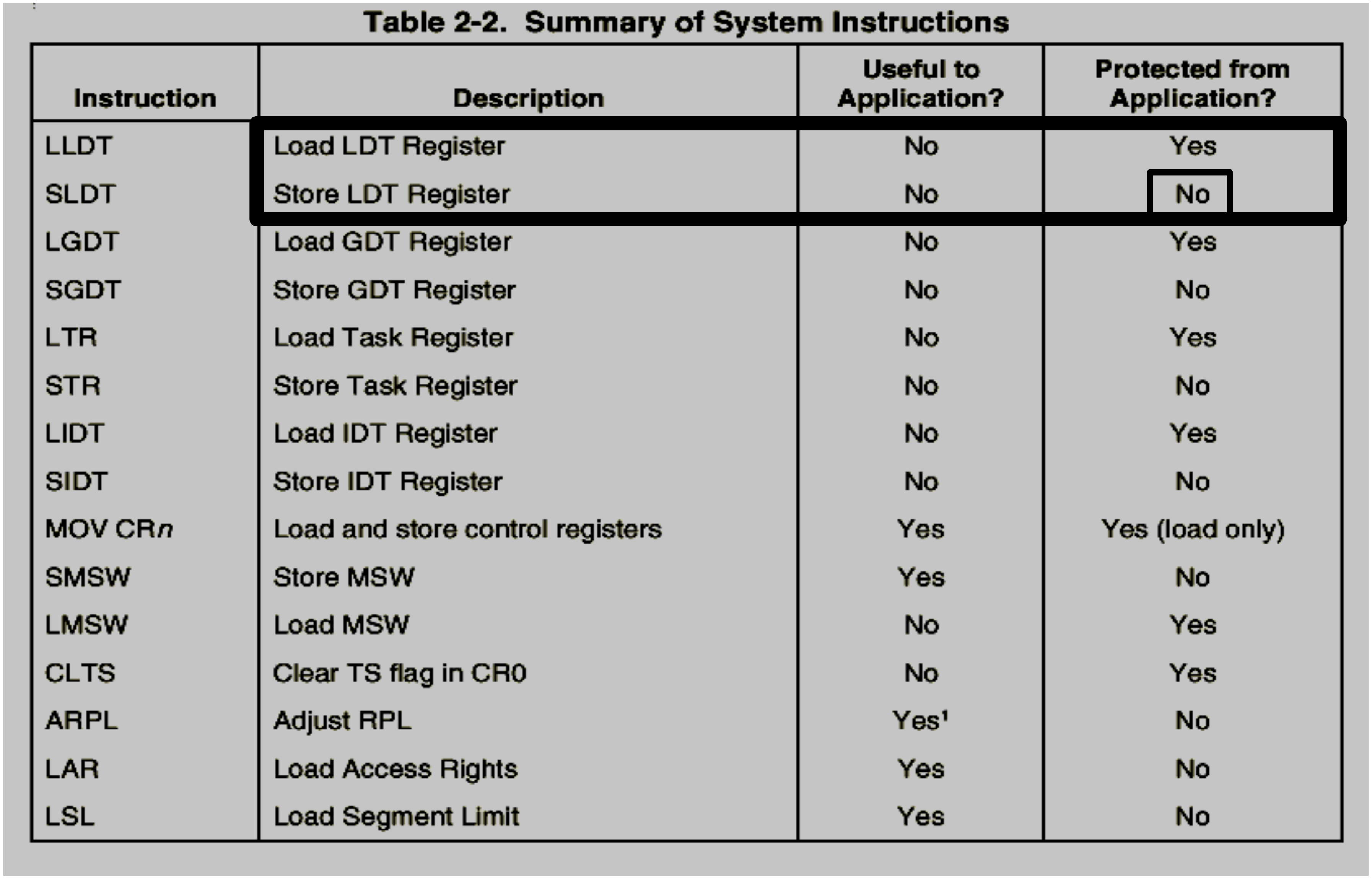


Table 2-2. Summary of System Instructions

Instruction	Description	Useful to Application?	Protected from Application?
LLDT	Load LDT Register	No	Yes
SLDT	Store LDT Register	No	$\boxed{\text{No}}$
LGDT	Load GDT Register	No	Yes
SGDT	Store GDT Register	No	No
LTR	Load Task Register	No	Yes
STR	Store Task Register	No	No
LIDT	Load IDT Register	No	Yes
SIDT	Store IDT Register	No	No
MOV CR*n*	Load and store control registers	Yes	Yes (load only)
SMSW	Store MSW	Yes	No
LMSW	Load MSW	No	Yes
CLTS	Clear TS flag in CR0	No	Yes
ARPL	Adjust RPL	Yes[1]	No
LAR	Load Access Rights	Yes	No
LSL	Load Segment Limit	Yes	No

