ECE391
Computer System Engineering

Lecture 19

Dr. Jian Huang

University of Illinois at Urbana- Champaign

Fall 2018

ECE391 EXAM 1

- **EXAM II – November 6; 7:00pm-9:00pm**
  - Seating:
    - ECEB 1002: last name starting with A to W
    - ECEB 1013: last name starting with X to Z
- **Review Session**
  - In Class; Thursday November 1
- **Conflict Exam**
  - Tuesday November 6, 2:00pm; Location: CSL 141
  - By Friday November 2 send email to kalbarcz@Illinois.edu to request Conflict Exam
- **NO Lecture on Tuesday, November 6**

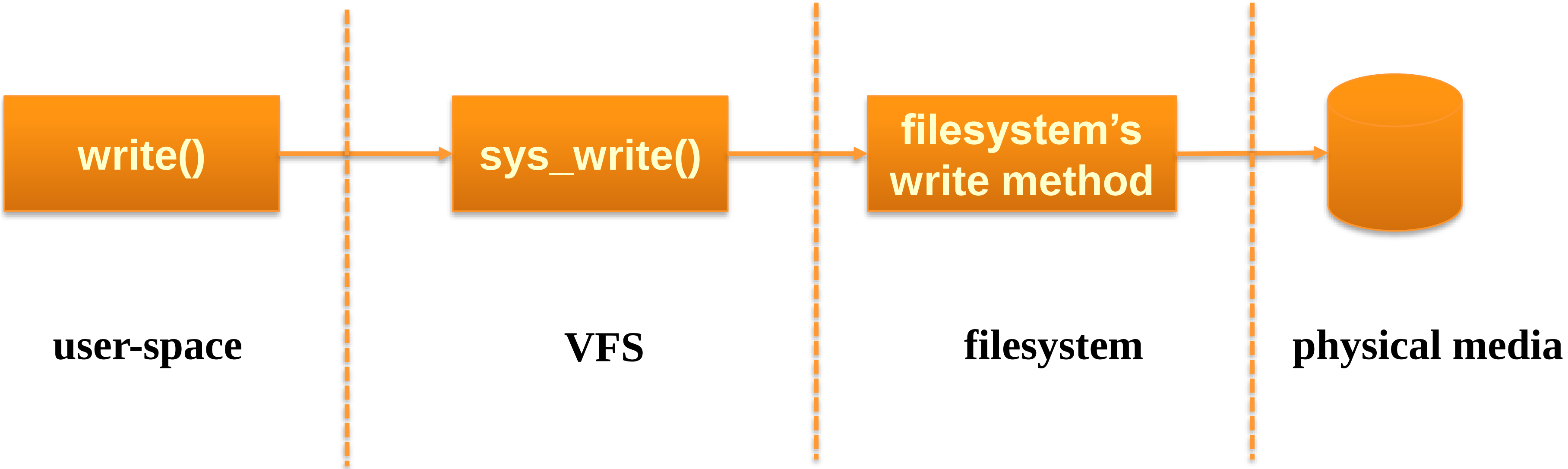
Lecture Topics

- System Calls

System Calls (Definition & An Example)

- Unix/Linux uses systems calls to implement most interfaces between User Mode processes and hardware device

➢ **write (fd, &buf, len)**



Why System Call ?

- Making programming easier (hiding low-level hardware)

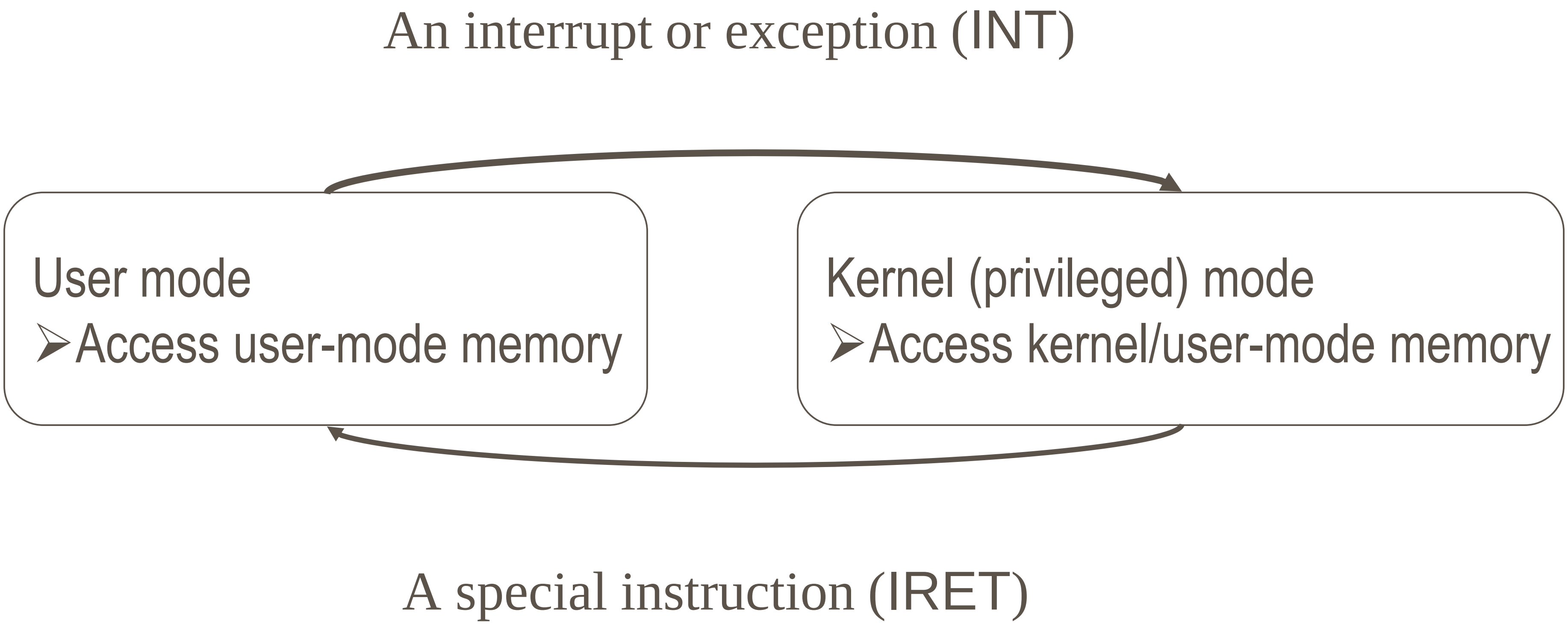
- Increasing system security (checking the correctness)

