ECE391
Computer System Engineering

Lecture 19

Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

Fall 2022

Lecture Topics

- System Calls

ECE391 EXAM 2

- **EXAM 2 – November 1 (Tuesday)**
  - **Time: 7:00pm to 9:00pm**

- **Conflict Exam**
  - Deadline to request conflict exam:

    Thursday, October 27, by 5:00pm

    (by email to: *kalbarcz@Illinois.edu*)

ECE391 EXAM 2

- **Exam 2 Review Session in Class :**
  - During the Lecture Time on Thursday, October 27
- **Exam 2 Review Session in collaboration with HKN**
  - **TBD**

ECE391 EXAM 2: Topics

- **Topics covered by the EXAM 2**
  - Material covered in lectures
    (Lecture12, Lecture13, Lecture 14, Lecture 15, Lecture 16, Lecture 17, Lecture 19)
  - MP2 (ModeX, Tux controller)
  - MP3.1 and MP3.2 (file system)
  - Material covered in discussions

NOTE: **Scheduling NOT** included in Exam 2

- **NO Lecture on Tuesday, November 1**

System Calls

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

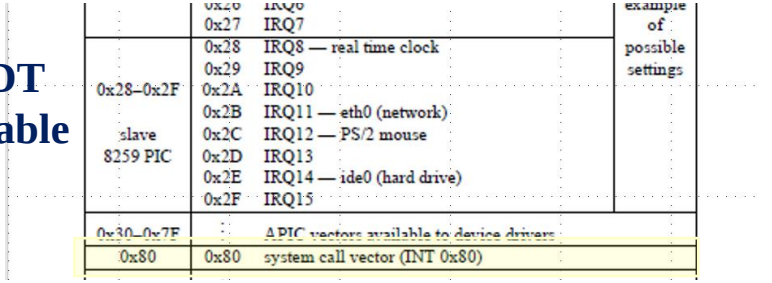
- System calls need protection boundary (user to kernel mode)
  - supported by ISA
  - need assembly linkage

System Call

- 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

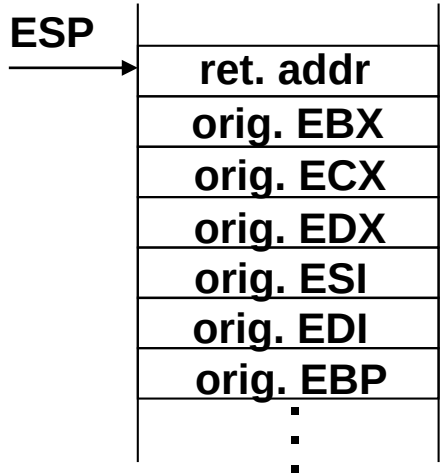
- Vector in IDT is **`system_call`** (in arch/i386/kernel/entry.S)



  - saves registers to stack
  - check for a valid system call #
  - call specific routine using a jump table:

```
call *sys_call_table(0, %eax, 4)
```

  - stack (in kernel level) now appears as shown… (note the argument order)



  - on completion of the system call handler
    - get the return code from **`eax`**
    - restore all the registers and **`IRET`** to resume the User Mode process

Example: 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 (1)

```
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 (2)

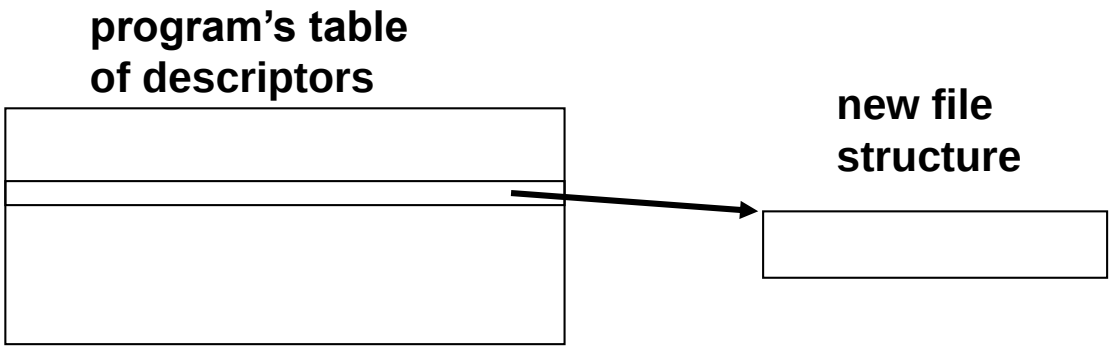
- Find static data (in this case ***errno***) in a relocatable library
  - the library may be loaded to different parts of memory from the program's point of view
  - the address must be adjusted for the current execution