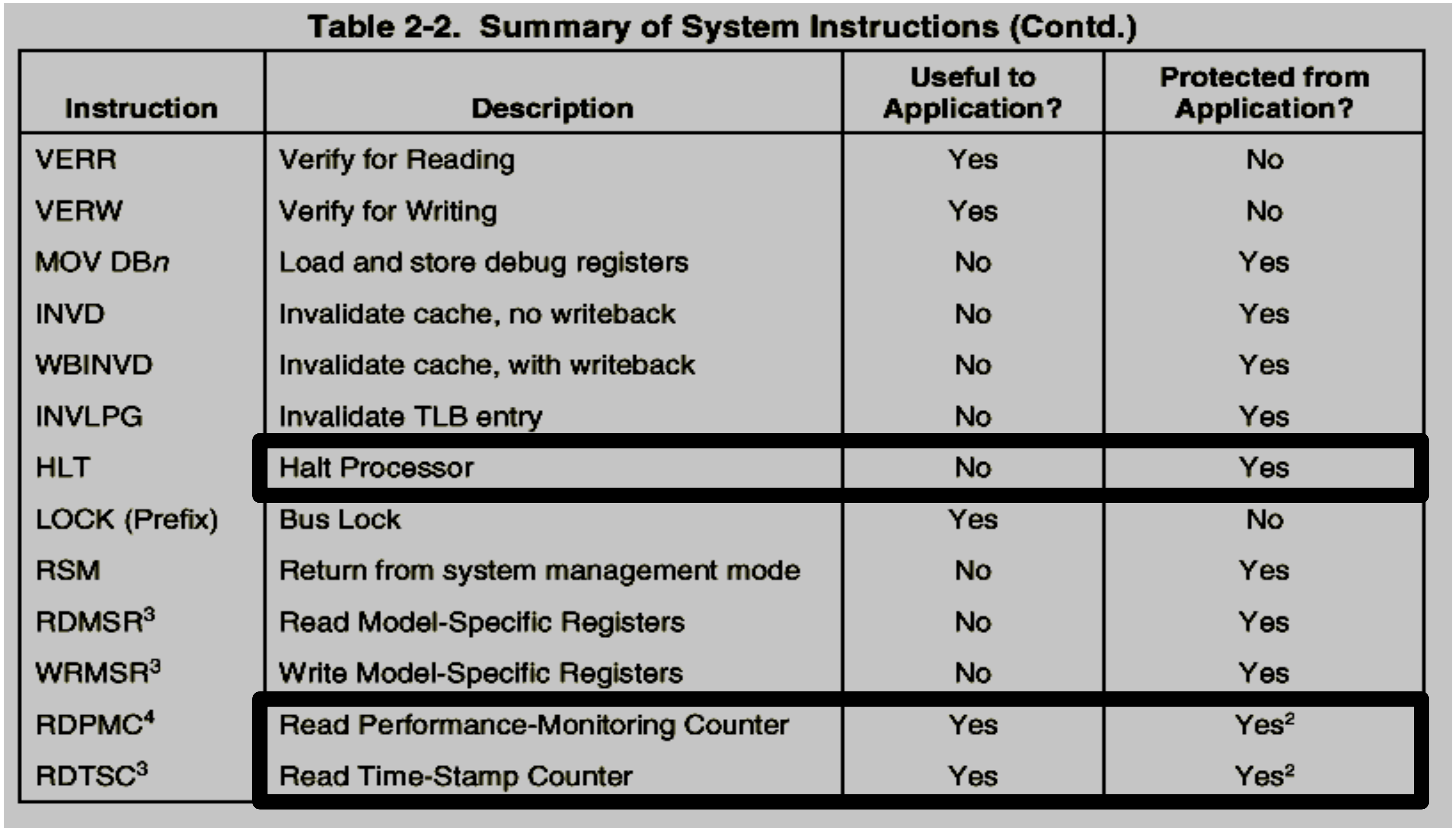
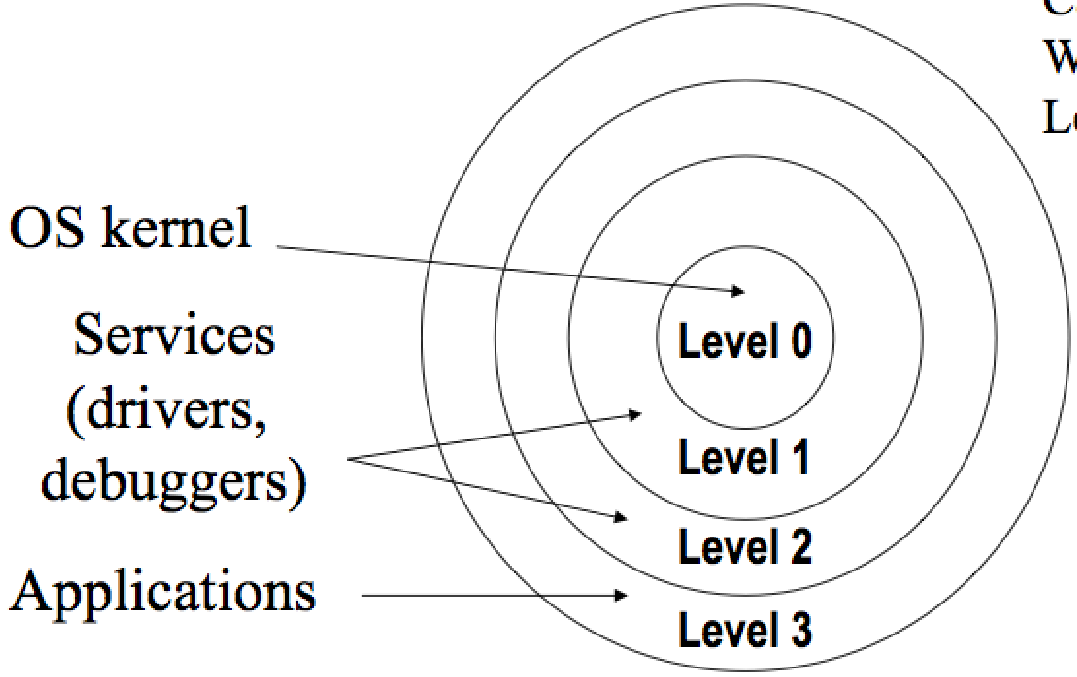


Table 2-2. Summary of System Instructions (Contd.)