- Making programs more portable (the same interfaces for different kernel)

System Call Categories

- Process management
- Memory management
- File management
- Device management
- Communication
- ……

User Mode vs. Kernel Mode



An interrupt or exception (INT)

User mode

- ➢Access user-mode memory

Kernel (privileged) mode

- ➢Access kernel/user-mode memory

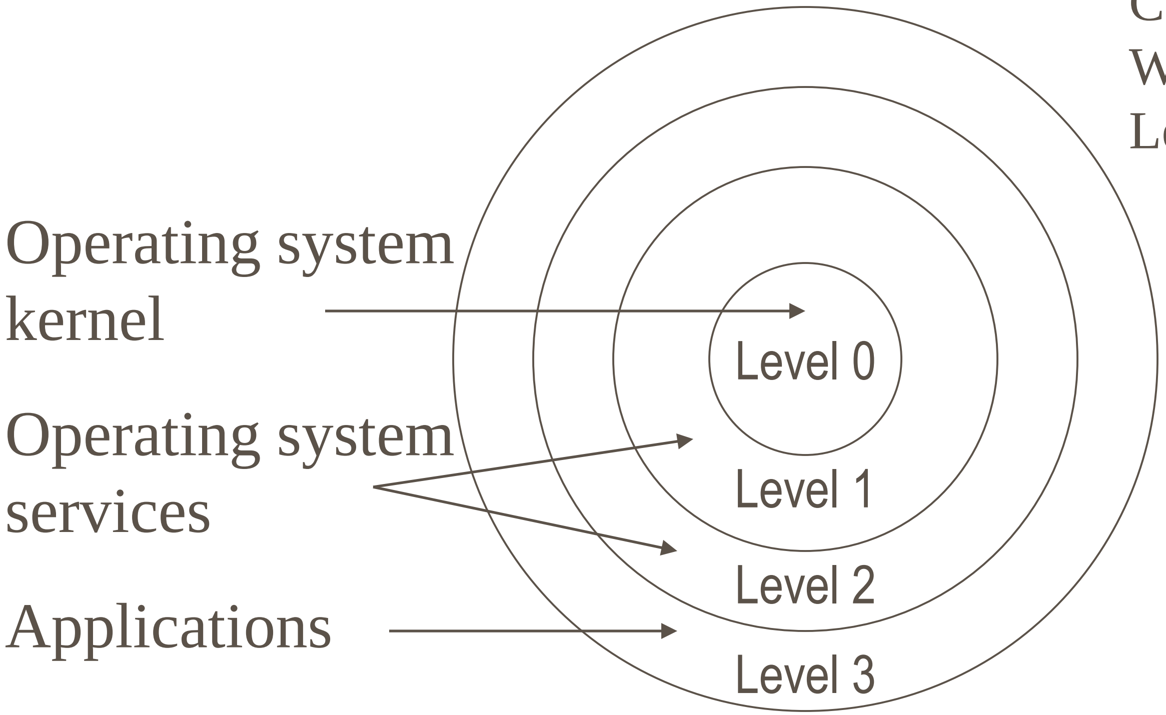
A special instruction (IRET)

Why Privilege Mode ?

- Special Instructions
  - Mapping, TLB, etc
  - Device registers
  - I/O channels, etc.
- Processor Features
  - SSE3 (Streaming Single-Instruction-Multiple-Data Extension 3)
- Device access