```
__errno_location :
        call    getIP                   # EAX   <-  raddr
raddr:  addl    $errno – raddr, %eax    # adjust return value
        ret
getIP:  movl    (%esp), %eax            # read return address into eax
        ret
```

- 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, ***errno*** in our case

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



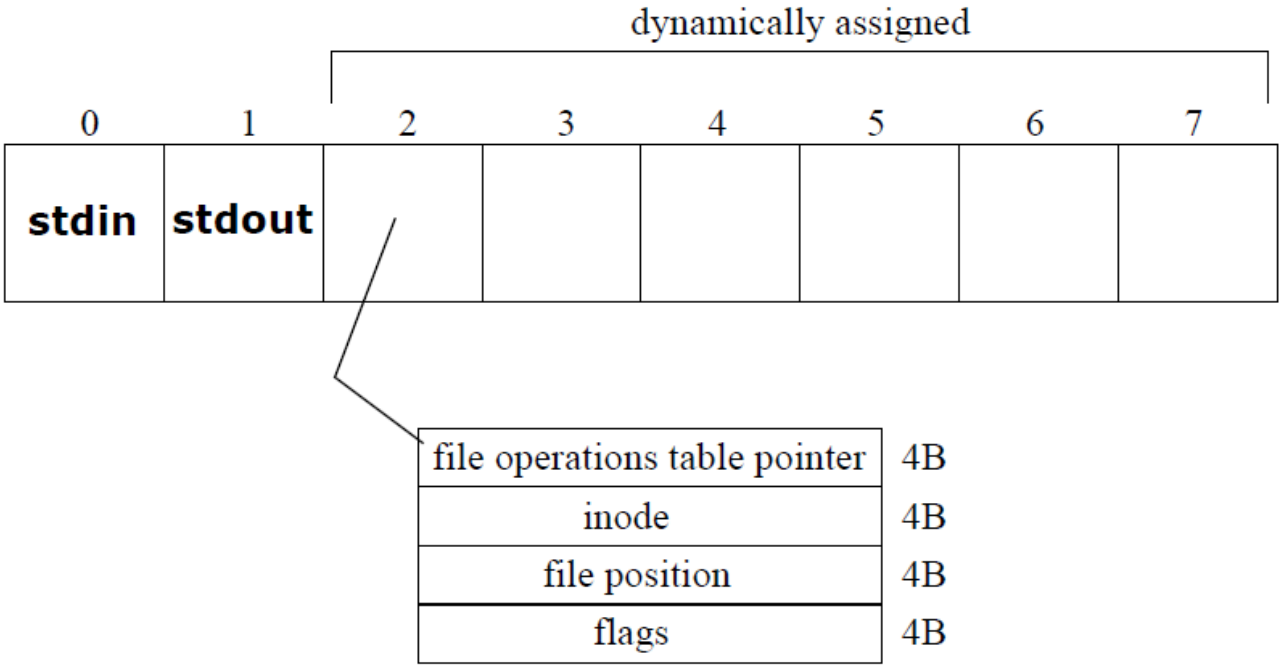
Open System Call Wrapper

- **`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`**

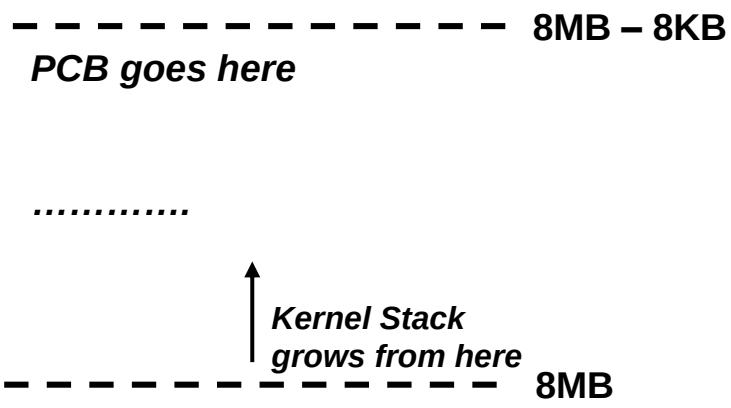
MP3 File System Abstractions

- **Each task can have up to 8 open files**
- **Open files are represented with a file array (in a Process Control Block; PCB)**
- **File array is indexed by *a file descriptor***



MP3: Process Control Block (PCB)

- Process Control Block (PCB) is analogous to a task struct (process descriptor) in Linux
- PCB for each process is stored above the kernel stack corresponding to this process



For more than one process
(e.g., shell running shell running ls)