Instruction	Description	Useful to Application?	Protected from Application?
VERR	Verify for Reading	Yes	No
VERW	Verify for Writing	Yes	No
MOV DB*n*	Load and store debug registers	No	Yes
INVD	Invalidate cache, no writeback	No	Yes
WBINVD	Invalidate cache, with writeback	No	Yes
INVLPG	Invalidate TLB entry	No	Yes
HLT	Halt Processor	No	Yes
LOCK (Prefix)	Bus Lock	Yes	No
RSM	Return from system management mode	No	Yes
RDMSR[3]	Read Model-Specific Registers	No	Yes
WRMSR[3]	Write Model-Specific Registers	No	Yes
RDPMC[4]	Read Performance-Monitoring Counter	Yes	Yes[2]
RDTSC[3]	Read Time-Stamp Counter	Yes	Yes[2]

IA32 Protection Rings

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



No worries, we will use level 0 and 3

System Calls

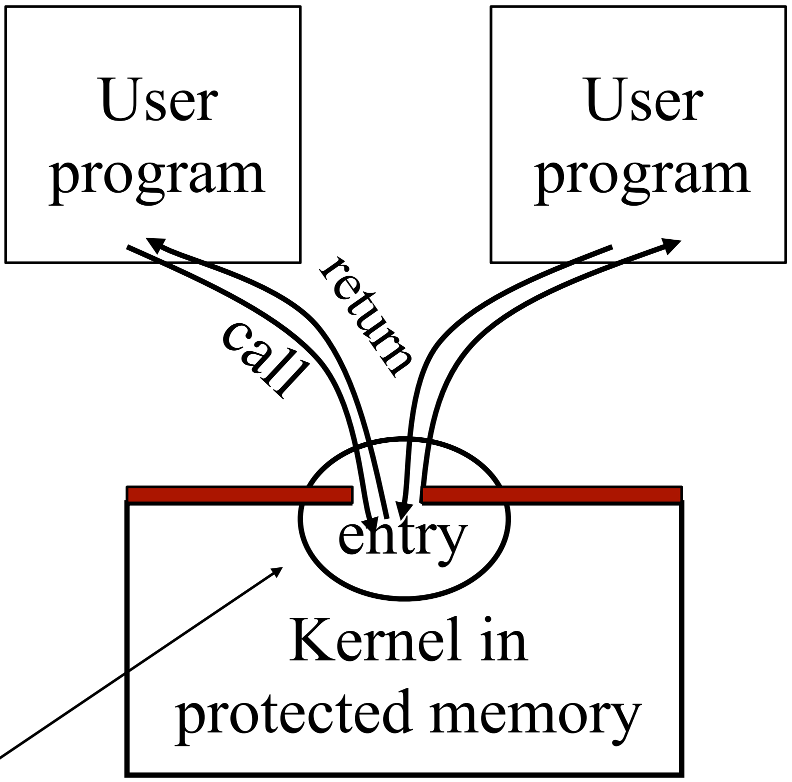
- Operating System API
  - Interface between a process and OS kernel
  - Seen as a set of library functions
- Categories
  - Process management
  - Memory management
  - File management
  - Device management
  - Communication