X86 Protection Ring

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



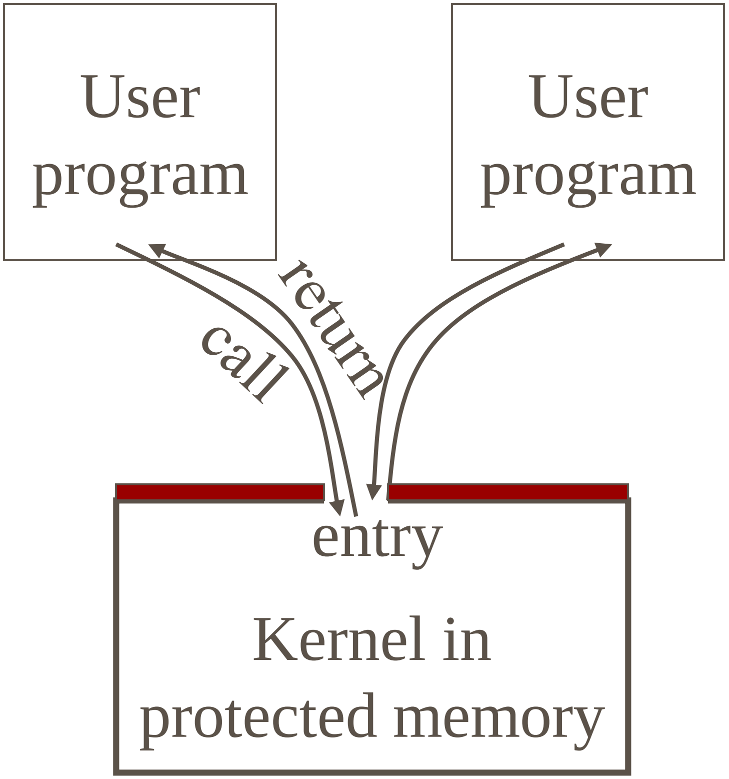
Operating system kernel

Operating system services

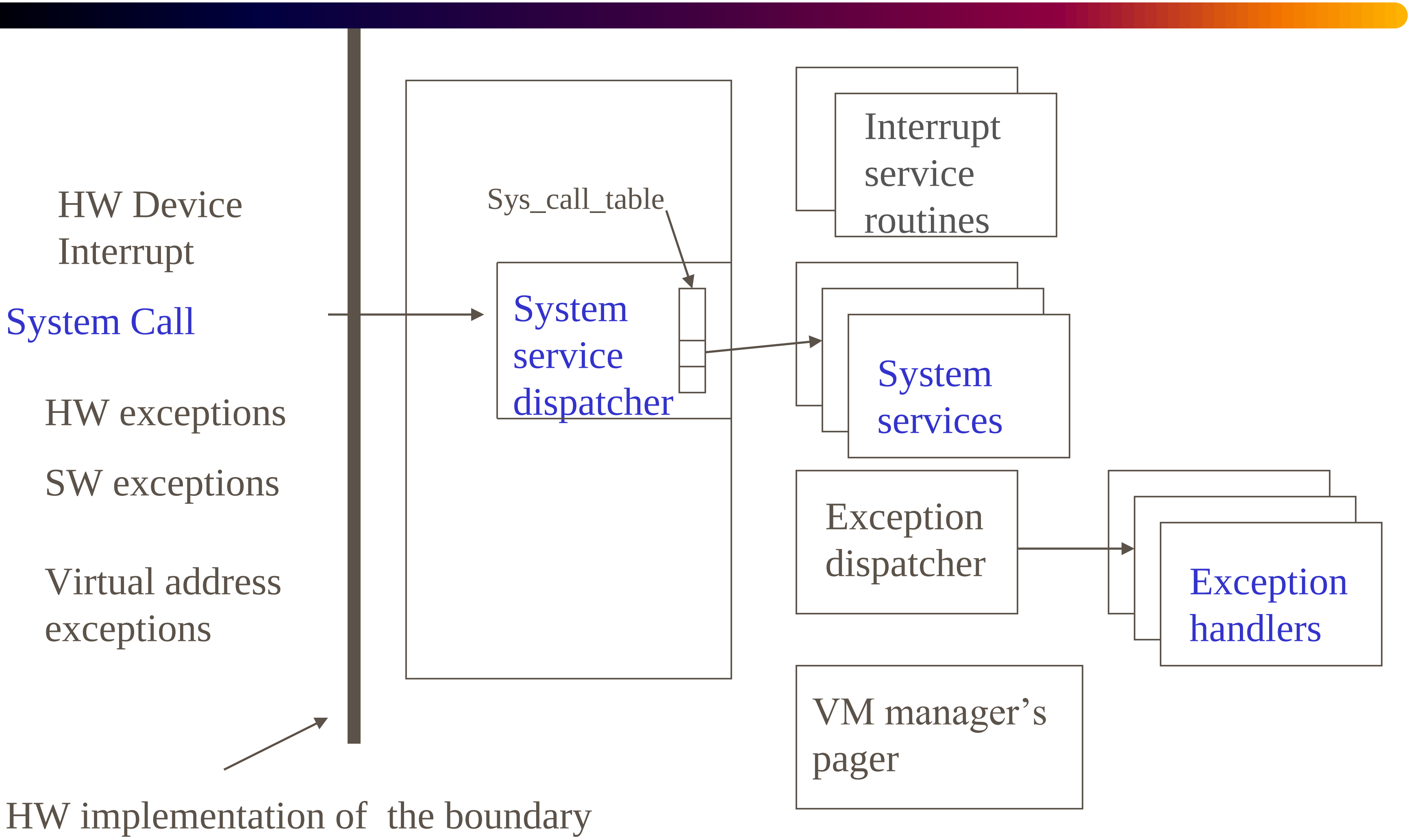
Applications

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



HW Device Interrupt

System Call

HW exceptions

SW exceptions

Virtual address exceptions

Sys_call_table

System service dispatcher

Interrupt service routines

System services

Exception dispatcher

Exception handlers

VM manager's pager

HW implementation of the boundary

System Call Entry Point

- Assume passing parameters in registers

  EntryPoint:

  switch to kernel stack

  save context

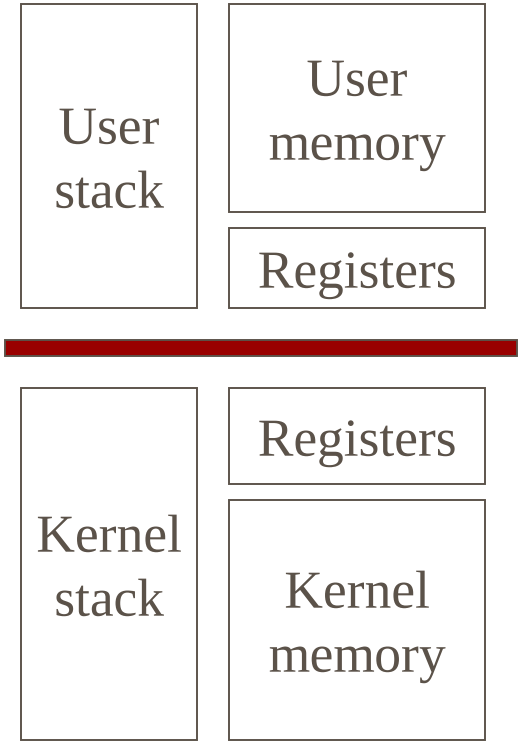
  check register

  call the real code pointed by register

  restore context

  switch to user stack

  iret (change to user mode and return)