System Calls

- 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

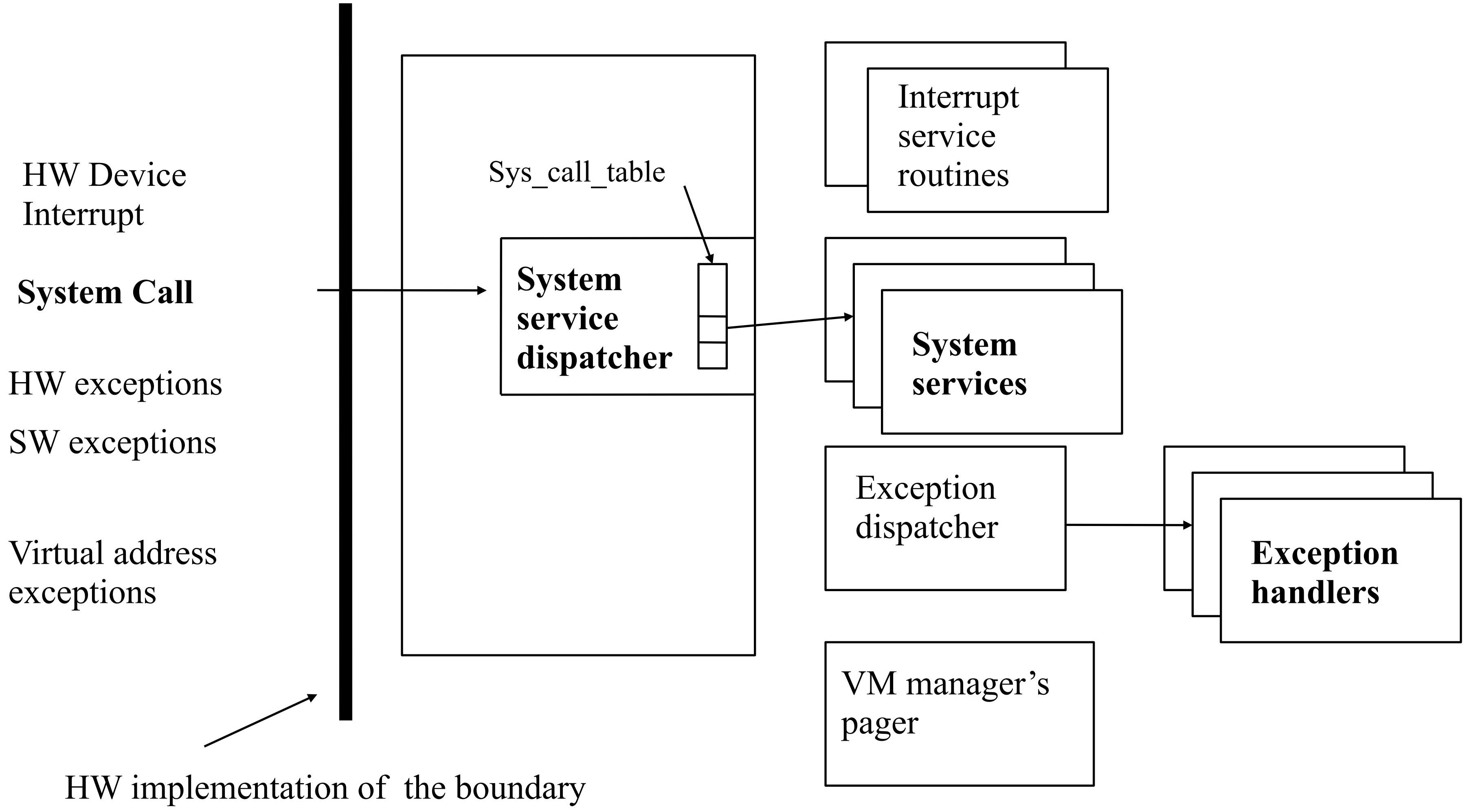


But HOW?

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)