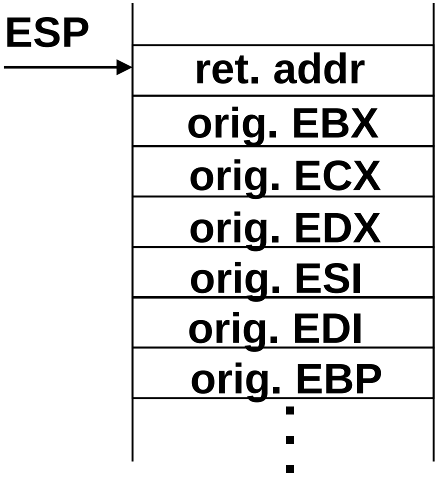


System Call in Linux

- In Linux, all system calls use the following conventions
  - INT 0x80 to invoke (IDT vector 0x80)
  - EAX = system call # (asm/unistd.h)
  - EAX = return value (negative for errors) (asm-generic/errno.h)

System Call

- Vector in IDT is **`system_call`** (in arch/i386/kernel/entry.S)
  - saves registers to stack
  - check for a valid system call #
  - call specific routine using a jump table: **`sys_call_table`** (syscall_table.S, included from entry.S)
  - stack (in kernel level) now appears as shown… (note the argument order)



ESP

ret. addr

orig. EBX

orig. ECX

orig. EDX

orig. ESI

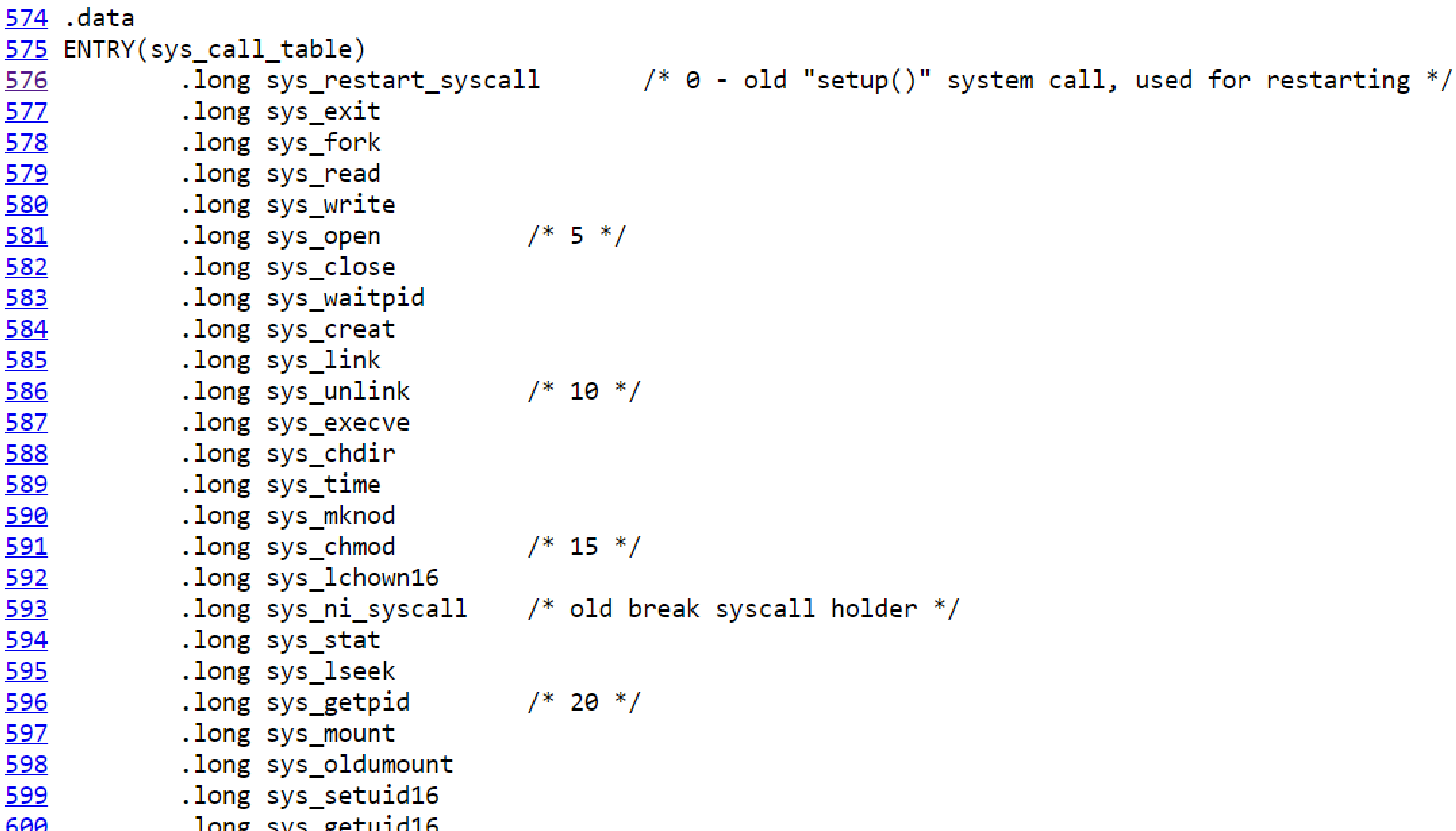
orig. EDI

orig. EBP

⋮

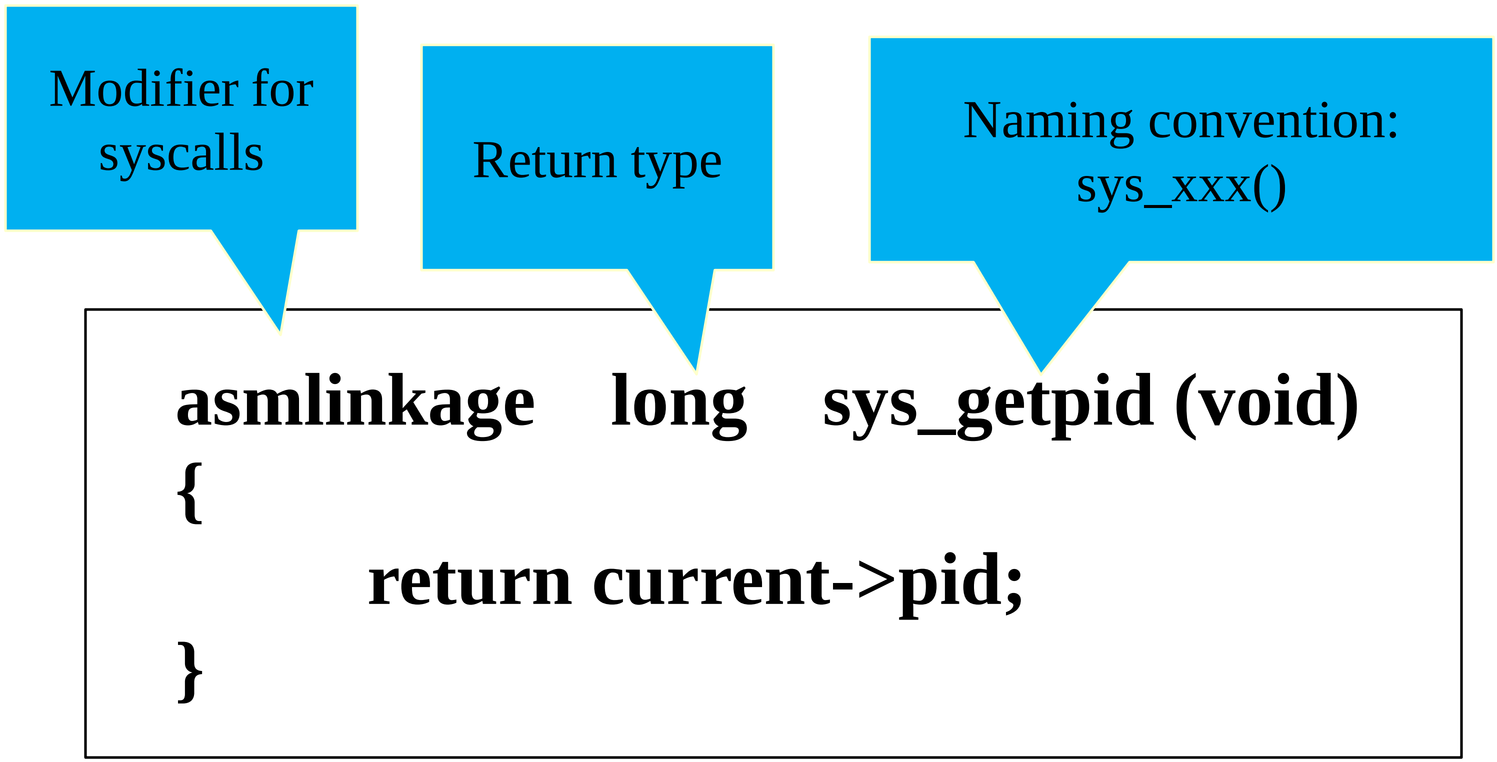
System Call Number

- Each system call is assigned a unique syscall number



```
.data
ENTRY(sys_call_table)
        .long sys_restart_syscall       /* 0 - old "setup()" system call, used for restarting */
        .long sys_exit
        .long sys_fork
        .long sys_read
        .long sys_write
        .long sys_open          /* 5 */
        .long sys_close
        .long sys_waitpid
        .long sys_creat
        .long sys_link
        .long sys_unlink        /* 10 */
        .long sys_execve
        .long sys_chdir
        .long sys_time
        .long sys_mknod
        .long sys_chmod         /* 15 */
        .long sys_lchown16
        .long sys_ni_syscall    /* old break syscall holder */
        .long sys_stat
        .long sys_lseek
        .long sys_getpid        /* 20 */
        .long sys_mount
        .long sys_oldumount
        .long sys_setuid16
        .long sys_getuid16
```