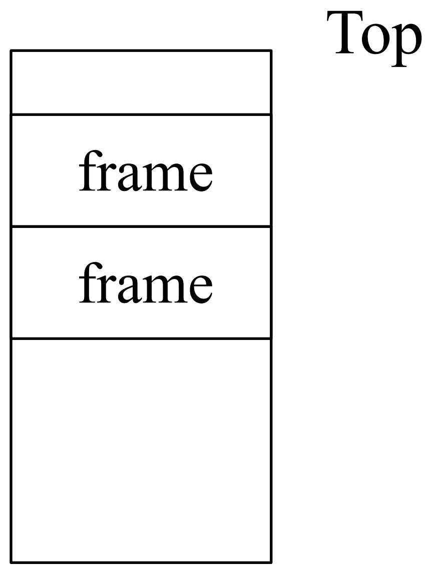
OS Kernel: Trap Handler



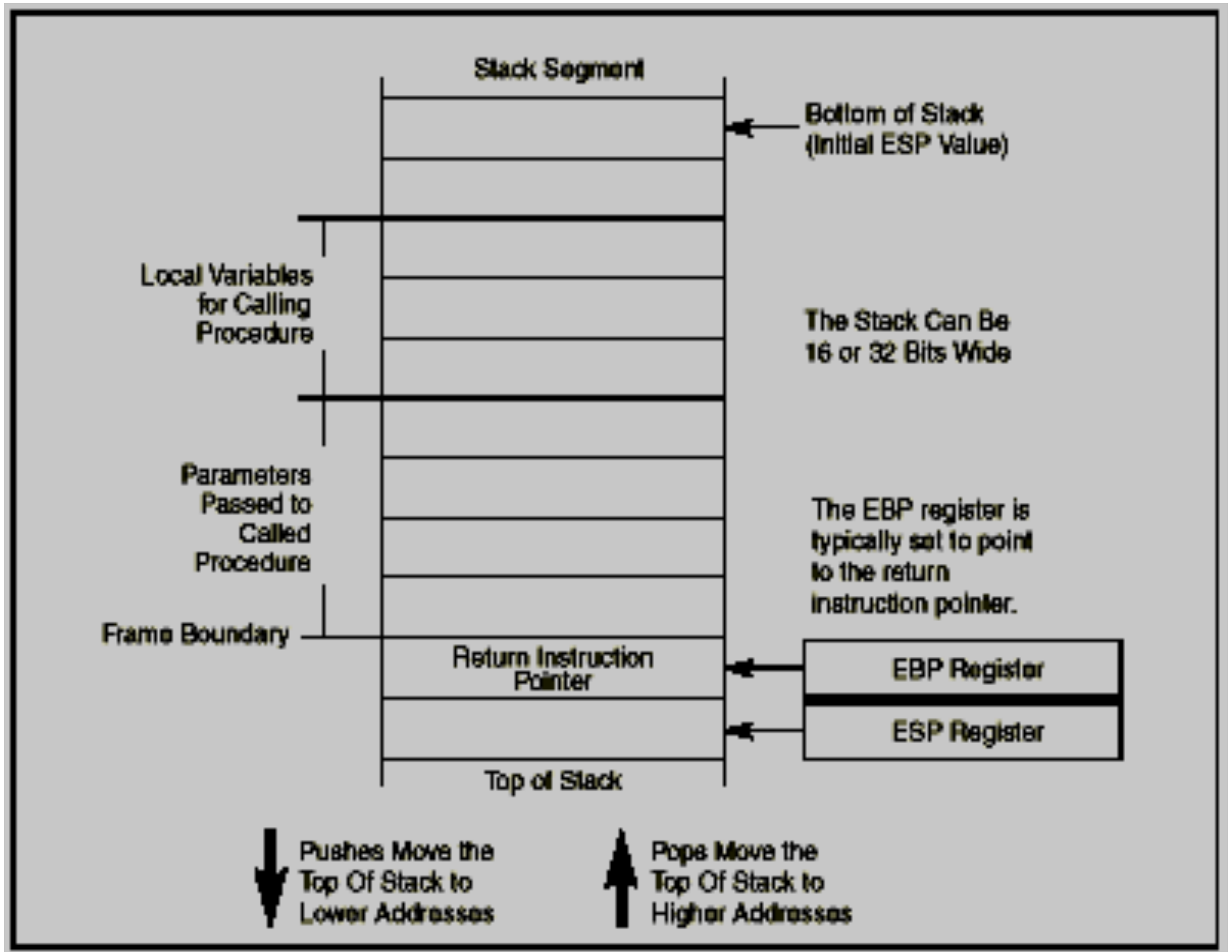
Passing Parameters

- 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



Stack Segment
Bottom of Stack (Initial ESP Value)
Local Variables for Calling Procedure
The Stack Can Be 16 or 32 Bits Wide
Parameters Passed to Called Procedure
The EBP register is typically set to point to the return instruction pointer.
Frame Boundary
Return Instruction Pointer
EBP Register
ESP Register
Top of Stack
Pushes Move the Top Of Stack to Lower Addresses
Pops Move the Top Of Stack to Higher Addresses