The Internal of A Simple System Call



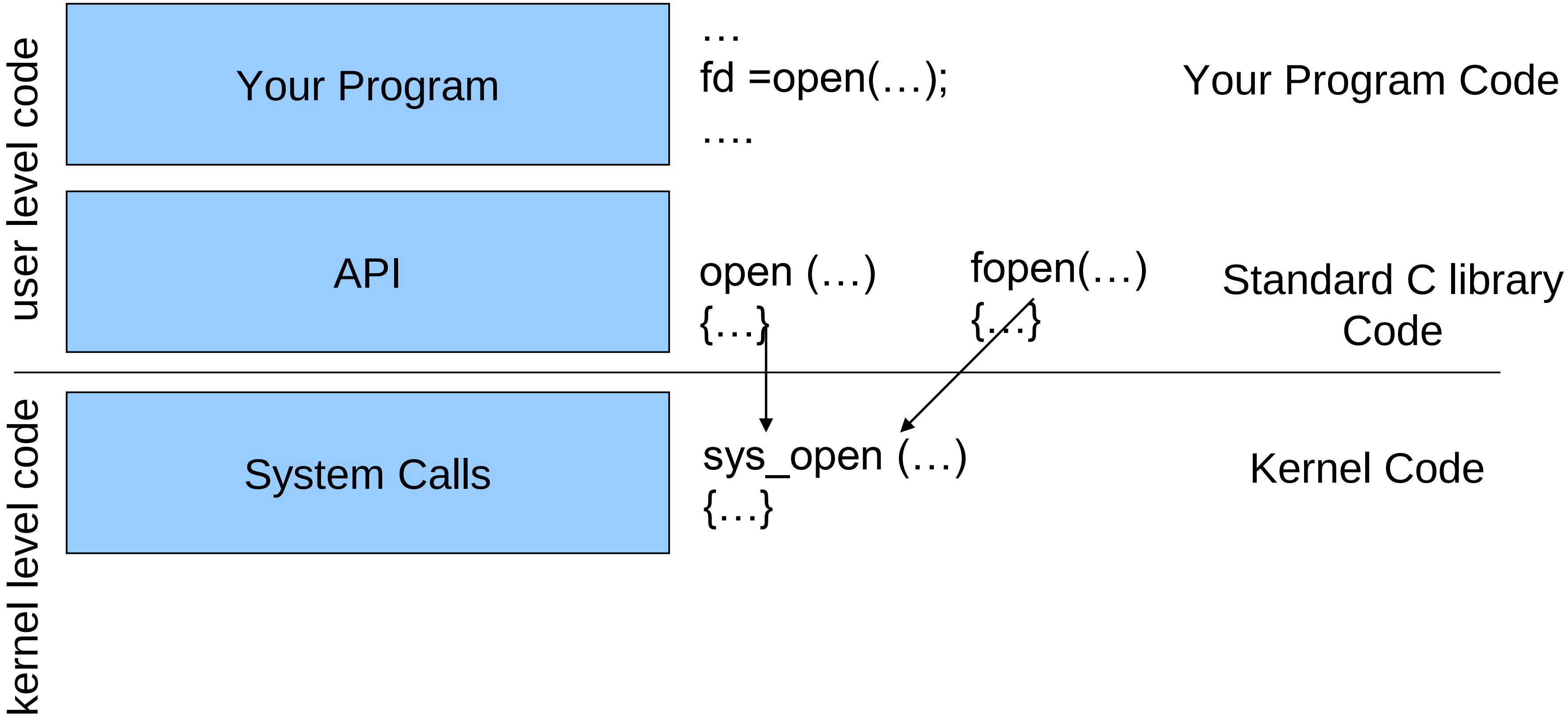
Modifier for syscalls

Return type

Naming convention: sys_xxx()

```
asmlinkage   long   sys_getpid (void)
{
        return current->pid;
}
```

An Example: open()



user level code

kernel level code

Your Program

…
fd =open(…);
….

Your Program Code

API

open (…)
{…}

fopen(…)
{…}

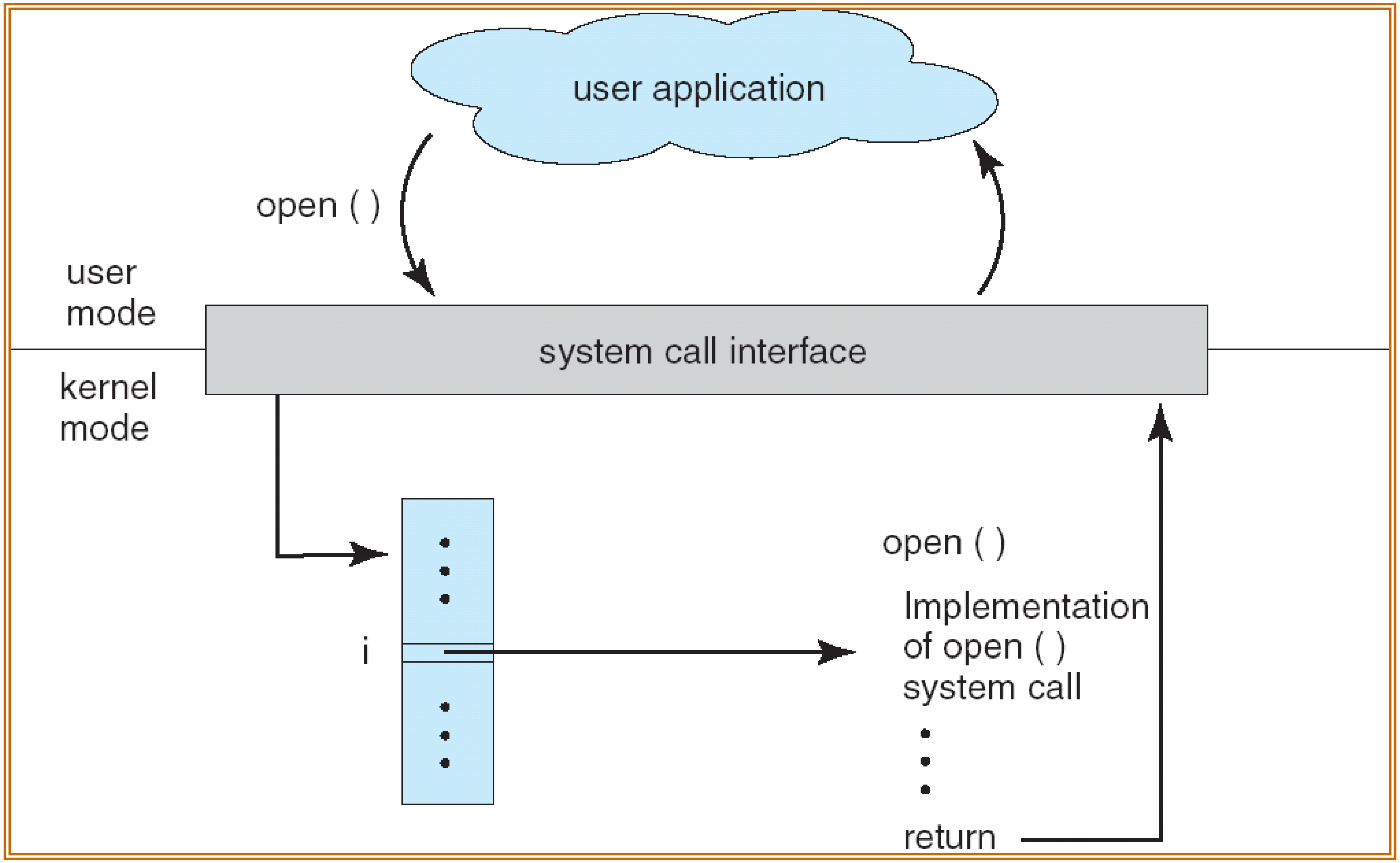
Standard C library Code

System Calls

sys_open (…)
{…}

Kernel Code

An Example: open()



From Lib Code to System Call

- C library code
  - arranges the arguments
  - before performing the system call
  - for example, open and sys_open

```
int open (const char* name, int flags, int mode);
```

```
asmlinkage long sys_open (const char* name,
                          int flags, int mode);
```

Open System Call Wrapper

```
open: pushl  %ebx                  # save EBX to stack   collee saved register
      movl   0x10(%esp),%edx       # EDX   mode
      movl   0x0C(%esp),%ecx       # ECX   flags
      movl   0x08(%esp),%ebx       # EBX   name
      movl   $0x05,%eax            # open is system call #5
      int    $0x80                 # do the system call
      cmpl   $0xFFFFF001,%eax      # -1 to -4095 are errors
      jb     done                  # others are valid descriptors
      xorl   %edx,%edx             # negate error number
      subl   %eax,%edx
      pushl  %edx                  # save EDX (caller-saved)
      call   __errno_location      # get pointer to errno
      popl   %ecx                  # pop error number into ECX
      movl   %ecx,(%eax)           # save error number in errno
      orl    $0xFFFFFFFF,%eax      # return -1
done: popl   %ebx                  # restore EBX from stack
      ret
```

Open System Call Wrapper

- Call to **`open`** translates to a call to **`sys_open`** inside the kernel

- Before going on, let's see error handling path
- Recall that, on error, C library
  - returns -1 (for many calls, including **`open`**)
  - stores error code in **`errno`** variable

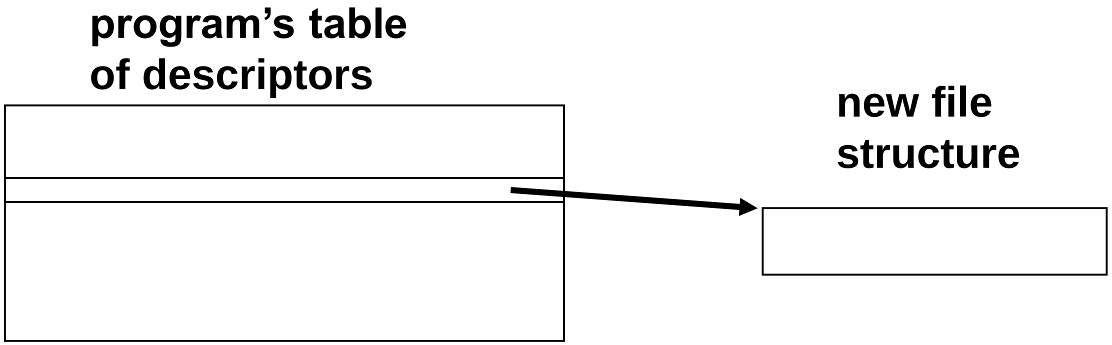
- But library code is relocatable, so **`errno`** address is not fixed

Open System Call Wrapper

- In dynamically-linked code
  - code addresses generally linked through jump tables
  - data addresses often use the trick
    - make a “fake” call
    - call pushes return address onto stack
    - return address is located at fixed offset from static variable
    - load from stack and add offset to obtain pointer to variable

More on sys_open

- **`sys_open`** (in fs/open.c) invokes **`do_sys_open`**
- **`do_sys_open`** does the following
  - get a free file descriptor
  - open the file (using **`do_filp_open`**)
  - attach the file to the descriptor

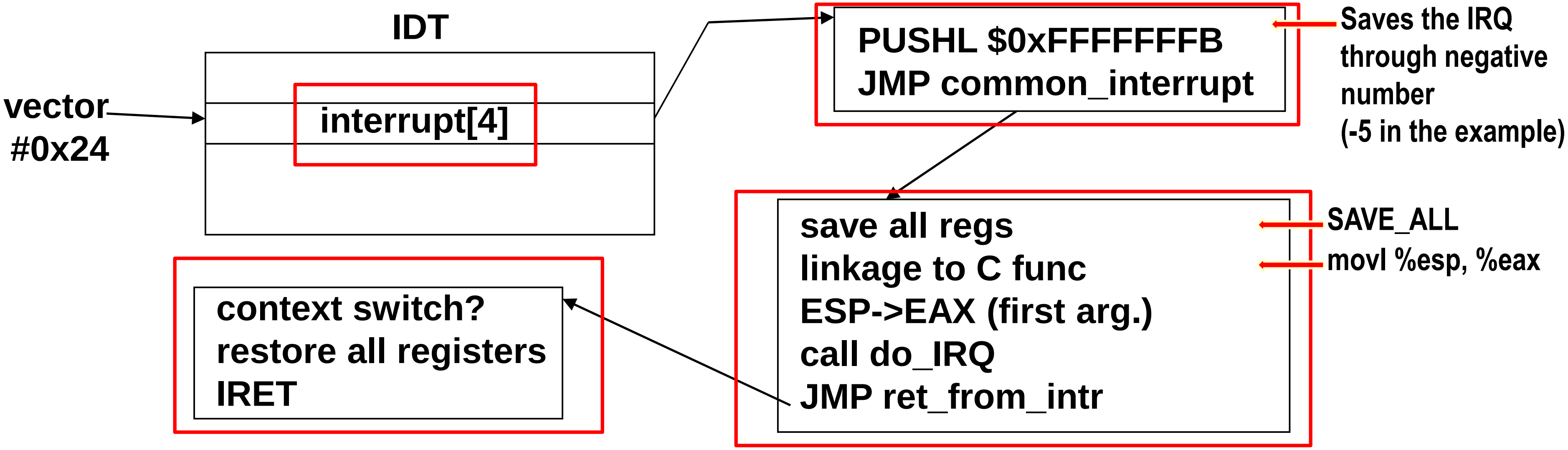


Open System Call Wrapper

- **`filp_open`**
  - check access rights
  - find VFS mount
  - get a file structure from a free list
  - open directory entry (using **`__dentry_open`**, below)