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** process: read(fd, buf, size)

```
int read( int fd, char * buf, int size)
{
    move READ to $R_0$
    move fd, buf, size to $R_1$, $R_2$, $R_3$
    int $0x80    HW takes over and IP is set to OS Kernel
    load result code from $R_{result}$
}
```

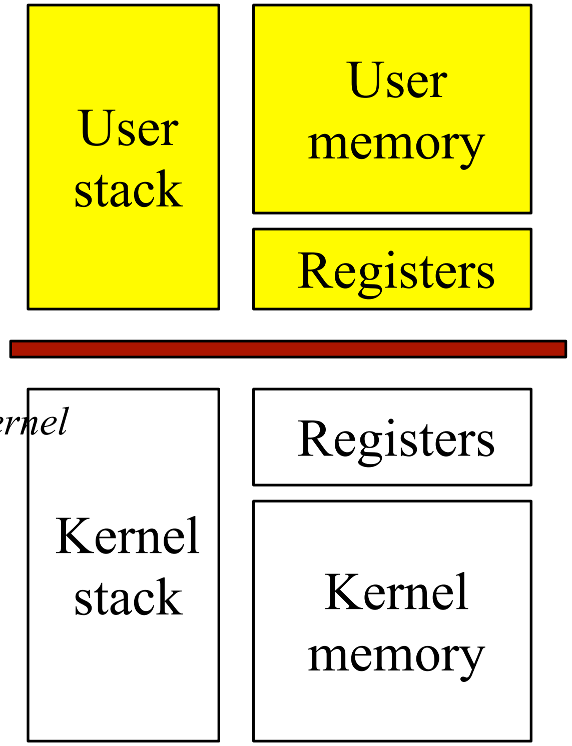
32-255 available to user

Returns here when work is done

Could be an error code

Win NT: 2E

Linux: 80



int 0x80

System Call Entry Point

- Assume passing parameters in registers

SW interrupt

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;
```

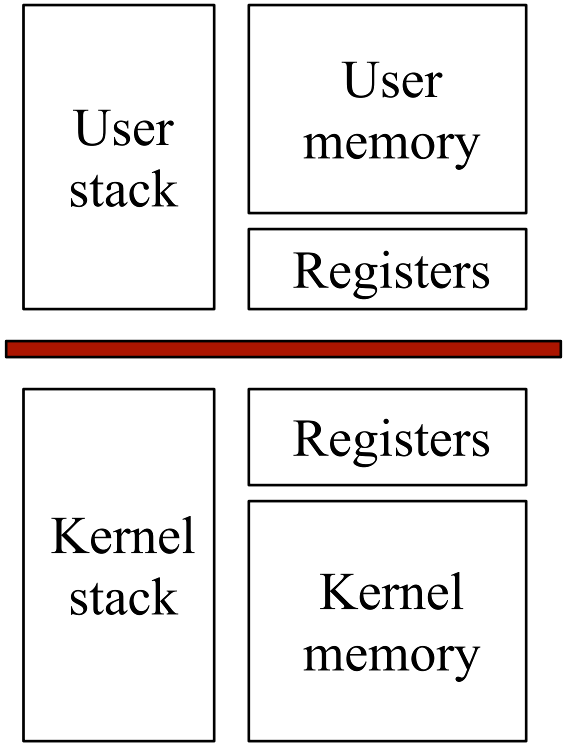
Kernel Mode: Total control. All interrupts are disabled

Change to user mode and return

Put results into buf

Or: User stack

Or: some register



int 0x80

System Call Entry Point

- Assume passing parameters in registers

SW interrupt

(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

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 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

- User process -> kernel
  - syscalls
- Kernel -> user process
  - Kernel is all powerful
    - Can write into user memory
    - Can terminate, block and activate user processes