- **`__dentry_open`**
  - fill in file structure
  - if **`fops`** structure is not NULL && **`fops->open`** entry is not NULL, call **`fops->open`**

Interrupt Invocation



IDT

vector #0x24

interrupt[4]

```
PUSHL $0xFFFFFFFB
JMP common_interrupt
```

Saves the IRQ through negative number (-5 in the example)

```
save all regs
linkage to C func
ESP->EAX (first arg.)
call do_IRQ
JMP ret_from_intr
```

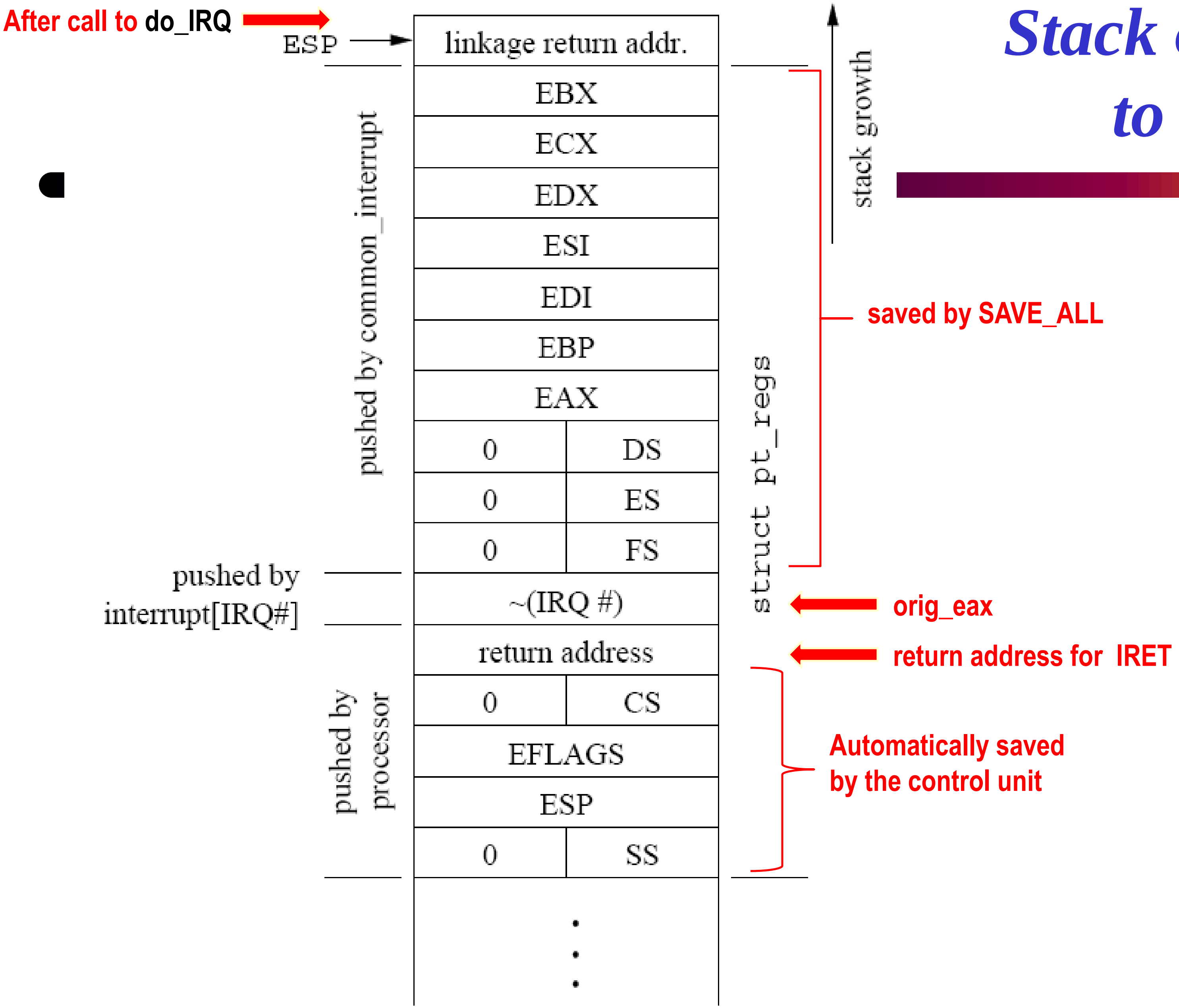
SAVE_ALL

movl %esp, %eax

```
context switch?
restore all registers
IRET
```

- Why funnel all IRQs into one piece of code(instead of producing one function per IRQ) ?
  - kernel code/size versus speed tradeoff
    - perhaps no big deal for 16
    - keep in mind that you’re saving a tiny number of instructions
    - APIC used by most SMPs has 256 vectors

Stack on Entry to do_IRQ



After call to do_IRQ

ESP

linkage return addr.	
EBX	
ECX	
EDX	
ESI	
EDI	
EBP	
EAX	
0	DS
0	ES
0	FS
~(IRQ #)	
return address	
0	CS
EFLAGS	
ESP	
0	SS

saved by SAVE_ALL

orig_eax

return address for IRET

Automatically saved by the control unit